# Code Flow Course

ThinkingElixir.com

Download Reference

by Mark Ericksen

This document is intended as a reference resource after course completion.

For the best learning experience and access to the accompanying download materials, enroll in the course.

# Code Flow Introduction

The Code Flow course deals with "control flow". This includes **branching logic**, **looping**, and **error handling**. These form the foundation of most application code. This is how you actually make your code **do stuff**. As a developer, your initial thought may be, "I already understand control flow!"

The following 3 factors impact a lot more than you may realize:

- Pattern matching
- Immutable data
- Functional programming

Each factor above has a huge impact on *how* you control the flow of your Elixir applications. Pattern matching opens up new coding patterns that did not exist for you before. Language features like the Pipe Operator and the `with` statement create new opportunities. Immutable data means the way you make changes is different.

## Changing your thinking

When I was learning Elixir, I learned the *mechanics* of the new features quickly, but it took me longer to change my *thinking*. It took longer to figure *why* I would want to use approach A over B. This course is going to save you so much time by helping you understand the *when* and *why* up front.

We learn best by *doing*. A downloadable project gives you a place to play, problems to solve, and tests to help you know you're getting it. This saves you time and avoids struggle.

## Prerequisites

1. Elixir is installed
2. Code editor setup for Elixir – Need a recommendation?
3. Comfortable using IEx (Interactive Elixir shell)
4. Understand the basic Elixir data types
5. Familiarity with pattern matching in Elixir

If you can't check-off all of the prerequisites then you risk having a hard time following in this course. Please review and complete the Free Pattern Matching course which covers all of this. Then come back!

## Our goals

You know how to use existing functions in Elixir and how to write your own. Now, it's time to figure out how to put the pieces together to solve actual problems.

- Solve problems that requires multiple steps
- Learn how to "flow" code together in an elegant way
- Learn a new way to loop in your code
- Get you productive and creating or contributing more quickly

## What we cover

To reach our goals, we will cover the following topics.

- The pipe operator `|>` and creating pipelines
- The Railway Pattern
- Writing `with` statements
- Branching with a `case` statement
- How to loop without a "for" loop or a "while" statement.
- Exception handling
- The `if` and `cond` statements

Of course, we'll also pick up a lot more as we go!

## Download reference

After completing the course you can download most of the course content as a PDF. The PDF is a ready, portable reference you can easily search and refer back to as needed.

## Let’s go!

This is going to be fun and challenging! I’m excited to share these things with you and help you become productive faster.

Let’s get started!

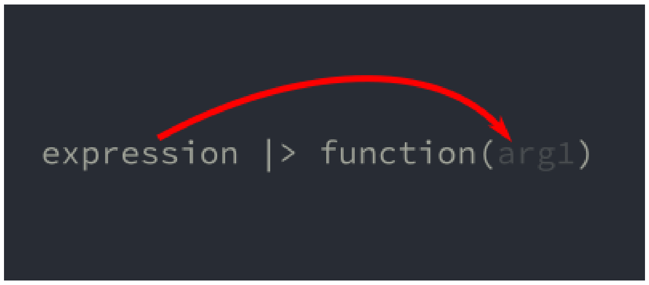
# Pipe Operator

### "Hello" |> say()

The pipe operator is made up of these two characters: `|>` . There are two inputs to this operator. A left and a right side. The left side is an expression (or data) and the right side is a function that takes at least 1 argument.

```
expression |> function
```

The pipe operator performs a special job of putting the value of the expression on the left into the first argument of the function on the right.



```
expression |> function(arg1)
```

This may not seem like much at first, but this is a very powerful and helpful feature. Let’s see it in action to start to understand what it does for us. The function `String.upcase/1` takes a string and returns a new string with all the characters converted to upper case.

```
String.upcase("testing")
#=> "TESTING"
```

This can be re-written using the pipe operator like so…

```
"testing" |> String.upcase()
#=> "TESTING"
```

The pipe operator pushes the `"testing"` string in as the first argument to `String.upcase/1` .

## Making a Pipeline

The real power with this operator is when we start chaining pipe operators together to create a “pipeline”. The result of one function can be piped into another function and then another.

Let’s look at a simple pipeline example:

```
"   Isn't that cool?   " |> String.trim() |> String.upcase()
#=> "ISN'T THAT COOL?"
```

> The real power with this operator is when we start chaining pipe operators together to create a “pipeline”.

If we re-write this without the pipe operator it looks like this:

```
String.upcase(String.trim("Isn't that cool?"))
#=> "ISN'T THAT COOL?"
```

This is functionally equivalent. However, in order to read this code, we have to work from the inside out. Mentally parsing the function calls and the parenthesis to find the innermost argument and then start reading out from there. It’s more mental work. The pipe operator makes the process much more readable and intuitive!

The other way to re-write this is to use new variables at each step. It might look like this:

```
data = "Isn't that cool?"
trimmed = String.trim(data)
upcased = String.upcase(trimmed)
#=> "ISN'T THAT COOL?"
```

This version reads easier because you still follow each step from the top going down instead of a sequence going from left to right. However, the pipe operator manages passing the return value from one step on to the next function for us, so we don’t need to keep binding it to a new variable.

**Thinking Tip: Pipes and IEx**

The pipe operator is limited in IEx. Because IEx evaluates each line as it is entered, IEx can’t tell when a pipeline is finished or not. This means for simple code in IEx, we must write a pipeline with the pipe operator inline. If we define a module and a function then it will be evaluated all together and works as we’d expect.

## Pipelines Flow from Top to Bottom

A pipeline is most readable when we write it vertically. We start with the data being passed in and build up our pipeline from there. For these to work in IEx, it must be declared in a module and a function. Open up your favorite text editor and write the code so you can paste into IEx. Here’s one you can copy to play with for a quick start.

```
defmodule PipePlay do

  def test do
    "   Isn't that cool?   "
    |> String.trim()
    |> String.upcase()
  end

end

PipePlay.test()
#=> "ISN'T THAT COOL?"
```

Do you see how it reads better? The pipeline can keep going by adding more steps.

## IO.inspect/2

This is a good time to introduce a great debugging and insight tool. The IO module manages writing to standard out and standard error. We've already used `IO.puts/2`, this is where it is defined.

The function we want to look at now is `IO.inspect/2`. This does the following:

- Creates a string representation of whatever data structure we give it
- Writes it to `stdout` (which prints it to the console)
- Returns whatever value we passed in to it

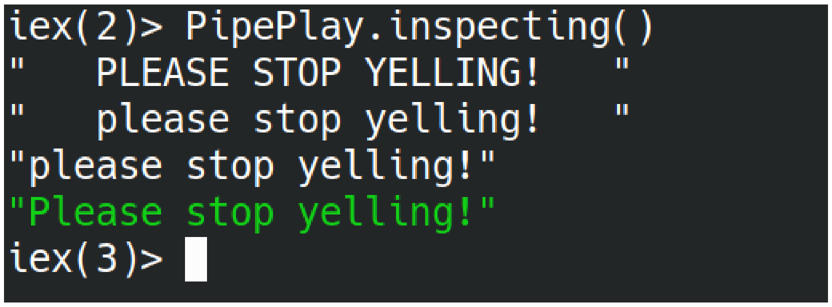
These features make it a valuable function for peering into a pipeline and seeing the transformations being performed.

We'll use a similar top-down pipeline that processes a string. Rather than only seeing the final result, we can sprinkle some `IO.inspect` calls directly into the pipeline. Since they pass on the data they were given, it is safe to insert almost anywhere. The side-effect that `IO.inspect` creates is to write the data to the console. This works really well for watching each step of the pipeline as it performs the transformations.

Try it out!

```
defmodule PipePlay do

  def inspecting do
    "   PLEASE STOP YELLING!   "
    |> IO.inspect()
    |> String.downcase()
    |> IO.inspect()
    |> String.trim()
    |> IO.inspect()
    |> String.capitalize()
  end

end
PipePlay.inspecting()
#=> "   PLEASE STOP YELLING!   "
#=> "   please stop yelling!   "
#=> "please stop yelling!"
#=> "Please stop yelling!"
```

Running this in an IEx shell you might notice the difference in colors. The `IO.inspect` output is written to the terminal's standard out. There is no text coloring added to this. The *result* of the function is recognized as a string and output with coloring. You might see something like this...



```
iex(2)> PipePlay.inspecting()
"    PLEASE STOP YELLING!    "
"    please stop yelling!    "
"please stop yelling!"
"Please stop yelling!"
iex(3)>
```

### Thinking Tip: Use IO.inspect Anywhere

IO.inspect/2 works really well in a pipeline because it writes to stdout but returns the thing passed in. This can be used in many situations (not just pipelines) to get insight into what something is doing. Try it out and use it liberally!

The easiest way to sprinkle it around is to add the pipe and an IO.inspect call like |> IO.inspect() after something you want insight into. You don't need to re-write code like IO.inspect(thing_to_peek_at) , just use the pipe!

Example: thing_to_peek_at |> IO.inspect()

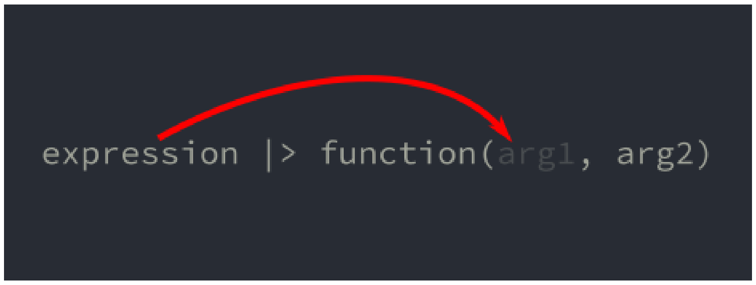
## Understanding IO.inspect output

IO.inspect/2 doesn't change the result of the function, it is a function who's side-effect is to write data to the console.

IO.inspect/2 is best used during development and testing as a debugging and insight tool. You don't want to leave this in production code. Assuming your application's logs are captured and used, any calls to IO.inspect will create "console noise" that doesn't conform to the logging pattern of your application. There are better ways to write data to your logs if that is your goal.

## What about multiple arguments?

What about a function that doesn't just take 1 argument? What if it takes 2 or more? The pipe operator handles putting the value into the *first* argument. Our call would be written as starting with the 2nd argument.



The function String.replace/3 takes 3 arguments. The first is the string to operate on. The 2nd is the pattern to match against (a string or regex) and the 3rd is the replacement string.

```
String.replace("I like Elixir!", "like", "love")
#=> "I love Elixir!"
```

Written with the pipe operator, it looks like this:

```
"I like Elixir!" |> String.replace("like", "love")
#=> "I love Elixir!"
```

We remove the first argument from the function on the right side of the pipe as it is put there for us. The first argument's value is on the left of the pipe operator. We still include the 2nd, 3rd, and so on arguments if applicable.

Our current pipeline examples are operating on strings and `String.replace/3` gives us a chance to include a function that takes more than 1 argument. Time for a practice exercise!

## Practice Exercise

Write a module pipeline that starts with this initial string `"   InCREASEd ProdUCtivitY is HEar?  "` and performs the following operations on it.

- Trim off the spaces using `String.trim/1`
- Give the string an initial capital letter using `String.capitalize/1` . This also fixes the mixed case characters.
- Replace the word `"hear"` with `"here"` using `String.replace/3`
- Replace the `"?"` character with `"!"` using `String.replace/3`
- Feel free to sprinkle in some `IO.inspect` calls to peek into the transformation pipeline

**Showing Solution**

### Take a moment to appreciate how this looks. It is visually easy to parse and see the pipeline of transformations being performed.

```
defmodule PipelineTest do

  def run do
    "   InCREASEd ProdUCtivitY is HEar?  "
    |> String.trim()
    |> String.capitalize()
    |> String.replace("hear", "here")
    |> String.replace("?", "!")
  end

end
PipelineTest.run
#=> "Increased productivity is here!"
```

## Custom pipelines

The Pipe Operator does a handy job of pushing the value on the left into the first argument of a function on the right. Pipe Operators can be chained together to create a "pipeline". Pipelines become powerful when ***you*** define the functions used in the pipeline.

## Pipelines with Local Functions

Up to this point, all the functions used in our pipelines are defined by the String module. We can further improve the readability of a pipeline when the functions are local to the module. Either they are imported to the module or declared there.

Let's declare some simple `add/2` and `subtract/2` functions and use them in a pipeline.

```
defmodule PipePlay do

  def perform do
    1
    |> add(3)
    |> add(10)
    |> subtract(5)
    |> add(2)
  end

  def add(value1, value2) do
    value1 + value2
  end

  def subtract(value1, value2) do
    value1 - value2
  end

end

PipePlay.perform()
#=> 11
```

See how well that reads? The pipeline of transformations is clear and intuitive. Don’t disregard the point because our functions are simple in purpose. The same benefits exist when something more meaningful to your project is used.

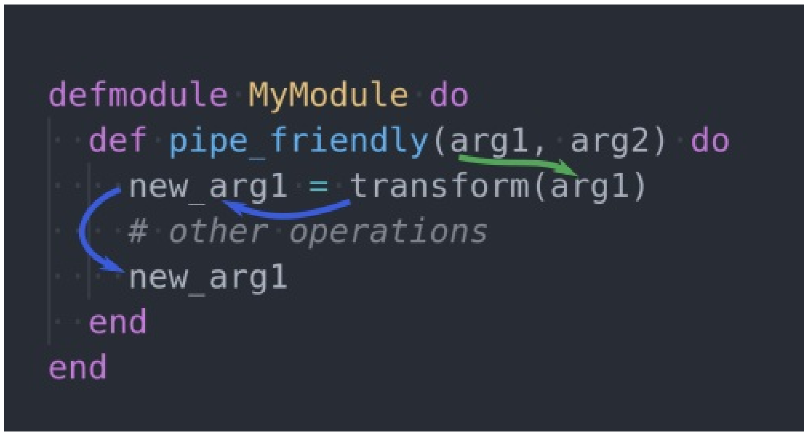
**Pseudo code example:**

```
defmodule MyApp.Ordering do

  # NOTE: pseudo-code only
  def place_order(customer, item) do
    customer
    |> order(item)
    |> allocate_inventory(item)
    |> generate_invoice()
    |> update_billing()
    |> notify_customer()
  end

end
```

Don’t worry yet about *how* to implement something like the above example, we’ll get to that later.

## Pipe-Friendly Functions

In order for a function to be pipe-friendly, it should return the first argument passed in. All the `String` functions we used take a string and return a string as the result. The `Enum.map/2` functions take a list and return a list. Our `add/2` and `subtract/2` functions take a number in the first argument and return a number as the result.



```
defmodule MyModule do
  def pipe_friendly(arg1, arg2) do
    new_arg1 = transform(arg1)
    # other operations
    new_arg1
  end
end
```

Example of a pipe-friendly function design.

You will see this a lot in popular community libraries. When working with web requests in Phoenix or database queries using Ecto, to make something pipe-friendly, give some thought to the arguments passed in and the returned data from a function.

You will also notice functions like `String.length/1` are most certainly ***not*** pipe-friendly. That's okay, the function wouldn't make sense to try and be pipe-friendly. The lesson here is to give a little extra thought when creating your functions and be *deliberate* about what **is** and **is not** designed to be pipe-friendly.

## Recap

Let's briefly review what we covered here:

- The Pipe Operator pushes the data on the left into the first argument of a function on the right.
- A series of pipes are combined to create a pipeline.
- Vertical pipelines are very readable.
- Local functions in pipelines are even more readable.
- Pipe-friendly functions return a transformed version of what they take in.

Creating custom functions that are pipe-friendly allow us to create custom pipelines that are tailored for our application and our problem.

We also covered how `IO.inspect/2` can be a powerful pipe-friendly debugging and insight tool. Make both the **Pipe Operator** and `IO.inspect/2` your new friends!

# Case Statement

The `case` statement is a little powerhouse of pattern matching. It is a common tool and you want to be comfortable with it.

## Simple Yet Powerful

Understanding the basics of the `case` statement is simple. If you are familiar with `case` or `switch` statements in other languages then you already have a good idea of how it works. The `case` statement evaluates an expression and starts matching it against the different conditions starting from the top. It stops at the first match.

The difference with the Elixir `case` compared to other languages is that **pattern matching** support is fully baked in. This is the real power of the `case` statement! It is a compact way to test something against a set of pattern match clauses.

```
case expression do
  pattern ->
    expression_when_matches

  pattern ->
    expression_when_matches
end
```

There are a few nuances we should cover, so let's get to it!

## *Something* Must Match

If *nothing* matches, a type of no-match error is thrown. In this case, a `CaseClauseError` .

```
case 10 do
  4 -> "Four"
  5 -> "Five"
end
#=> ** (CaseClauseError) no case clause matching: 10
#=>     (stdlib) erl_eval.erl:965: :erl_eval.case_clauses/6
```

The equivalent of an `else` or `default` in other languages is a completely open pattern. The underscore character `_` works well for this.

```
case 10 do
  4 -> "Four"
  5 -> "Five"
  _ -> "Something else..."
end
#=> "Something else..."
```

## Support for Guard Clauses

Guard clauses are supported in `case` statement patterns.

```
case {:ok, 86} do
  {:ok, grade} when grade >= 90 ->
    "A"

  {:ok, grade} when grade >= 80 ->
    "B"

  {:ok, grade} when grade >= 70 ->
    "C"

  {:ok, grade} when grade >= 60 ->
    "D"

  _ ->
    "F"
end
```

## Comparison Operators

Guard clauses are great for comparing values in a pattern. If you want to use Comparison Operators like `>` , `>=` , `<` , and `<=` as part of a pattern, then guard clauses are what you want. This is a good time to explain how different *types* are compared to each other when using these operators. You will see how this is important in a moment.

It should be obvious that you can compare values of the same type and get an expected result.

```
1 < 5
#=> true

1 > 5
#=> false
```

What may *not* be obvious is how Comparison Operators work when comparing **different types**. Here is the type comparison order used in Elixir.

```
number < atom < reference < function < port < pid < tuple < map < list < bitstring
```

Exactly how things compare isn't important to remember. The point is that *different types are comparable*. You can read more about the specifics on operator ordering in the documentation on Elixir's operators.

The reason for this is purely a pragmatic one. This means any sorting algorithms you write don't need to worry about how to compare different types. This sorting example shows this in action:

```
Enum.sort([100, :nugget, 2.0, nil, :an_atom, 99, -10, nil, "ABC"])
#=> [-10, 2.0, 99, 100, :an_atom, nil, nil, :nugget, "ABC"]
```

The most relevant point for our guard clause discussion here is how `nil` is handled. Remember, `nil` is an atom! This means...

```
100 < nil
#=> true

nil > 100
#=> true

100 > nil
#=> false

:any_atom > 100
#=> true
```

This means that a pattern match with a guard clause comparing a `nil` to a numeric value might surprise you. Example:

```
case {:ok, nil} do
  {:ok, value} when value > 100 ->
    "Matched > 100"

  _ ->
    "Did not match > 100"
end
#=> "Matched > 100"
```

Because `nil` is an atom, it is sorted higher than a number. So `nil > 100` is `true`.

Pattern matching can help solve this for us. Testing explicitly for the `nil` and handle that.

```
case {:ok, nil} do
  {:ok, nil} ->
    "Was nil"

  {:ok, value} when value > 100 ->
    "Matched > 100"

  _ ->
     "Did not match > 100"
end
#=> "Was nil"
```

My preferred solution is that more functions return `{:ok, result}` and `{:error, reason}` tuples. Then never return a `nil` as a successful result unless that truly is considered a valid and expected response. This pattern of coding helps avoid the problem of an unexpected `nil` comparison at all.

To be clear, this comparison behavior isn't unique to guard clauses. This applies everywhere in Elixir.

## Errors in Guards Prevent Match

An expression that would cause an error in normal code can still be used in a guard clause. The `hd/1` function returns the head of a list. It throws an error when given an argument that isn't a list.

```
hd([1])
#=> 1

hd(1)
#=> ** (ArgumentError) argument error
#=>     :erlang.hd(1)

case 1 do
  value when hd(value) == 1 ->
    "Won't match"

  _value ->
    "Matches here"
end
#=> "Matches here"
```

So you can safely use functions in a guard clause that would otherwise cause errors. If it isn't valid for the data being evaluated, it results in that clause not matching.

## Pattern Matching from the Top

The `case` statement works by matching the data against the patterns starting from the top. It stops on the first pattern match.

```
case 100 do
  value when is_integer(value) ->
    "It's a number!"

  100 ->
    "It's the number 100!"

  _ ->
    "Not sure what that is..."
end
#=> "It's a number!"
```

In the above example, matching against the pattern `100` is a much more specific and exact match, but it stops at the *first* match it makes. So take care to think about the order your patterns are listed! Put the most specific patterns first.

## Match a Bound Variable

The Pin Operator `^` helps to pattern match against an existing bound variable. This can be used in a `case` statement.

```
ten = 10
case 10 do
  4 -> "Four"
  5 -> "Five"
  ^ten -> "Ten"
  _ -> "???"
end
#=> "Ten"
```

Notice how the variable must be bound *prior* to being used in a pattern match? For this reason, the Pin Operator does not work when pattern matching in function declarations. This is an advantage for the `case` statement.

## Assigning a Value From

Variables don't leak. You cannot bind and override a value *inside* a case and have it leak *outside* the `case`. Here's an example of something common in other languages that *does not* work in Elixir the way you might expect.

```
value = 1

case true do
  true ->
    IO.puts "Made it here"
    value = 2

  _ ->
    value
end
#=> warning: variable "value" is unused (if the variable is not meant to be used, prefix it with an under
score)
#=>   iex:6
#=>
#=> Made it here

value
#=> 1
```

In other languages you would expect `value` to have been set to the value `2`. This doesn't happen because inside the scope of the `case`, it is creating a *new* local variable called `value`. It's the new variable that the warning is referring to.

If you want to bind the result of a `case` statement to a variable, do it like this instead.

```
result =
  case true do
    true ->
      "My desired result"

    _ ->
      "Dunno..."
  end

result
#=> "My desired result"
```

## Pipe Friendly

You already know how cool the Pipe Operator is. Did you know that you pipe into a case statement? Because we're using a pipeline, let's try an example using a module. It may look weird at first. Play with it a little.

```
defmodule Testing do

  def classify_text(value) do
    value
    |> String.downcase()
    |> case do
      "hello" <> _rest ->
        :greeting

      _ ->
        :unknown
    end
  end

end

Testing.classify_text("HELLO!")
#=> :greeting

Testing.classify_text("Hello...")
#=> :greeting

Testing.classify_text("I like tacos")
#=> :unknown
```

This comes in handy at the end of a pipeline where you want to convert the result and return it. The alternative is to break it into two statements.

```
defmodule Testing do

  def classify_text(value) do
    to_test = value |> String.downcase()

    case to_test do
      "hello" <> _rest ->
        :greeting

      _ ->
        :unknown
    end
  end

end
```

The variable `to_test` exists only to shuttle the data between the pipeline and the `case`. You have the option to pipeline directly *into* the case as well.

## Practice Exercises

The following practice exercises all use the downloaded project. Make sure you have that available. There is real value in a hands-on experience! It's when you do it yourself that you really learn it and put it together.

I recommend you still check out the solutions after you have the tests passing. It can be helpful to see another way to express the solution.

## Exercise #1 – Guard Clauses

In this exercise use guard clauses with the `case` statement to classify a user struct passed to the function. The job of your function is classify the user as an `:adult` or a `:minor`. If it can't be classified you return an error and the reason (ie. `{:error, reason}`).

To do this, write guard clauses that examine the user's age. If `age >= 18`, return `{:ok, :adult}`. If the `age < 18`, return `{:ok, :minor}`.

Remember the information on Comparison Operators and `nil` ? Pay attention to the tests to see how failure cases are expected to be handled.

NOTE: You could solve this using function clause pattern matching, however for now, focus on solving it with the `case` statement.

```
# the whole describe block
mix test test/case_test.exs:18

# individual tests
mix test test/case_test.exs:20
mix test test/case_test.exs:25
mix test test/case_test.exs:33
mix test test/case_test.exs:35
```

**Showing Solution**

```
  def classify_user(user) do
    case user do
      %User{age: nil} ->
        {:error, "Age is required"}

      %User{age: age} when age >= 18 ->
        {:ok, :adult}

      %User{age: age} when age >= 0 and age < 18 ->
        {:ok, :minor}

      _ ->
        {:error, :invalid}
    end
  end
```

## Exercise #2 – Converting an Error

The downloaded project code contains a text file under `test/support/secret_numbers.txt` . In this exercise, we use the File.read/1 function to open and read the contents of the file.

The “problem” with the File.read/1 function is that when a file does not exist, it returns `{:error, :enoent}` . The value `:enoent` means “No such file or directory” and comes from Posix systems. While it is an *accurate* error, it isn’t what we want to use in our system. Using a `case` statement, open and read a file. When it succeeds, return `{:ok, file_contents}` . When it fails, convert the `:enoent` into a friendly message and return `{:error, "File not found"}` .

The tests for this exercise are:

```
mix test test/case_test.exs:45
mix test test/case_test.exs:50
```

**Showing Solution**

```
  def read_file(filename) do
    case File.read(filename) do
      {:ok, contents} ->
        {:ok, contents}

      {:error, :enoent} ->
        {:error, "File not found"}
    end
  end
```

## Exercise #3 – Converting a DB Result

Ecto is the default library for working with a relational SQL database in Elixir.

For simplicity, the practice project doesn't expect or require a database to be present. The project provides a fake interface that, for our purposes, acts like a database query result.

A common pattern is to run a query that returns the 1 "thing" you want. An example is the need to return a specific Customer or User. The *real* function used is `Ecto.Repo.one/2`. It returns "the thing" or `nil`. In this exercise we're going to get a `User` but we'll use the fake interface `CodeFlow.Fake.Users.one/1`. It receives a `user_id` and returns the `%User{}` with that ID or `nil` if not found.

Create a `find_user` function that receives a `user_id`, uses `CodeFlow.Fake.Users.one/1` in a `case` statement to transform the "user or nil" result into an `{:ok, user}` or `{:error, "User not found"}` result. We want the result as a tuple because it works better for pattern matches that we'll see in future examples.

The tests for this exercise are:

```
mix test test/case_test.exs:55
mix test test/case_test.exs:65
```

**Showing Solution**

Notice that the `:ok` match includes the `%User{}` struct for the pattern match. This is a handy and more explicit way of ensuring we got *exactly* what we expected.

```
  def find_user(user_id) do
    case Users.one(user_id) do
      %User{} = user ->
        {:ok, user}

      nil ->
        {:error, "User not found"}
    end
  end
```

## Recap

The `case` statement helps us do the following things:

- Compare data against a set of patterns.
- Use the full power of pattern matching including guard clauses.
- Convert general function results to something tailored to our application and needs.

A few extra things we learned along the way:

- A `case` must provide a pattern that matches.
- Comparison Operators may treat `nil` differently than you'd expect.
- A pipeline can pipe into a `case`.

# Keyword List

Keyword lists are used *a lot* in Elixir. They are almost treated like a unique data type but they aren't. This is a good time to introduce them because we will be using them in small ways throughout this course.

## A compound type

You already know what a list in Elixir is:

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

You already know what a tuple in Elixir is:

```
{:ok, "The answer is 42!"}

{2020, 1, 1}
```

A Keyword list is the **combination of the two**. It is a list of specially crafted tuples.

The tuple must be a **two element tuple** where the first element is the "key" and it *must be an atom*. The second element is the "value" and it can be anything.

```
# The structure
{:key, value}

# Examples
{:name, "Howard"}
{:level, :debug}
{:user, %User{}}
```

Elixir uses this so frequently, that there is a special syntax used to make it easy and elegant to work with. Check this out, when I write the Keyword list out like described above, Elixir formats the output in a different way.

```
[{:name, "Jane"}, {:age, 36}, {:awesome, true}]
#=> [name: "Jane", age: 36, awesome: true]
```

What just happened there? Try that out in IEx for yourself.

Can we write it the short way too? Yes!

```
[name: "John", age: 29, awesome: false]
#=> [name: "John", age: 29, awesome: false]
```

Just to prove to myself that this other way of writing it is still tuples, I can pull off the head of the list and see it by itself.

```
keywords = [name: "John", age: 29]
[first | _rest] = keywords
first
#=> {:name, "John"}
```

Yes. It really *is* a list of tuples. Cool!

## Working with a Keyword list

A Keyword list is just a list of tuples so normal list processing works. However, because a Keyword list is used more specifically as a key-value data structure, the `Keyword` module provides a number of functions that are helpful for working with a Keyword list.

Here are a couple key functions to know about.

- Keyword.fetch/2 – Return the value for a key, but return it in an `{:ok, value}` tuple.
- Keyword.get/3 – Get the value for a key, provide a default value to return if not found.

## Keyword lists used as options to functions

A very common usage of Keyword lists is to pass options to a function that may change the behavior. This is mostly how we will be using Keyword lists in this course.

Let's see an example in the String module. The function `String.split/3` declaration looks like this:

```
String.split(string, pattern, options \\ [])
```

Let's see how we can use it. First, lets see a basic usage of the function, splitting a string on the colon character.

```
String.split("a:b:c:::f:g", ":")
#=> ["a", "b", "c", "", "", "f", "g"]
```

Notice that we didn't provide any value to `options` argument so the default value of `[]` was used. Now let's use the `:trim` option. It takes a boolean value. Setting it to `true` removes empty strings from the results.

```
String.split("a:b:c:::f:g", ":", [trim: true])
#=> ["a", "b", "c", "f", "g"]
```

Passing a Keyword list as a set of options changed the behavior of the function.

## Shorter syntax when used as option

Another special syntax is supported when our Keyword list is passed as the ***last argument*** to the function. **The square brackets `[]` are optional!** So our `String.split` example can be written like this:

```
String.split("a:b:c:::f:g", ":", trim: true)
#=> ["a", "b", "c", "f", "g"]
```

## Naming convention

When receiving a set of options as a Keyword list in your *own* function, a common naming convention for that argument is to call it `opts` or `options`. When looking at functions in the Elixir standard library, you will frequently see this convention used.

## Default value

In the `String.split/3` example, the default value for options was an empty list. (ex: `options \\ []`). This is important. Your may be tempted to populate the list with the default values you want but this doesn't work as you'd expect. Let's write our own function to see what happens.

```
defmodule Testing do
  def testing_options(opts \\ [trim: false, other_default: 10]) do
    IO.puts(inspect opts)
  end
end

Testing.testing_options()
#=> [trim: false, other_default: 10]
#=> :ok

# I want to override `:trim` to true
Testing.testing_options(trim: true)
#=> [trim: true]
#=> :ok
```

What happened? I wanted to override the `:trim` option to `true`. That desired value *was* set, but it took the *entire* Keyword list as a full replacement and **discarded the other default values I wanted**.

Elixir won't merge the Keyword list we provide with the default one in the function. So what should we do? We can use `Keyword.get/3` inside our function to get the desired value and fallback to the default.

```
defmodule Testing do
  def testing_options(opts \\ []) do
    want_trim = Keyword.get(opts, :trim, false)
    other_default = Keyword.get(opts, :other_default, 10)

    # [...function code...]

  end
end
```

In the above example, I used `Keyword.get/3` to pull out the desired values from the options argument and applied the desired default value if the caller didn't provide an override.

## Practice Exercises

Let's apply what we covered here and give you a chance to create functions that take options as a Keyword list. You learn best by doing so take the time to become more familiar with it.

## Exercise #1 – Rounding precision

Your team has tasked you with writing a function to round numbers to a desired decimal precision. By default, it should round to 4 decimal places. The function will be used in other contexts as well, so it should be configurable through an `opts` argument that takes a Keyword list to specify a different level of precision as appropriate.

The function should take a `:decimals` option. Default it to `4`.

The `Float.round/2` function can perform the rounding work. See the documentation if needed.

The following `mix test` command runs the tests for the describe block without needing to know the line number. Review the tests to see the expected behavior.

```
mix test test/keywords_test.exs --only describe:"rounded/2"
```

**Showing Solution**

### This uses the argument name `opts`. `Keyword.get/3` gets the value and provides the default.

```
  def rounded(value, opts \\ []) do
    decimals = Keyword.get(opts, :decimals, 4)
    Float.round(value, decimals)
  end
```

## Exercise #2 – Refactor `rounded/2` to handle `nil`

After putting your `rounded/2` function into use, the team found another case they would like the function to handle. When the value used for the number of decimals was loaded from the database, they found there are situations when it is `nil`. The following call does not give the desired result!

```
CodeFlow.Keywords.rounded(123.45678901, decimals: nil)
#=> ** (ArgumentError) precision  is out of valid range of 0..15
#=>     (elixir) lib/float.ex:268: Float.round/2
```

A test was added to show the failing code.

```
mix test test/keywords_test.exs --only describe:"refactoring rounded/2"
```

Why is this failing? `Keyword.get/3` gets the value from the Keyword list. If the value is **not in** the list, then it uses the default. But if the value is **explicitly stated** in the list as `nil`, it uses that value. Sometimes that's the behavior you want. If the caller says to use `nil`, then we do.

However, in this case, that is *not* the behavior we want. We want to treat `nil` the same as "not specified". You should note that the `Keyword.get/3` function defaults `nil` as the value returned. So the following two are functionally the same.

```
Keyword.get([], :decimals, nil)
#=> nil
Keyword.get([], :decimals)
#=> nil
```

Your task now is to refactor the solution to handle `nil` the same as not specified. Make sure to check out the elegant solution below!

HINT: Remember that `nil` is "falsey".

**Showing Solution**

The code `nil || 4` returns the value `4` because `nil` is falsey so the logical OR `||` returns the truthy value of `4`. This allows for a nice and elegant solution!

```
  def rounded(value, opts \\ []) do
    decimals = Keyword.get(opts, :decimals) || 4
    Float.round(value, decimals)
  end
```

Make sure the previous tests still pass after you refactored your code for the solution. We don't want our refactor to create a regression!

## Exercise #3 – Compute unit price

Your team has a new mission for you. You are to create a function that can take an `%Item{}` struct which has a `price` and a `quantity` and compute the "unit price". The `quantity` is the number of units in the package. So the "unit price" is the `price / quantity`. Sometimes the function will be used as part of a math operation and it should be returned as a float. Sometimes the function will be used to *display* the unit price and in that case it should display as money. Example: display as `5.10` instead of `5.1`.

The function should take a `:mode` option. When the mode is `:float`, return the value as a float. When the mode is `:money`, convert it to a string representation with a hundreds precision. Default the mode to `:float`.

The Erlang function `float_to_binary/2` can be used for converting to a string. From Elixir you call it like this: `:erlang.float_to_binary(value, options)`. Note it takes a Keyword list in `options` and has an option called `:decimals` that expects an integer.

The Item is defined in the downloaded project in `CodeFlow.Schemas.Item`.

The following `mix test` command runs the tests for the describe block without needing to know the line number. Review the tests to see the expected behavior.

```
mix test test/keywords_test.exs --only describe:"unit_price/2"
```

HINT: Use a `case` statement to handle the different `mode` values.

**Showing Solution**

This uses a `case` statement to handle the different `mode` behaviors. This matches on the `%Item{}` struct only to ensure that we received the expected data type.

```
  def unit_price(%Item{} = item, opts \\ []) do
    raw_price = item.price / item.quantity
    case Keyword.get(opts, :mode, :float) do
      :float ->
        raw_price

      :money ->
        :erlang.float_to_binary(raw_price, decimals: 2)
    end
  end
```

## Revisiting IO.inspect/2

Now that you have spent some time with Keyword lists, let's revisit that great debugging and insight tool IO.inspect/2 . Our original look at it only used the *first* argument but it takes two. The *second* argument is `opts \\ []` . Hey, you know what to expect with that! You can pass a Keyword list with options to change the behavior of the function!

We want to use the `:label` option.

Once you start sprinkling `IO.inspect` calls around in your code, you can't easily tell *which one* you are looking at in the console. The `:label` option will, wait for it, **label** the output for us.

Let's use a previous pipeline exercise to see what it does.

```
defmodule InspectTest do

  def run do
    "   InCREASEd ProdUCtivitY is HEar?  "
    |> IO.inspect(label: "Original")
    |> String.trim()
    |> IO.inspect(label: "Trimmed")
    |> String.capitalize()
    |> IO.inspect(label: "Capitalized")
    |> String.replace("hear", "here")
    |> IO.inspect(label: "Replaced 'hear'")
    |> String.replace("?", "!")
    |> IO.inspect(label: "Replaced '?'")
  end

end

InspectTest.run
#=> Original: "   InCREASEd ProdUCtivitY is HEar?  "
#=> Trimmed: "InCREASEd ProdUCtivitY is HEar?"
#=> Capitalized: "Increased productivity is hear?"
#=> Replaced 'hear': "Increased productivity is here?"
#=> Replaced '?': "Increased productivity is here!"
#=> "Increased productivity is here!"
```

Using the `:label` option takes whatever text you provide as a label, outputs that followed by a `:` , then outputs the inspected data. This is a **very helpful** little tweak that can make a big difference for you.

Note: `IO.inspect/2` supports *many* options. The most commonly used will be `:label` . If you want to dig deeper into what other options are available, see the Inspect.Opts documentation

## Keyword list compared to a Map

A Map is your go-to key-value data structure. After the Map, the next most common one is a Keyword list. Let's consider some of the ways that one is better than the other for different needs.

- Maps are better for *nested* data structures.
- Maps can have *string keys* which works better for receiving user provided values. This avoids the potential Denial of Service attack from a malicious user exhausting your atom table limits. A Keyword list must have atom keys.
- Maps do not stay in the order you declare them. A Keyword list remains in the order you declared it.
- Pattern matching to destructure the data works better on a Map than a Keyword list. See example below:

```
# this works for a map
%{name: user_name} = %{name: "Howard", age: 30, score: 250}
user_name
#=> "Howard"

# this does NOT work for a Keyword list
[name: user_name] = [name: "Howard", age: 30, score: 250]
#=> ** (MatchError) no match of right hand side value: [name: "Howard", age: 30, score: 250]
```

- A Keyword list can contain duplicates and a Map cannot. At first, you might think that's a bad thing. If you imagine using a Keyword list like you would a Map, then yeah, you don't want that. However a Keyword list could be used to express a series of commands. Here's an example:

```
[initial_value: 100, increment_by: 25, decrement_by: 90, increment_by: 10, increment_by: 22]
```

I could write a function that would process the series of commands and a Keyword list works well for that.

## Recap

- A Keyword list is just a list of tuples that conforms to this pattern.

```
{:key, value}
```

- Keyword lists can be written like this:

```
[name: "Tim", points: 30]
```

- Keyword lists create a key-value pair data structure that:
    - is ordered
    - can contain duplicates
- Keyword lists are used for passing a set of options to a function.
- Keyword module has helpful functions for working with a Keyword list.

We also saw how we can pass a Keyword list to `IO.inspect/2` and set a `:label` that really helps identify debug output.

# The Railway Pattern

When talking about the Railway Pattern, the analogy uses train railroad tracks. Until now, we've only looked at straight pipelines where data passes through and is transformed at each step. In essence our train tracks were straight, single line tracks. The Railway Pattern gives us a way to introduce branching logic to a pipeline. It gives us a track switch that can change the flow and cause the train to leave one track and switch to another.

The Railway Pattern introduces branching logic to a pipeline.

## Pipeline is the “Happy Path”

Pipelines read well going from top to bottom. Each step in the pipeline performs some operation. The step’s return value is piped into the next step. Let’s look at a pseudo-code pipeline used to bake a cake.

```
defmodule Life.Cooking do

  def bake_cake(%Person{} = person, %Recipe{} = recipe) do
    %{person: person, recipe: recipe}
    |> gather_ingredients()
    |> prepare_pans()
    |> mix_dry_ingredients()
    |> mix_wet_ingredients()
    |> mix_wet_and_dry()
    |> pour_batter_to_pans()
    |> bake(recipe)
    |> allow_cooling()
    |> frost_cake()
    |> invite_friends()
  end

end
```

I start the pipeline with a map holding the data of a person and the recipe being used. Each step in the pipeline gets me closer to creating my confection and inviting friends to indulge with me.

This pipeline reads as the “Happy Path”. If everything goes well, then I’ll have a delicious cake ready to share with my friends. But what if it doesn’t go well? There are lots of things that *could* go wrong. Here are some possible issues that could occur in the steps.

- I don’t have all the ingredients
- I don’t have a cake pan
- I spill all the wet ingredients while mixing
- My oven is broken
- I burn the cake

If *any* of these steps go really wrong, I **do not** want to invite my friends over to eat a cake that doesn’t exist!

I need a way to run through the steps of the pipeline and handle if things go well or if things go wrong.

The Railway Pattern helps me do this!

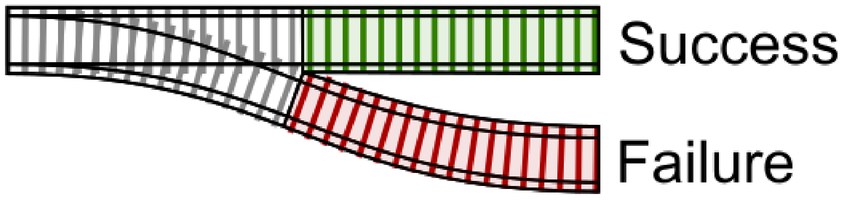
## Forking the Flow

Let’s start with the first step in the pipeline. If I have all the ingredients, I want to proceed with my cake project. The code for the first function might look something like this...

```
defmodule Life.Cooking do

  # ...

  def gather_ingredients(%{recipe: %Recipe{ingredients: ingredients}} = data) do
    case Ingredients.find(ingredients) do
      {:ok, found_ingredients} ->
        # Found all the ingredients needed!
        # Add the found ingredients to the map
        {:ok, Map.put(data, :ingredients, found_ingredients}

      {:error, reason} ->
        # One or more required ingredients are missing
        {:error, "Cake ingredients missing: #{inspect(reason)}"}
    end
  end

end
```

My function has two possible return values. An `{:ok, ...}` tuple with updated data being passed on and an `{:error, ...}` tuple explaining why it failed.

Essentially, what I’ve just done is created a function with one entry point but 2 possible exit paths. I forked the flow that the code can take and created a second “failure” track.



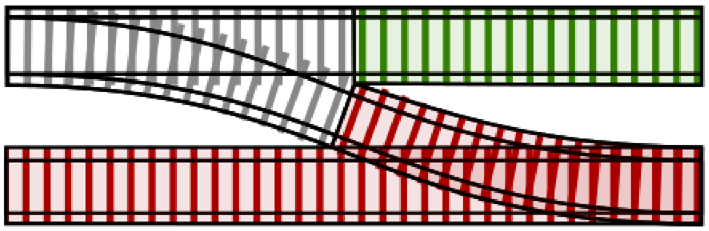
## The Next Segment

The second step in the pipeline is `prepare_pans/1`. This second step, or second segment of track, must deal with there being two possible tracks coming in.

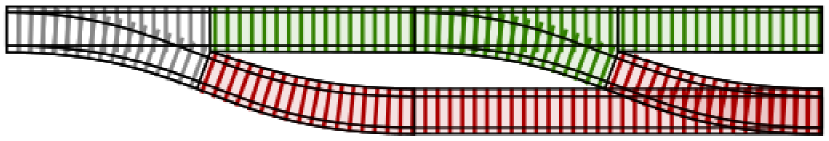
This function might look something like this...

```
defmodule Life.Cooking do

  # ...

  def prepare_pans({:ok, %{recipe: %Recipe{equipment: equipment}} = data}) do
    case Equipment.find(equipment) do
      {:ok, found_pans} ->
        # Found all the right kinds of pans needed!
        # Add the equipment to what we have available
        {:ok, Map.put(data, :equipment, found_equipment}

      {:error, reason} ->
        # Unable to find all the needed equipment for the receipe
        {:error, "Equipment missing: #{inspect(reason)}"}
    end
  end

  def prepare_pans(error), do: error
end
```

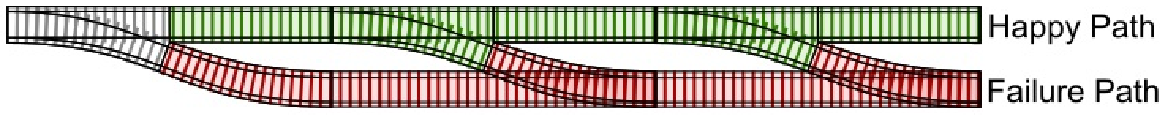
There are 2 function clauses here. The first pattern matches on it being an `{:ok, data}` tuple. The second clause takes whatever else it was (because it wasn't OK with what was expected) and treats that as an `:error` by just passing it along. This "bypass" function is the "failure" track. It does nothing but provide a way through this function when it didn't succeed. Our function looks like this.



Putting our first two functions together, our track looks like this...



If we were to continue building out steps in our pipeline, we would see that each step provides a success track and a failure track. A full string of successes creates our “Happy Path” all the way to the end. However, at any point in the flow, something can go wrong and shunt the flow onto the error path.

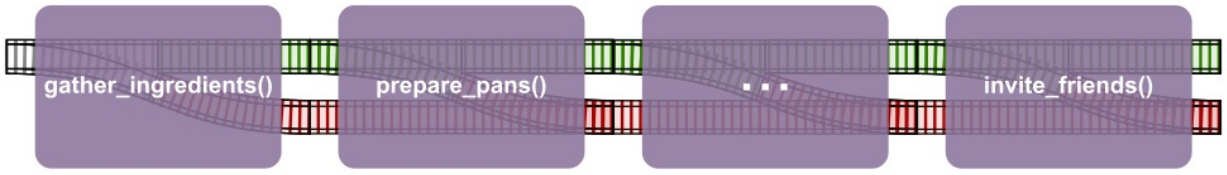


**Thinking Tip: The switches go one way!**

Notice that once on the error path, **there is no way back onto the success path**. The track switches only go the one way. It either *all* goes perfectly or *something* fails and we end up on the failure path and stay there through the end of the pipeline.

## No Embarrassment Guarantee

My initial concern was wanting to avoid the embarrassment of inviting friends for cake if the baking project fails. To guarantee this, my `invite_friends/1` function only needs to send out the invitations when a recognized pattern for success is seen. In this case, when a `{:ok, data}` tuple is received. Any problem encountered along the way directs the Code Flow to the failure path. With an `invite_friends/1` function clause that handles the `{:error, reason}` then I am guaranteed a no embarrassment evening. At least as far as the cake is concerned.



The same can be said for your application. You don’t want to send out an email thanking your customer or user for an action that failed somewhere along the way!

The Railway Pattern gives us an elegant way to express the “Happy Path” that clearly shows the workflow. It adds branching logic to our pipelines to take the flow *off* the Happy Path *when* something goes wrong.

## Strengths and Weaknesses

As with everything, there are strengths and weaknesses. Let’s consider some with the Railway Pattern.

## Strengths

The pipeline literally outlines our “Happy Path”. As developers, we typically code the Happy Path first anyway. As an afterthought we return to think about potential problems. This pattern communicates the “big picture” of what is going on without being muddied with the details of *how* it happens yet. Take another look at the pipeline:

```
    %{person: person, recipe: recipe}
    |> gather_ingredients()
    |> prepare_pans()
    |> mix_dry_ingredients()
    |> mix_wet_ingredients()
    |> mix_wet_and_dry()
    |> pour_batter_to_pans()
    |> bake(recipe)
    |> allow_cooling()
    |> frost_cake()
    |> invite_friends()
```

This code is highly readable. The big picture is laid out and the developer's intent is clear. Additionally, this pattern lends itself well to refactoring and even re-arranging steps if needed.

## Weaknesses

The drawbacks for the Railway Pattern are:

- You *must* create at least 2 functions clauses for each step in the pipeline. One to act as the bypass failure function clause and one or more to handle the success path.
- Doesn't adapt well to functions defined in other modules that weren't created with your pipeline in mind. Therefore all the functions in your pipeline must be created *for the purpose of the pipeline*. The functions can delegate out to other modules and functions, but they need to be created for the pipeline.

## When to use the Railway Pattern?

When the benefits and drawbacks are all considered together, it's fair to ask, "When should I use the Railway Pattern"?

As a starting point and a rule of thumb, it makes sense for a workflow that can all be defined in a single module. An example could be a module like `AccountRegistration`. The entire module is dedicated to a single task. This works well for a workflow with multiple steps but there are really only two possible outcomes. It *all* worked or it didn't.

Each step in the pipeline can be a public function with it's own set of unit tests. The top-level entry point function defines the pipeline in a clear, declarative way. The rest of the module is concerned with *how* to perform the steps.

The Railway Pattern is a coding pattern much like Object Oriented patterns you may already be familiar with. There are many different and appropriate ways to use it. It is a tool that is now available to you. Be willing to try it out in a project and develop a feel for where it works best for you. If it isn't the right fit, you can refactor the code into a different pattern.

## What Should I Pipe?

You don't have to keep piping a map of data through to each function, it can literally be anything from one step to another. **The next function in line just needs to accept the success and failure outputs from the previous function.**

Does it have to be an `:ok` or `:error` tuple? No! However, using an `:ok` and `:error` tuple work well because they make it easy to pattern match and tell the difference between being on the "success" track or the "failure" track. Whatever lets you correctly and reliably tell the difference works.

## Practice Exercise – Award Points

The following exercise uses the downloaded project. Remember you learn best by doing! Use the tests to verify you have a working solution.

Make sure to check out the solutions after you have the tests passing. It can be helpful to see other ways to express the solution.

## Specification

Given a `User` struct and a number of points to award, create a pipeline using the Railway Pattern that does the following things:

1. validates that the user is active
2. validates that the user is at least 16 years old
3. checks that the user's name is not on the blacklist of `["Tom", "Tim", "Tammy"]`
4. increment the user's points by the desired number

***Only*** increment the user's points if *all* the previous steps are valid and pass. If any of the checks fail, return an `{:error, reason}` where reason is the explanation of why it failed.

## Two ways to Test

The tests to focus on are in `test/code_flow/railway_test.exs` . There are two major sections of the test file.

1. A `describe` block that tests the **full pipeline**
2. A separate set of tests for **each step in the pipeline**

The describe block `award_points/2` tests the 4 conditions in the above specification. How you name each function in the pipeline is up to you. The function names are "internal details". However, if you want more guidance or suggestions, the set of tests for each step can help walk you through it in smaller chunks.

**If you don't want the step-by-step tests, then either delete them or comment them out.**

To run all the tests inside the `award_points/2` describe block:

```
mix test test/railway_test.exs --only describe:"award_points/2"
```

To run the tests that cover each specification step, you can run them individually:

```
mix test test/railway_test.exs:30
mix test test/railway_test.exs:35
mix test test/railway_test.exs:40
mix test test/railway_test.exs:45
```

## Tips

Here are a few tips to help you get started:

- One way to start a pipeline using TDD is to start by creating your pipeline steps.
- Tests won't run if the code doesn't compile. Either comment out the steps you aren't ready to work on OR create function stubs so it compiles.
- For checking the name against the blacklist, you can use an `in` guard clause or inside the function you can use `Enum.member?/2` to check against the list.

**Showing Pipeline Solution**
There are a number of valid ways to express this. The names of the functions are up to you as well. This is an example of one possible solution.

```
def award_points(%User{} = user, inc_point_value) do
  user
  |> validate_is_active()
  |> validate_at_least_age(16)
  |> check_name_blacklist()
  |> increment_points(inc_point_value)
end

# Steps implemented using the above names

def validate_is_active(%User{active: true} = user) do
  {:ok, user}
end

def validate_is_active(_user) do
  {:error, "Not an active User"}
end

def validate_at_least_age({:ok, %User{age: age} = user}, cutoff_age) when age >= cutoff_age do
  {:ok, user}
end

def validate_at_least_age({:ok, _user}, _cutoff_age) do
  {:error, "User age is below the cutoff"}
end

def validate_at_least_age(error, _cutoff_age), do: error

def check_name_blacklist({:ok, %User{name: name} = _user})
    when name in ["Tom", "Tim", "Tammy"] do
  {:error, "User #{inspect(name)} is blacklisted"}
end

def check_name_blacklist({:ok, %User{} = user}) do
  {:ok, user}
end

def check_name_blacklist({:error, _reason} = error), do: error

def increment_points({:ok, %User{points: points} = user}, inc_by) do
  {:ok, %User{user | points: points + inc_by}}
end

def increment_points(error, _inc_by), do: error
```

## Recap

The Railway Pattern uses pipelines, pattern matching, and multiple function clauses to create an elegant control flow solution. Here are a few points to keep in mind about the Railway Pattern:

- Since the Railway Pattern needs to pipe data through each step, it works best when there is a structure that can be piped through.
- The top-level pipeline is a clear declaration of what is happening and defines the “happy path”.
- When the flow leaves the “happy path”, it goes onto the “failure path” and flows through the rest of the pipeline on that track.
- It works best when you control the functions that implement each step. This usually means the functions are created for the workflow and live in a module together.

# The “with” Statement

Pattern matching is everywhere in Elixir. This holds true for the `with` statement as well. To understand how it helps, let’s start by looking at a problem it helps fix.

## Solving a Problem

Let’s look at an example of the `case` statement where we want to process some data in multiple steps.

This is a simple example just so we can focus on the *steps* involved and still have something easy to copy and paste into IEx so you can play with it.

We start with a simple map. The following steps happen in order:

1. Fetch the `:width` value from the map
2. Fetch the `:height` value from the map
3. If we got both values, multiply the values and return the result

```
data = %{width: 10, height: 5}

case Map.fetch(data, :width) do
  {:ok, width} ->
    case Map.fetch(data, :height) do
      {:ok, height} ->
        {:ok, width * height}

      error ->
        error
    end

  error ->
    error
end
```

This created an awkward nested `case` statement. It isn’t very readable or pleasant. Seeing the “success path” requires more mental parsing. Nested `case` statements are a code smell. When you see nested `case` statements in your code, you may want to use a `with` instead.

## Introducing `with`

The `with` statement is a very common pattern. The `with` statement documentation says it is:

> Used to combine matching clauses.
>
> https://hexdocs.pm/elixir/master/Kernel.SpecialForms.html?#with/1

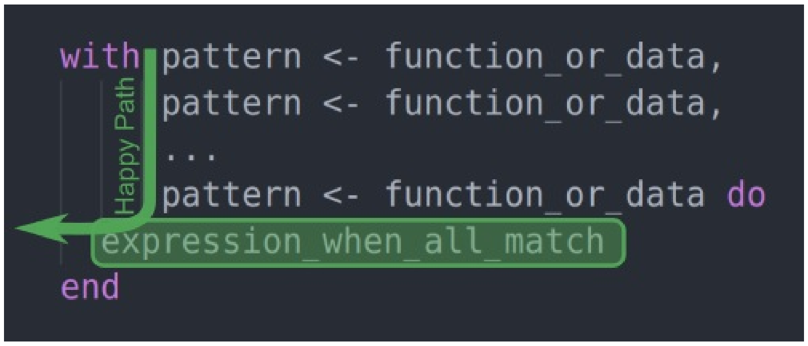
Let’s re-write this using a `with` statement to see how it works.

```
data = %{width: 10, height: 5}

with {:ok, width} <- Map.fetch(data, :width),
     {:ok, height} <- Map.fetch(data, :height) do
  {:ok, width * height}
end
#=> {:ok, 50}
```

This is functionally equivalent to the nested `case` example!

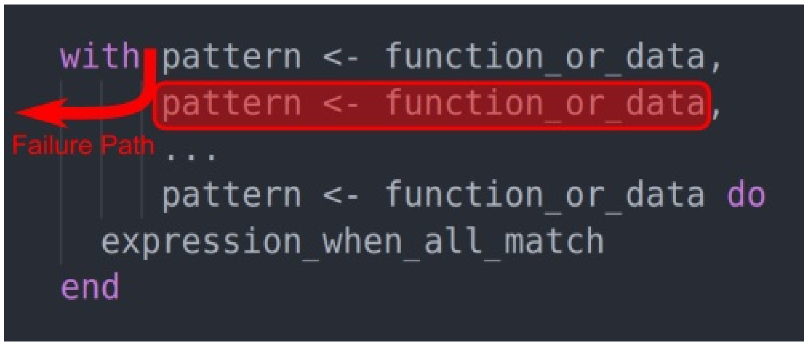
This `with` supports a series of clauses. When *all* clauses match, the `do` block is evaluated and is returned as the result. A `with` clause is made up of a pattern on the left, a left-pointing arrow `<-`, and data on the right.

The series of clauses is evaluated top down with the “success result” returned from the `do` block. This is our “happy path”!



```
with pattern <- function_or_data,
     pattern <- function_or_data,
     ...
     pattern <- function_or_data do
  expression_when_all_match
end
```

What happens when one of the clauses fails?



```
with pattern <- function_or_data,
     pattern <- function_or_data,
     ...
     pattern <- function_or_data do
  expression_when_all_match
end
```

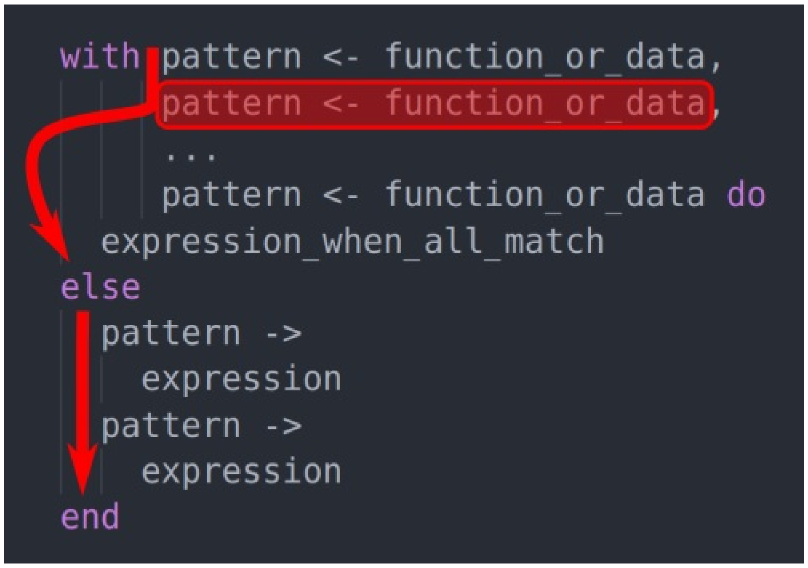
The first clause to not match stops the flow and the non-matching data is returned as the result.

> When you see nested `case` statements in your code, you may want to use a `with` instead.

## Optional `else`

What if a step fails and we want to perform some special handling? To address this, the `with` statement supports an optional `else` clause. We only include it when we want to respond to an error. We may log the failure, transform the result or take some other action.

When we provide an `else` clause, instead of a match failure being returned right way, it skips any remaining steps and begins pattern matching the else clauses starting from the top. It stops at the first match and the expression is used as the return value.



```
with pattern <- function_or_data,
     pattern <- function_or_data,
     ...
     pattern <- function_or_data do
  expression_when_all_match
else
  pattern ->
    expression
  pattern ->
    expression
end
```

The else clause pattern matches look more like a case . The arrows go to the right like a case statement. It behaves like a case as well so it should feel familiar.

Let’s re-visit the previous example. If a Map.fetch/2 call fails, it returns the atom :error .

```
data = %{width: 10}

with {:ok, width} <- Map.fetch(data, :width),
     {:ok, height} <- Map.fetch(data, :height) do
  {:ok, width * height}
end
#=> :error
```

## Practice Exercise #1

Using your understanding of the else clause, match on the :error atom and instead return the error tuple and message {:error, "Invalid data"} .

**Showing Solution**

```
data = %{width: 10}

with {:ok, width} <- Map.fetch(data, :width),
     {:ok, height} <- Map.fetch(data, :height) do
  {:ok, width * height}
else
  :error ->
    {:error, "Invalid data"}
end
#=> {:error, "Invalid data"}
```

## Telling the Difference

This demonstrates a potential issue. How can you tell the difference between a missing :width or :height ? In both cases it goes into the else as :error . Let’s consider a pseudo code example to illustrate the point. Assume we have two functions:

- Users.get/1 – returns a %User{} struct or nil if not found
- Customers.get/1 – returns a %Customer{} struct or nil if not found

If I have a with statement like the one below, where I only consider it a match when I get the struct and not a nil result. Then the question is, if I end up in the else clause with a nil , how can tell if we didn’t find the user or if it was the customer that failed?

```
with %User{} = user <- Users.one(1),
     %Customer{} = customer <- Customers.one(100) do
  # return success result
else
  nil ->
    "???"
end
```

I can’t tell which failed! Patterns that **did** match and **were** bound are **not available** in the `else` clause to try and test for.

A common way to solve this is to not call functions that return a `nil`. Instead, call functions that return a result like `{:ok, user}` or `{:error, "User not found"}`. Then the cause of failure is clear. The more you adopt and use pattern matching in your applications, the more it impacts the design of your APIs. You want easier ways of matching and knowing if a call succeeds or not. Returning tuple results is a common way to do that.

But what if there are existing functions I need to call and they return something where I can’t tell the reason for a failure? There are a couple ways to deal with this issue.

1. Create a local private function that calls the function you need and converts a `nil` result into an error with a message.
2. Wrap the function in a tuple in the `with` statement to identify where they come from.

## Local Private Functions

Let’s re-visit the width and height example. Create local private functions to solve this issue:

```
defmodule Playing do

  def compute_area(data) do
    with {:ok, width} <- get_width(data),
        {:ok, height} <- get_height(data) do
      {:ok, width * height}
    end
  end

  defp get_width(data) do
    case Map.fetch(data, :width) do
      {:ok, width} ->
        {:ok, width}

      :error ->
        {:error, "Width not found"}
    end
  end

  defp get_height(data) do
    case Map.fetch(data, :height) do
      {:ok, height} ->
        {:ok, height}

      :error ->
        {:error, "Height not found"}
    end
  end
end

Playing.compute_area(%{width: 10})
#=> {:error, "Height not found"}

Playing.compute_area(%{height: 5})
#=> {:error, "Width not found"}
```

## Wrapping the Result

Another way to solve this is to wrap the function call in a tuple used to identify the result. The pattern match on the left side must unwrap it when successful. However, it is most helpful in the `else` when identifying why it failed.

```
defmodule Playing do

  def compute_area(data) do
    with {:width, {:ok, width}} <- {:width, Map.fetch(data, :width)},
         {:height, {:ok, height}} <- {:height, Map.fetch(data, :height)} do
      {:ok, width * height}
    else
      {:width, :error} ->
        {:error, "Width not found"}

      {:height, :error} ->
        {:error, "Height not found"}
    end
  end
end

Playing.compute_area(%{width: 10})
#=> {:error, "Height not found"}

Playing.compute_area(%{height: 5})
#=> {:error, "Width not found"}
```

## Guard Clauses

Guard clauses are supported in both the patterns in the `with` clauses and the `else` clauses. Just be aware that this level of pattern matching is available to you.

```
data = %{width: 10, height: 5}

with {:ok, width} when is_number(width) <- Map.fetch(data, :width),
     {:ok, height} when is_number(height) <- Map.fetch(data, :height) do
  {:ok, width * height}
end
```

## Excessive Use

It is possible to over apply the `with` clause. I see people write a 1-line `with` clause and it feels excessive. Let’s see an example of two ways to write the same basic thing. One way uses `with` and the other uses `case` .

```
defmodule Example do
  def excessive_with(data) do
    with {:ok, width} <- Map.fetch(data, :width),
         do: {:ok, width}
  end

  def using_case(data) do
    case Map.fetch(data, :width) do
      {:ok, width} ->
        {:ok, width}

      :error ->
        {:error, "Width not found"}
    end
  end
end
```

When learning a new language, it’s understandable that we “latch on” to a new idea that works. When you realize that a `with` can be used anywhere in place of a two pattern `case` statement, then it may be tempting to use the tool *everywhere* that it *can* work.

Personally, I value code with this mindset:

- Clarity > Brevity
- Explicit > Implicit

A one-line `with` statement is more brief and compact than the `case` version. However, the error return type doesn’t appear to have been given any thought. Exactly what it returns when it errors is ***unclear***. In this case, `Map.fetch/2` will return an `:error` atom when it fails. This now requires the *caller* of the `excessive_with/1` function to know that they might get an `:error` back and have to deal with that. However, just a quick look into the the `excessive_with/1` function won’t tell you what it returns on a failure. The reader has to read the code and know what `Map.fetch/2` returns on a failure. The return value is ***implicit***.

The `case` version is longer. It ***explicitly*** deals with the error response. It is more ***clear*** when glancing at the function what 2 return types it will have.

The `with` statement is powerful. It’s greatest power comes when *chaining multiple patterns together*. Avoid using a single-line `with` when a `case` works and can be clearer.

## Strengths and Weaknesses

The `with` statement has it’s own set of strengths and weaknesses. Let’s consider what they are.

## Strengths

- The `with` statement works well for pulling together different functions to create a Code Flow. The functions do not have to be created for the purpose.
- Each step lets you pattern match and gather the values you need from one step to use in the next steps.
- Solves the nested `case` statement code-smell by making the “Happy Path” easier to see and reason about.
- It is a light-weight pattern and can be used quickly to pull different operations together.

## Weaknesses

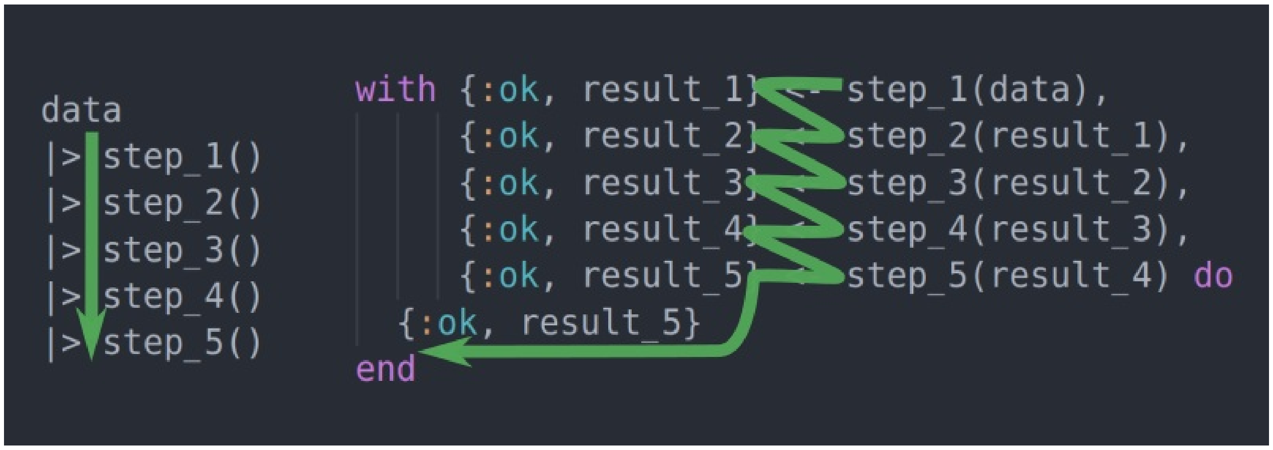
- Any non-matching step in the sequence is returned immediately from a `with` statement. You need to think about the return types of all the functions you call.
- If you provide an `else` clause, any match failure drops into the `else` body. Depending on the functions called in the steps, it may be difficult to identify *which* match failed.
- When a match fails, you are left dealing with the non-match in a fairly disconnected way. In the Railway Pattern, you deal with the non-match in a purpose-built function.

## When to use the With Statement?

Honestly, when dealing with a series of steps in your application, the `with` statement is likely your go-to solution. Most business logic is made of using lower level accessing functions to perform meaningful and deliberate operations.

## Compared to Railway

The `with` statement isn’t quite as clear and clean as the Railway Pattern’s top level “Happy Path” pipe. The “Happy Path” in the `with` clause is a little jagged by comparison. It doesn’t read quite as clearly.



```
data
|> step_1()
|> step_2()
|> step_3()
|> step_4()
|> step_5()
```

```
with {:ok, result_1} <- step_1(data),
     {:ok, result_2} <- step_2(result_1),
     {:ok, result_3} <- step_3(result_2),
     {:ok, result_4} <- step_4(result_3),
     {:ok, result_5} <- step_5(result_4) do
  {:ok, result_5}
end
```

Railway Pattern vs With

However, the `with` statement does a better job of pulling together different functions in your project that weren’t designed to be part of the Railway Pattern. For this reason, and the brevity with which you can write `with` statements, you will likely find many applications and uses for this in your projects.

Handling errors is more explicit in the Railway Pattern because you already have functions dedicated to the step. If you have the need to deal with a step failure in more than a trivial way, then the Railway Pattern can work well. In the `with` statement, an error at *any* step is either returned as-is or drops into the `else` body where all other failures go as well. Depending on the error, it may not be easy to identify what went wrong in an `else` clause. For this reason, the `with` clause works best when pulling together functions that provide a good pattern matching result value.

## Quick Note on Aliases

In Elixir, an alias can create a shortcut to a module in your code. Since the `with` statement makes it easier to pull together different parts of our application, this is a good time to start using aliases.

Let’s say the module we want to use is `CodeFlow.Schemas.Order`. Every time we call a function or reference the struct we’d be writing something like `%CodeFlow.Schemas.Order{}` which can be long, tedious, and error prone. Adding an `alias` in the file creates a shortcut to that module by just writing `Order`.

Be default, the shortcut name is the last segment of the namespace. The `alias` command also allows you to *change* the name. This is helpful if you encounter names that would otherwise collide.

```
# an alias can't be use on both as they'd both try and locally use `Order`
alias MyApp.Web.Controllers.Order
alias MyApp.Schemas.Order

# using `:as`, we can specify the name to use
alias MyApp.Web.Controllers.Order, as: OrderController
alias MyApp.Schemas.Order
```

Refer to the official documentation for more information. Note that we use a Keyword list to specify the `:as` option to `alias`.

### Thinking Tip: Aliases are Optional

You do not need to alias a module in order to use it. You can always use the fully qualified module name from anywhere in your application. An `alias` is most commonly used to give you a shortcut to the module so you can locally reference it in a more direct way.

## Practice Exercise #2

In this exercise, you will complete the `CodeFlow.With.place_new_order/3` function located in the download project in file `lib/with.ex`.

Write the `with` statement that performs the task of placing a new order for a customer. All of the supporting code is already prepared in the project. There are fake business layer contexts for working with Customers, Orders, and Items. There are structs that represent database entries as well.

Aliases are already created for you but commented out as they aren’t used yet.

## Specification

1. Find the customer – `Customers.find/1`
2. Find the item being ordered – `Items.find/1`
3. Start a new order – `Orders.new/1`
4. Add the found item to the order – `Orders.add_item/3`
5. Notify the customer of the new order – `Customers.notify/2`
6. Return `{:ok, order}` when it succeeds

When notifying the Customer, you specify the “event” using a tuple like `{:order_placed, order}`.

The `Customer.notify/2` function may fail with an `{:error, :timeout}`. There is a test case showing we want this translated into a friendlier error message.

Any other failures messages should be returned as an `{:error, text_reason}` tuple.

The tests to run are in `test/with_test.exs`. You choose the order you want to run them in.

```
mix test test/with_test.exs
```

Make sure to checkout the solution once you are done! There are some more thoughts and points to consider.

**Showing Solution**

Do you see the benefit of functions returning tuples? It is easier for pattern matching. Also notice that if `Customers.notify/2` handled the error differently and returned a string, then the `else` wouldn’t be needed at all.

What is the “right” thing to return depends on the project. If you expect to handle and *deal* with the error, then working with an atom is much better than getting meaning from human friendly text. However, if the error is considered “unrecoverable”, then a text message is fine.

```
defmodule CodeFlow.With do
  @moduledoc """
  Defining a workflow or "Code Flow" using a `with` statement.
  """
  alias CodeFlow.Schemas.Order
  alias CodeFlow.Fake.Customers
  alias CodeFlow.Fake.Orders
  alias CodeFlow.Fake.Items

  @spec place_new_order(customer_id :: integer, item_id :: integer, quantity :: integer) ::
          {:ok, Order.t()} | {:error, String.t()}
  def place_new_order(customer_id, item_id, quantity) do
    with {:ok, customer} <- Customers.find(customer_id),
         {:ok, item} <- Items.find(item_id),
         {:ok, order} <- Orders.new(customer),
         {:ok, order} <- Orders.add_item(order, item, quantity),
         :ok <- Customers.notify(customer, {:order_placed, order}) do
      {:ok, order}
    else
      {:error, :timeout} ->
        {:error, "Timed out attempting to notify customer"}

      error ->
        error
    end
  end
end
```

## Recap

Make sure you are comfortable using the `with` statement. It is a handy language feature and you are likely to encounter it frequently in Elixir projects.

- The `with` statement can be a good solution to a nested `case` code-smell.
- Any match failure in a `with` clause is either returned immediately or drops into the optional `else` clause.
- Pattern matching, including guard clauses, are used in the `with` statement and in the `case`-like `else` clause as well.
- The `with` statement complements the Railway Pattern. The `with` provides a brief pipeline-like ability you can use anywhere.
- Pulls disparate re-usable business logic functions together into an ad hoc flow.

Refer to the official docs on the `with` statement which provide additional examples and explanation.

# Looping with Recursion

When people newly come to Elixir, it is common to struggle with doing “basic” things like looping. “Looping” is when you want to perform an action a number of times or you want to perform an action on each item in a list. This really is a fundamental need in programming!

The problem people encounter is that the familiar tools and patterns they are comfortable with aren’t available in Elixir. Of course this doesn’t mean you can’t accomplish the your goal, it just means you need to learn some new ways to do it. You need a new way to think about it.

## Recursive Functions

Let’s briefly define what a Recursive Function is:

> A recursive function is a function that calls itself during its execution. This enables the function to repeat itself several times, outputting the result at the end of each iteration.
>
> https://techterms.com/definition/recursivefunction

A recursive function is one that calls itself. Recursion, combined with pattern matching and other features of the BEAM enable for some powerful abilities and looping is one of them.

## Revisiting the `for` loop

Functional Programming languages don’t have the common `for` loop you are likely familiar with. Here’s an example of a Javascript `for` loop.

```
var data = [1, 2, 3, 4, 5];

for (var i = 0; i < data.length; i++) {
  console.log(`Value of data[${i}] =`, data[i]);
}
```

After running this in a browser, the console shows this output:

```
Value of data[0] = 1
Value of data[1] = 2
Value of data[2] = 3
Value of data[3] = 4
Value of data[4] = 5
```

Writing a `for` loop like this isn’t allowed in functional languages because each iteration is *mutating* the variable `i`. Functional languages have *immutable* data so right away this construct isn’t allowed.

How do we then solve problems in Elixir where we need to “loop”? Elixir has some great convenience functions available to help us solve “looping” problems and we *will* cover them. Before we talk about the shortcuts available, let’s talk about how it fundamentally works. Understanding the fundamentals is important because it gives you confidence in the system and the principles are central to how more advanced features and behaviors work.

## Recursively processing a list

Let’s convert that Javascript example into an Elixir one and see how it works. We want to write a function that can process a list of numbers and print to the console the value of the number. Using pattern matching, I can pull off the first number of the list `num` and get the rest of the list in `rest`. Ex: `[num | rest]`. This works! Almost…

```
defmodule Playing do
  def process([num | rest]) do
    IO.puts "Value of num = #{num}"
    process(rest)
  end
end

data = [1, 2, 3, 4, 5]
Playing.process(data)
#=> Value of num = 1
#=> Value of num = 2
#=> Value of num = 3
#=> Value of num = 4
#=> Value of num = 5
#=> ** (FunctionClauseError) no function clause matching in Playing.process/1
#=>
#=>     The following arguments were given to Playing.process/1:
#=>
#=>         # 1
#=>         []
```

It correctly prints out all the numbers but then fails with a function clause matching error when it gets to an empty list. Because `[]` does not match the pattern `[num | rest]`. The fix is to add the second clause that matches an empty list.

```
defmodule Playing do
  def process([num | rest]) do
    IO.puts "Value of num = #{num}"
    process(rest)
  end

  def process([]), do: "Done!"
end

data = [1, 2, 3, 4, 5]
Playing.process(data)
#=> Value of num = 1
#=> Value of num = 2
#=> Value of num = 3
#=> Value of num = 4
#=> Value of num = 5
#=> "Done!"
```

Now it works correctly! It processes the first item in a list and passes on the remaining items to be processed the same way. It matches when there are no items left to process and stops.

Right now, the final return is the string `"Done!"`. That `"Done!"` clause can be helpful when we want to return a computed result. Let’s change the function to instead **sum** all the numbers in the list and see how that works.

## Summing a list

To get a more complete view of how this works, let’s look at how we could sum a list of numbers in Javascript first.

```
function sum(list) {
  var acc = 0;
  for (var i = 0; i < data.length; i++) {
    acc = acc + data[i];
  }
  return acc;
}

var data = [1, 2, 3, 4, 5];
sum(data);
//=> 15
```

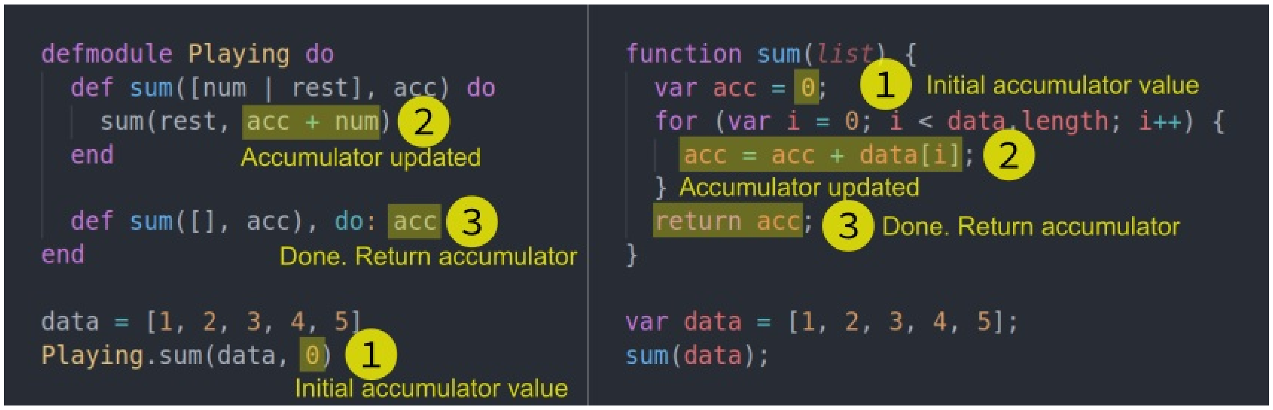
In Javascript we declare a variable to *accumulate* the sum of values as we go. We will call this accumulator `acc` and it starts with a value of `0`. The `for` loop mutates both the `i` counter and the `acc` accumulator as it runs. After the loop completes, we have our total stored in `acc` and finally return `acc` as the answer.

How can we do that in Elixir without mutating variables?

```
defmodule Playing do
  def sum([num | rest], acc) do
    sum(rest, acc + num)
  end

  def sum([], acc), do: acc
end

data = [1, 2, 3, 4, 5]
Playing.sum(data, 0)
#=> 15
```

The Elixir version has all the same parts as the Javascript version, just arranged differently. The most important change is that variables are *never mutated*!

A side-by-side comparison will help solidify this point.



```
defmodule Playing do
  def sum([num | rest], acc) do
    sum(rest, acc + num)
  end

  def sum([], acc), do: acc
end

data = [1, 2, 3, 4, 5]
Playing.sum(data, 0)
```

```
function sum(list) {
  var acc = 0;
  for (var i = 0; i < data.length; i++) {
    acc = acc + data[i];
  }
  return acc;
}

var data = [1, 2, 3, 4, 5];
sum(data);
```

2 Accumulator updated
3 Done. Return accumulator
1 Initial accumulator value
1 Initial accumulator value
2 Accumulator updated
3 Done. Return accumulator

Pattern matching tells us when we should stop recursively calling `sum` and return the accumulator.

**Thinking Tip**

To recursively loop in Elixir we need **2 function clauses**. One that does the iteration work and another to tell us when we are done. Pattern matching works perfectly for this. Also, if you want to accumulate a value in the process, you add arguments to pass it along.

Example of the 2 function clauses using pattern matching needed for looping:

```
def process([item | rest]) do
  # do work
  process(rest)
end

def process([]) do
  # we are done
end
```

## A `while` Equivalent

A `while` loop is also a common feature in imperative languages. Like the `for` loop, this isn't allowed in functional languages, and similarly it isn't needed. Let's look at what makes a `while` loop.

```
while (condition) {
  // code block to be executed
}
```

While some condition remains true, execute the code inside the while block.

Let’s create a Javascript example and convert it to Elixir to see how it compares. This defines a `build_text` function. You pass in the number of lines to generate and it returns a string with that number of lines added.

Like the `for` loop, it mutates a looping counter `num` and builds up a `text` variable to be returned as the result.

```
function build_text(number) {
    var num = 0;
    var text = "";

    while (num < number) {
        text += `Processed number ${num}\n`;
        num++;
    }
    return text;
}

build_text(5)
#=> "Processed number 0
#=> Processed number 1
#=> Processed number 2
#=> Processed number 3
#=> Processed number 4
#=> "
```

Let’s see how this can be done in Elixir.

```
defmodule Playing do
  def build_text(number) do
    do_build_text(number, 0, "")
  end

  defp do_build_text(total, num, text) when num < total do
    do_build_text(total, num + 1, text <> "Processed number #{num}\n")
  end

  defp do_build_text(_total, _num, text), do: text
end

text = Playing.build_text(5)
#=> "Processed number 0\nProcessed number 1\nProcessed number 2\nProcessed number 3\nProcessed number 4\n"

IO.puts text
#=> Processed number 0
#=> Processed number 1
#=> Processed number 2
#=> Processed number 3
#=> Processed number 4
```

Wow! This one looks a little different! We have some fun things to talk about here.

- The first is that our `build_text/1` function declaration looks just like the Javascript one, meaning that the API for the function is the same.
- Remember that to do recursive looping functions in Elixir we need *at least 2 function clauses*? These are the `do_build_text/3` function clauses.
- The `do_` prefix is an **Elixir naming convention** for this pattern. Specifically, the `do_` prefix naming convention is used when you want **private functions** to do the work and the **pattern matching clauses** don’t match the **API you want for the public interface.**
- The `do_` functions are declared using `defp` making them private to the module.

- The public `build_text/1` function handles starting the process with all the initial values. It starts the counter at `0` and the initial `text` with `""`. This provides the *same behavior* as the `while` version. It sets up the state for the loop then calls the private function to do the looping.
- The guard clause `when num < total` behaves like the condition in the `while` loop! That guard clause tells us how long we should stay in the loop.
- The final `do_` clause matches when we have looped the desired amount and it returns the accumulated `text`.

## Practice Exercises

Now it's time to put theory into practice! You learn best by doing. The following exercises give you an opportunity to experiment and apply what we are learning here. Take the time to play with it!

## Exercise #1 – Compute Order Total

Using recursion, process a list of `OrderItem` structs to compute an order total. Refer to the following structs for details as needed:

- lib/schemas/order_item.ex
- lib/schemas/item.ex

The order total is computed as an Item's price * the quantity being ordered. Summing all of those together for the total order price.

The following `mix test` command will run the tests for the describe block without needing to know the line number. Review the tests to see the data samples being tested.

```
mix test test/recursion_test.exs --only describe:"order_total/1"
```

HINT #1: Define `defp` functions with a `do_` prefix to handle the iterating.

HINT #2: This exercise is more like the `for` loop examples above.

**Showing Solution**

This solution *could* pattern match on all the variables being used, but it isn't necessary. Pattern matches should focus on what makes a data situation unique and what is necessary to be guaranteed. In this case, I do want to guarantee that I have an `OrderItem` struct.

```
  def order_total(order_items) do
    do_order_total(order_items, 0)
  end

  defp do_order_total([%OrderItem{} = order_item | rest], total) do
    do_order_total(rest, (order_item.quantity * order_item.item.price) + total)
  end

  defp do_order_total([], total) do
    total
  end
```

## Exercise #2 – Counting Active

Using recursion, process a list of Customer structs to count the number of active customers. A Customer has an `active` boolean flag. Conditionally increment the counter only when a Customer is active. Refer to the following as needed:

- lib/schemas/customer.ex – Defines the customer struct
- test/recursion_test.exs – Shows data examples your code needs to process

The following `mix test` command will run the tests for the describe block without needing to know the line number.

```
mix test test/recursion_test.exs --only describe:"count_active/1"
```

HINT #1: Define `defp` functions with a `do_` prefix to handle the iterating.

HINT #2: This exercise is more like the `for` loop examples above.

**Showing Solution**

This solution uses additional pattern matching clauses to detect when the counter should increase. Comments in the code describe what each clause is doing.

```
alias CodeFlow.Schemas.Customer

def count_active(customers) do
  do_count_active(customers, 0)
end

# Active customer, increment counter and recurse
defp do_count_active([%Customer{active: true} | rest], acc) do
  do_count_active(rest, acc + 1)
end

# Not active, don't increment counter, continue recursing
defp do_count_active([_customer | rest], acc) do
  do_count_active(rest, acc)
end

# Reached end of list, return total count
defp do_count_active([], acc) do
  acc
end
```

## Exercise #3 – Creating Customers

The goal is to create a desired number of new customer entries. Use the function `CodeFlow.Fake.Customers.create/1` to perform the customer create operation. In order to create a valid customer, you must at least provide a name. Something like this could easily exist in a testing framework setup.

Don’t get too hung up on checking that the customer was correctly created, that’s not the focus here. The focus is on loop control.

```
mix test test/recursion_test.exs --only describe:"create_customers/1"
```

HINT: This exercise is focused more on re-implementing the `while` loop we looked at.

**Showing Solution**

```
def create_customers(number) do
  do_create_customers(number, 0)
end

defp do_create_customers(total, num) when num < total do
  # for simplicity, not handling a failed create
  {:ok, _customer} = Customers.create(%{name: "Customer #{num}"})
  do_create_customers(total, num + 1)
end

defp do_create_customers(total, _num) do
  "Created #{total} customers!"
end
```

## Extra Credit Exercise – Fibonacci Sequence

This one is just for fun! This is only about writing a recursive function. In Computer Science and even job interviews, people want to see that you know how to use recursion. If you aren’t familiar with a Fibonacci Sequence, it is a number sequence where a number is computed by adding the two previous numbers in the sequence together. The only initial given numbers are the first two numbers of the sequence which are `0` and `1`.

Confusing? Let's see an example.

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

The 6th index in the sequence is `8` (using a zero based index). It is computed by adding the previous two numbers `3` and `5` together. Executing your implemented function `Recursion.fibonacci(6)` should return the value `8`.

The test checks that your solution correctly solves for a number of different indexes. Remember: pattern matching is your friend!

```
mix test test/recursion_test.exs:80
```

**Showing a Hint**

Index `0` returns `0` and index `1` returns `1`.

Pattern matching with function clauses… hmm.

**Showing Solution**

This solution is a brute force version. It's typically what people want to see as a demonstration of a working solution. If you run this with an index like `43`, it will be *very* slow. It gets exponentially slower with higher indexes. There are many fun ways to play with creating optimized solutions. Feel free to play with it if you like.

```
  def fibonacci(0), do: 0
  def fibonacci(1), do: 1

  def fibonacci(index) do
    fibonacci(index - 2) + fibonacci(index - 1)
  end
```

To play with some values in IEx you can do this:

```
$ iex -S mix

CodeFlow.Recursion.fibonacci(43)
#=> 433494437
```

## Recap

Do you see the pattern here? The recursive versions have all the same parts as the imperative Javascript versions, just organized differently. The new organization does the following for us:

- removes all variable mutations
- uses recursion for looping
- pattern matching 2 or more function clauses controls the loop
- accumulators are passed through the functions

Additionally, we covered the `defp do_perform_some_work` naming convention. This naming convention puts a `do_` prefix on private functions that do the looping work you need while keeping your module's public API clean and consistent.

# Tail Recursion

In many languages like Javascript, if a function recursively calls itself, after a certain number of recursive calls, it fails with a "stack overflow" error. This is when the call stack depth exceeds some limit.

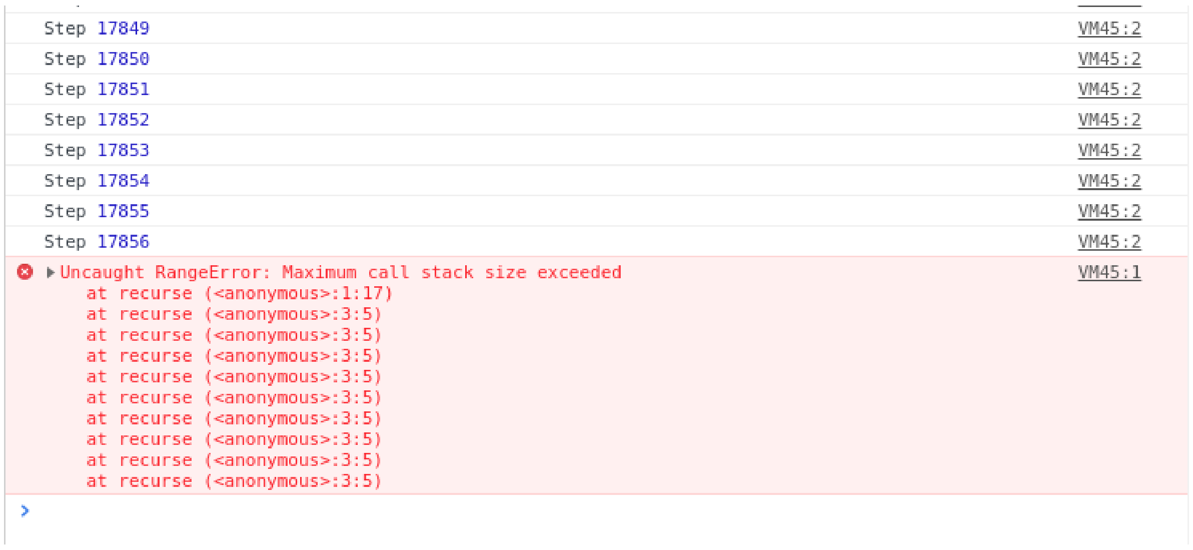
This doesn’t happen with Elixir and other Functional Programming languages.

## Watch a Stack Blow

As an example, let’s run this in a Javascript console. You can paste this into a browser’s developer console to try it out.

```
function recurse(num) {
    console.log("Step", num);
    recurse(num + 1);
}

recurse(0);
```

After letting it run for a bit, the console struggles then throws the error.



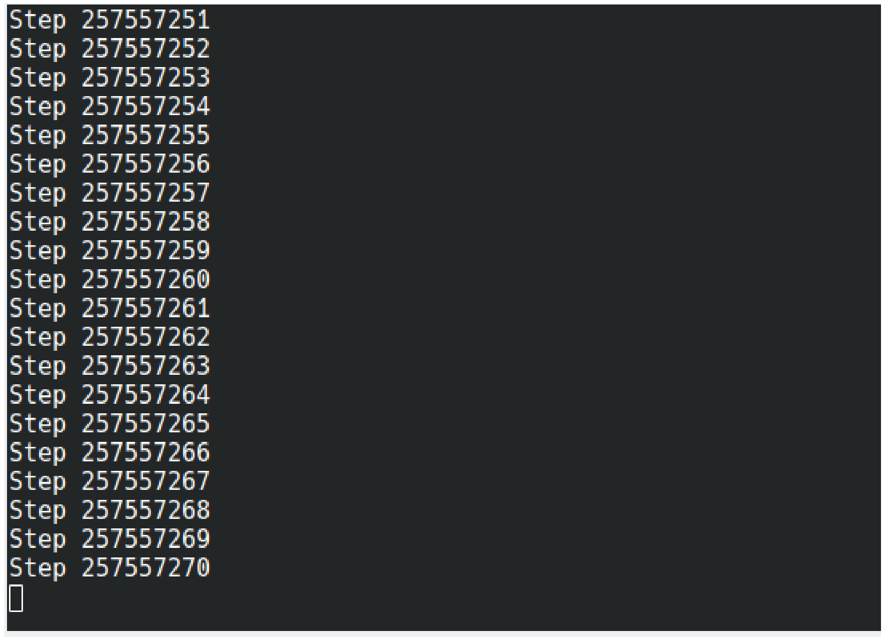
```
Step 17849                                                     VM45:2
Step 17850                                                     VM45:2
Step 17851                                                     VM45:2
Step 17852                                                     VM45:2
Step 17853                                                     VM45:2
Step 17854                                                     VM45:2
Step 17855                                                     VM45:2
Step 17856                                                     VM45:2
▶Uncaught RangeError: Maximum call stack size exceeded         VM45:1
    at recurse (<anonymous>:1:17)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
    at recurse (<anonymous>:3:5)
>
```

## Using Tail Recursion

Let’s try the same thing in Elixir…

```
defmodule StackTest do

  def recurse(num) do
    IO.puts "Step #{num}"
    recurse(num + 1)
  end

end

StackTest.recurse(0)
```

Run this in an IEx session and see it continually churn out numbers.



```
Step 257557251
Step 257557252
Step 257557253
Step 257557254
Step 257557255
Step 257557256
Step 257557257
Step 257557258
Step 257557259
Step 257557260
Step 257557261
Step 257557262
Step 257557263
Step 257557264
Step 257557265
Step 257557266
Step 257557267
Step 257557268
Step 257557269
Step 257557270
```

No slowing down, no stopping. This will literally run forever. To stop it, use `CTRL+C`, `CTRL+C`.

How does this work? How does this not blow up with a StackOverflow error?

## Thinking About Tail Recursion

A pseudo-code way of expressing what's happening is to understand that it is performing a GOTO-like operation where a new value is used for the argument.

```
0x01    defmodule StackTest do
0x02      def recurse(arg0) do
0x03        IO.puts "Step #{arg0}"
0x04        GOTO 0x03, arg0 is (arg0 + 1)
0x05      end
0x06    end

0x07    StackTest.recurse(0)
```

### Thinking Tip: An Optimization

Tail Recursion is also called Tail Call Optimization. When the **last statement** is a recursive call, it gets *optimized*.

Another way of thinking about it is imagining it gets converted into a `while` statement like you see in other languages.

Thinking about it as a `goto` or a `while` statement both work as a mental model. Internally however, it's closer to an Assembler JMP (ie. jump) command.

If you unwrap what's happening here, the instructions essentially ends up looking like this...

```
StackTest.recurse(0)
StackTest.recurse(1)
StackTest.recurse(2)
StackTest.recurse(3)
StackTest.recurse(4)
StackTest.recurse(5)
# ...
```

Where a new input value is used and it is run again. There is no call stack being generated!

## Creating a Recursion Error

You *can* create a recursion error **when the last line is NOT a recursive call** When there are more instructions to perform following a recursive call, tail recursion cannot be used. The BEAM needs to track the call stack so it can return and execute the remaining statements. However, it isn't really a StackOverflow error, it's a memory allocation error.

If you'd like to see it in action, this code will do it. It will take a lot longer to run before it errors. **TIP:** Before you run it, you might want to read what it does first!

```
defmodule StackTest do

  def recurse(num) do
    IO.puts "Step #{num}"
    recurse(num + 1)
    IO.puts "Another instruction but you'll never see it."
  end

end
StackTest.recurse(0)
```

Finally it errors because it can't allocate enough RAM. You may not *actually* want to run it, since it will crash at some point, then generate a **very large** `erl_crash.dump` crash dump file. I killed mine before it finished writing the file and the file was **already 27 GB in size** and growing!

```
iex> StackTest.recurse(0)

# ...

eheap_alloc: Cannot allocate 6801972448 bytes of memory (of type "heap").

Crash dump is being written to: erl_crash.dump...
```

A number of things in Elixir use and depend upon tail recursion. It is an important feature and behavior of the BEAM. Know what it is so you can use it well!

---

> When the last command is a recursive call, then tail-recursion is used and it can efficiently loop forever.

---

# Enum Looping Shortcuts

Looping using recursion typically won't be your first choice when you need to loop. Sometimes using explicit recursion really is the best approach for a situation and it is very beneficial to understand how to do it. However, Elixir provides a number of helpful shortcut functions we use when we need to loop. Many of them are in the `Enum` module. The `Enum` module defines many functions for working with things that are "enumerable". It can work on lots of things, not just lists. That's why these functions aren't on the `List` module.

The top 3 functions we want to start with are:

- `Enum.each/2` – Invokes a function for each element in the enumerable. Returns `:ok`.
- `Enum.map/2` – Returns a list where each element is the result of invoking a function on each element of enumerable.
- `Enum.reduce/3` – Invokes a function for each element in the enumerable with an accumulator.

## A common mistake with `Enum.each/2`

Coming from an imperative language, it might be your knee jerk reaction to reach for `Enum.each/2`. Doing that would be a mistake. It probably isn't what you *actually* want.

Let's look at a Javascript example of how many people use OOP & imperative patterns to loop. Even if Javascript isn't your thing, something like this probably feels familiar.

```
var data = [
    {name: "Customer 1", total: 0},
    {name: "Customer 2", total: 100},
    {name: "Customer 3", total: 200},
];

data.forEach(function(customer) {
    customer.total += 50;
});

data
//=> [{name: "Customer 1", total: 50},
//=> {name: "Customer 2", total: 150},
//=> {name: "Customer 3", total: 250}]
```

People coming to functional languages like Elixir get confused and frustrated when they try to do something similar in Elixir. Let's see how people might initially try to do the same operation in Elixir.

```
data = [
  %{name: "Customer 1", total: 0},
  %{name: "Customer 2", total: 100},
  %{name: "Customer 3", total: 200},
];

Enum.each(data, fn(customer) ->
  Map.put(customer, :total, customer.total + 50);
end)

data
#=> [
#=>   %{name: "Customer 1", total: 0},
#=>   %{name: "Customer 2", total: 100},
#=>   %{name: "Customer 3", total: 200}
#=> ]
```

The developer trying this probably got pretty frustrated. To top it all off, the resulting data *isn't even updated* with the incremented totals!

Let's assume the developer is trying to create a `1:1` conversion of how they'd solve the problem in Javascript but now using Elixir.

First of all, they'd try to use `+=`, but Elixir doesn't have a `+=` operator because by definition, that mutates a variable. Bam! <Head hits wall>

Second, then the developer might write something like `customer.total = customer.total + 50`. Attempting this will cause a syntax error because `customer.total` is not a *pattern* and it's on the left side of the **match operator** `=` which expects `pattern = data`. Bam! <Head hits wall>

Lastly, after figuring out what they *couldn't* do, they tried `Map.put/3` on the data and it *still* didn't work. Bam! Man, I feel for that poor, frustrated developer. Thankfully, that won't be you!

The reason the last `Map.put/3` operation didn't work as expected is because it return a *new* map with the value updated. The code above creates an updated map and then promptly discards the newly created map. Unintentionally of course. The developer making this mistake was still expecting data mutation. They reached for the `Enum.each/2` function because that seemed to match with what they would do in an imperative language.

What the developer *actually* wanted was `Enum.map/2` .

## Using `Enum.map/2`

If you want to update the things in a list, then you probably want `Enum.map/2` . Remember in Elixir we are working with immutable data. So if I want to update a list of numbers, what I *actually* want is a ***new*** list of numbers that are updated in the way I want.

To update a list of numbers adding 10 to every number, I could write:

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

updated = Enum.map(data, fn(val) -> val + 10 end)

updated
#=> [11, 12, 13, 14, 15]
```

In this example, `Enum.map/2` takes two arguments. The first is an enumerable list. The second is a function to perform on each element in the list. The result of `Enum.map/2` is a *new* list with the *same number of elements* as the list going in, but each element in the new list is the value returned by the function.

## Exercise #1 – Fixing the Example

The example the frustrated developer was working on needs to get fixed! We can't leave the code unfinished! Use `Enum.map/2` to create an `updated_data` variable. Your goal is to update the `total` in each map by increasing it by `50` . Use IEx to try out your solution. You can write it in a text editor for convenience.

```
data = [
  %{name: "Customer 1", total: 0},
  %{name: "Customer 2", total: 100},
  %{name: "Customer 3", total: 200},
];
```

**Showing Solution**

### A working example that uses `Enum.map/2` . Note that `updated_data` holds the desired result and that `data` still references the original unmodified list.

```
data = [
  %{name: "Customer 1", total: 0},
  %{name: "Customer 2", total: 100},
  %{name: "Customer 3", total: 200},
];

updated_data =
  Enum.map(data, fn(customer) ->
    Map.put(customer, :total, customer.total + 50)
  end)

updated_data
#=> [
#=>   %{name: "Customer 1", total: 50},
#=>   %{name: "Customer 2", total: 150},
#=>   %{name: "Customer 3", total: 250}
#=> ]
```

## Anonymous function shorthand

Remember that Elixir supports a shorthand for writing anonymous functions. This is relevant because all the `Enum` functions we are working with take a function as an argument. It is very common to see and use the shorthand version. It's worth spending some time getting comfortable with it. The following are equivalent:

```
Enum.map([2], fn(val) -> val * 2 end)
#=> [4]

Enum.map([2], &(&1 * 2))
#=> [4]
```

In the shorthand function, the first `&` means "this is an anonymous function". The `&1` means, "this is the first argument passed to the function". Similarly, `&2` stands in for the second argument, `&3` the third argument and so forth. To compare the two functions, the `&` means `fn` and `&1` is the same as `val`. Also notice that no `end` is used for the anonymous function.

Another point to note is that parenthesis are optional in Elixir. So the following two statements are equivalent as well.

```
Enum.map([2], &(&1 * 2))
#=> [4]

Enum.map([2], & &1 * 2)
#=> [4]
```

The reason I point this out is because the Elixir formatter removes parenthesis when they aren't needed. So you may end up seeing this on your *own* code even if you didn't write it that way.

## Exercise #2 – Fixed Example using Shorthand

Rewrite the fixed `Enum.map/2` version from Exercise #1 using the function shorthand. Use IEx to try out your solution. You can write it in a text editor for convenience.

**Showing Solution**

### The solution from Exercise #1 updated to use the anonymous function shorthand.

```
data = [
  %{name: "Customer 1", total: 0},
  %{name: "Customer 2", total: 100},
  %{name: "Customer 3", total: 200},
];

updated_data = Enum.map(data, &Map.put(&1, :total, &1.total + 50))

updated_data
#=> [
#=>   %{name: "Customer 1", total: 50},
#=>   %{name: "Customer 2", total: 150},
#=>   %{name: "Customer 3", total: 250}
#=> ]
```

## Using local functions

Another approach is to use local module functions as the function being used in the `Enum.map/2` call. Here's an example of what that might look like:

```
defmodule Playing do
  def values_doubled(list) do
    Enum.map(list, &doubler/1)
  end

  defp doubler(val), do: val * 2
end

Playing.values_doubled([1, 2, 3])
#=> [2, 4, 6]
```

The module defines a function `doubler/1` that takes a single value, multiplies it by 2 and returns the result. The function `values_doubled/1` passes `doubler` as a **function reference** which identifies the function using its **name** and **arity**. Writing the function this way looks similar to `&doubler(&1)` but this form is actually **executing the function**, not passing a reference to it. In total it looks like this:

```
defmodule Playing do
  def values_doubled(list) do
    Enum.map(list, &doubler(&1))
  end

  defp doubler(val), do: val * 2
end

Playing.values_doubled([1, 2, 3])
#=> [2, 4, 6]
```

A benefit to using a function declared in a module (either passed by reference or executed directly) is that it makes it easier to use pattern matching in the function doing the work. As a slightly absurd example, this works:

```
defmodule Playing do
  def values_doubled(list) do
    Enum.map(list, &doubler(&1))
  end

  defp doubler(val) when is_number(val), do: val * 2
  defp doubler(val) when is_binary(val), do: val <> val
end

Playing.values_doubled([1, 2, 3])
#=> [2, 4, 6]

Playing.values_doubled(["Hi", "Hello"])
#=> ["HiHi", "HelloHello"]
```

I hope you get the point that you have options with what you pass to an `Enum.map/2` call for the function it uses to operate on the elements of an enumerable. Of course, this applies to other `Enum` calls as well.

## Ranges

This is a good time to talk about another data type in Elixir. It's called a " range". A range is most often expressed using this syntax: `1..10` . This range represents the range of numbers from 1 up to and including 10. When we want to perform an operation a specific number of times, then a range can be helpful! Let's see an example:

```
Enum.each(1..5, fn(num) ->
  IO.puts "Processed num #{num}"
end)
#=> Processed num 1
#=> Processed num 2
#=> Processed num 3
#=> Processed num 4
#=> Processed num 5
#=> :ok
```

It is worth pointing out here that the no matter the size of the range (ex `1..1_000_000`), an `Enum` module function will be memory efficient because it uses logic to iterate. Stated another way, an `Enum` function doesn't fully expand the range into a list, it uses logic and the range's endpoint values to iterate.

If you want to see what a range includes, you can use `Enum.to_list/1` to expand it.

```
Enum.to_list(1..5)
#=> [1, 2, 3, 4, 5]

Enum.to_list(1..0)
#=> [1, 0]

Enum.to_list(1..-5)
#=> [1, 0, -1, -2, -3, -4, -5]

Enum.to_list(1..1)
#=> [1]
```

Notice that the range can reverse order and go from high to low as well. The important point with ranges and `Enum` functions is that you can never get an empty list. A range of `0..0` expands to `[0]`. So if you want to use a range and a variable for the high-end, you may need to explicitly handle receiving a `0`. An example will help make this clear.

```
defmodule Playing do
  def counting(up_to_number) do
    Enum.each(1..up_to_number, fn(num) ->
      IO.puts "Counting: #{num}"
    end)
  end
end

# this is as you'd expect...
Playing.counting(5)
#=> Counting: 1
#=> Counting: 2
#=> Counting: 3
#=> Counting: 4
#=> Counting: 5
#=> :ok

# this may surprise you...
Playing.counting(0)
#=> Counting: 1
#=> Counting: 0
#=> :ok
```

The lesson to take away is that using a range with `Enum` functions work best when the range is developer declared. Not necessarily when taking arbitrary input. The alternative is to explicitly handle receiving something like a `0`.

## Using `Enum.each/2`

So what is `Enum.each/2` actually good for? It's great when you want to iterate and perform some operation but don't care about the result. I mean you don't care about the result of `Enum.each/2` because it always returns `:ok`. You may want the *side-effects* that you create during the iterations, but you aren't creating a modified list and returning it.

This is a great time to practice doing that!

## Exercise #3 – Creating Customers

A previous exercise using recursion can now be re-implemented using `Enum.each/2`. The objective here is to create a set number of new customer entries. Use the function `CodeFlow.Fake.Customers.create/1` to perform the customer create operation. In order to create a valid customer, you must at least provide a name. Again, the focus here is on loop control.

```
mix test test/enum_shortcut_test.exs --only describe:"create_customers/1"
```

**Showing Solution**

### To ensure expected behavior, handle receiving a `0` and do nothing.

```
  def create_customers(0), do: :ok

  def create_customers(number) do
    Enum.each(1..number, fn(num) ->
      # for simplicity, not handling a failed create
      {:ok, _customer} = Customers.create(%{name: "Customer #{num}"})
    end)
  end
```

## A good place to pause

We have already covered quite a bit! You may feel a bit overwhelmed. However don't worry about it! These are a lot of new concepts you're being exposed to.

Before we end this lesson, let's review a bit about what we just covered.

- We saw some common mistakes people make when first using `Enum`.
- We looked at 2 important `Enum` functions.
    - Enum.each/2
    - Enum.map/2
- We again covered using anonymous functions.
- We looked at how to use local functions.
- We also introduced Ranges.

In the next lesson, we'll go deeper with `Enum`. We'll look at `Enum.reduce/3` and see different things we can do with anonymous functions. We will also see how the `Enum` functions can enumerate over more than just a list!

# Enum Part 2

In Part 1, you were introduced to the `Enum` module. We also covered 2 really important `Enum` functions.

- Enum.each/2
- Enum.map/2

Now we will continue looking at `Enum`. We'll start by looking the powerhouse Enum.reduce/3 function. We then re-visiting anonymous functions because they are used so frequently with `Enum`.

## Using `Enum.reduce/3`

Enum.reduce/3 is a powerful function. It may feel a little foreign to people new to Elixir at first. Particularly if you haven't use a "reduce" function in other languages or frameworks. However, this function is definitely worth understanding.

Reduce executes the function for each item in the enumerable. The special power is that it has an accumulator and the result of the reduce function is the accumulated value. Remember the recursion exercises? Passing an accumulator solves a number of looping situations. Internally, reduce is implemented using the recursion pattern. So using reduce can be a handy shortcut!

Let's review the recursive way we summed a list of numbers:

```
defmodule Playing do
  def sum([num | rest], acc) do
    sum(rest, acc + num)
  end

  def sum([], acc), do: acc
end

data = [1, 2, 3, 4, 5]
Playing.sum(data, 0)
#=> 15
```

Now let’s see the same solution using `Enum.reduce/3` .

```
data = [1, 2, 3, 4, 5]
Enum.reduce(data, 0, fn(num, acc) ->
  num + acc
end)
#=> 15
```

The 1st argument is the list being enumerated. The 2nd argument is the **initial value** of the accumulator or `acc` . The anonymous function receives 2 arguments. The first is a value from the list (ie `num` ) and the second is the current accumulated value (ie `acc` ). The result of the anonymous function becomes the new accumulator value.

We could further condense this using the anonymous function shorthand.

```
data = [1, 2, 3, 4, 5]
Enum.reduce(data, 0, &(&1 + &2))
#=> 15
```

## Exercise #4 – Compute Order Total

This is re-implementing a recursion exercise. Using `Enum.reduce/3` , process a list of `OrderItem` structs to compute an order total. Refer to the following structs for details as needed:

- lib/schemas/order_item.ex
- lib/schemas/item.ex

The order total is computed as an Item’s price * the quantity being ordered. Summing all of those together for the total order price.

Review the tests to see the data samples being tested.

```
mix test test/enum_shortcut_test.exs --only describe:"order_total/1"
```

**Showing Solution**

The `reduce` solution is more to the point than the recursion version. Also note that the Elixir formatter removed the extra parenthesis on the anonymous function as it isn’t required.

```
  def order_total(order_items) do
    Enum.reduce(order_items, 0, fn %OrderItem{} = order_item, total ->
      order_item.quantity * order_item.item.price + total
    end)
  end
```

## Anonymous function clauses

When working with the `Enum` module, we are frequently also working with anonymous functions. We talked about using the shorthand version which can be very condensed. There is a benefit to using the longer form version of anonymous functions as well. It supports ***multiple clauses*** which we can use for pattern matching. Yes. You read that right. Let's see it to understand it.

The structure of it looks like this:

```
fn
  pattern_clause_1 ->
    expression_when_clause_1_matches
  pattern_clause_2 ->
    expression_when_clause_2_matches
  pattern_clause_3 ->
    expression_when_clause_3_matches
end
```

Here's a working version that you can edit and play with.

```
fun =
  fn
    0 ->
      :ok
    nil ->
      "You passed nil!"
    num when is_number(num) ->
      "#{num} + 1 = #{num + 1}"
  end

fun.(0)
#=> :ok

fun.(1)
#=> "1 + 1 = 2"

fun.(2)
#=> "2 + 1 = 3"

fun.(nil)
#=> "You passed nil!"
```

With this new understanding, let's apply it to *conditionally* increment an accumulator in a `reduce` function.

## Exercise #5 – Counting Active

One of the recursion exercises we did previously used function clause pattern matching to identify when we should conditionally increment the accumulator. Using 2 different ways, let's do that with `Enum.reduce/3` just to get the practice. In both cases, the goal is the same. Given a list of Customers, count how many are active. A Customer has an `active` boolean flag. Conditionally increment the accumulator counter only when a Customer is active. Refer to the following as needed:

- `lib/schemas/customer.ex` – Defines the customer struct
- `test/recursion_test.exs` – Shows data examples your code needs to process

There are tests for this exercise here:

```
mix test test/enum_shortcut_test.exs --only describe:"count_active/1"
```

## Using an anonymous function with clauses

Using the approach we just discussed with anonymous functions supporting multiple pattern clauses, write the Elixir code that makes the tests pass.

**Showing Solution**
**When to use this approach?**

This approach works well when the logic in each function clause is pretty simple and direct.

```
def count_active(customers) do
  Enum.reduce(customers, 0, fn
    %Customer{active: true}, total -> total + 1
    _customer, total -> total
  end)
end
```

## Using module function clauses

Let’s solve the same problem in a different way. Write private `defp do_count_active` function clauses that use pattern matching to conditionally increment the accumulator. Your call to `Enum.reduce/3` can pass the function by reference (meaning the version where you specify the arity).

**Showing Solution**
**When to use this approach?**

### This approach works well when there are a number of clauses, the clauses are complex or the logic being perform is more involved.

Side note: It would be equally valid to directly execute the function using `&do_count_active(&1)` .

```
def count_active(customers) do
  Enum.reduce(customers, 0, &do_count_active/2)
end

defp do_count_active(%Customer{active: true}, total), do: total + 1
defp do_count_active(_customer, total), do: total
```

## Reduce accumulates more than a number

Up to this point, all the reduce examples have been focused on accumulating a number like a count or a total value. It is important to realize that you can accumulate *any* data structure that makes sense for your problem.

As an example, let’s take a list of maps where each map has a `:status` key. We want to group all the maps by their status values. Let’s look at the example.

```
data = [
  %{name: "Tammy", status: "pending"},
  %{name: "Joan", status: "active"},
  %{name: "Timothy", status: "closed"},
  %{name: "Alan", status: "active"},
  %{name: "Nick", status: "active"}
]

results =
  Enum.reduce(data, %{}, fn(%{status: status} = item, acc) ->
    case Map.fetch(acc, status) do
      {:ok, items} ->
        Map.put(acc, status, [item | items])

      :error ->
        Map.put(acc, status, [item])
    end
  end)

results
#=> %{
#=>   "active" => [
#=>     %{name: "Nick", status: "active"},
#=>     %{name: "Alan", status: "active"},
#=>     %{name: "Joan", status: "active"}
#=>   ],
#=>   "closed" => [%{name: "Timothy", status: "closed"}],
#=>   "pending" => [%{name: "Tammy", status: "pending"}]
#=> }
```

Notice that the initial value for the accumulator (the 2nd argument in `Enum.reduce/3` ) is an empty map `%{}` . The current accumulated value is passed to our function for each item. If the the status key already exists, we add our item to the list.

The message here is that you can use "reduce" to build and transform data structures in powerful ways. You can "reduce" into other data structures, not just to count up totals.

## Enumerate over non-lists

We've covered lots of examples of enumerating over a list or a range. Did you know you can also enumerate over a map? When you do this, the "item" passed to your function is a tuple of `{key, value}` . Let's see an example:

```
data = %{name: "Howard", age: 32, email: "howard@example.com"}

Enum.each(data, fn({key, value}) ->
  IO.puts("The #{key} = #{inspect(value)}")
end)
#=> The age = 32
#=> The email = "howard@example.com"
#=> The name = "Howard"
#=> :ok
```

You can also make a custom struct work with `Enum` . To do this, you need to implement the Enumerable protocol. This means you implement 4 functions for your data structure and then it can mapped, reduced, and more.

## `Enum.map/2` and pipelines

The Pipe Operator takes in "data" or an "expression" on the left. A **list** is data too! Let's see an example of processing a small list using a pipeline. This could be a list of products to order, customers to bill, etc.

This sample is already decorated with several `IO.inspect` calls to help see into the transformations at each step.

```
defmodule PipePlay do

  def lists do
    [1, 2, 3, 4, 5, 6]
    |> IO.inspect()
    |> Enum.map(fn(num) -> num * 10 end)
    |> IO.inspect()
    |> Enum.map(fn(num) -> num + 1 end)
    |> IO.inspect()
    |> Enum.map(fn(num) -> to_string(num) end)
  end

end
PipePlay.lists()
#=> [1, 2, 3, 4, 5, 6]
#=> [10, 20, 30, 40, 50, 60]
#=> [11, 21, 31, 41, 51, 61]
#=> ["11", "21", "31", "41", "51", "61"]
```

The list is piped through 3 different calls to `Enum.map/2` where different functions are being applied to the list at each step. In this case we are using anonymous functions. When we re-write the anonymous functions to use the abbreviated `&` syntax, it makes our example code easier to look at. It removes a lot of the visual noise of parenthesis, arrows, and `end`'s.

```
defmodule PipePlay do

  def lists do
    [1, 2, 3, 4, 5, 6]
    |> IO.inspect()
    |> Enum.map(& &1 * 10)
    |> IO.inspect()
    |> Enum.map(& &1 + 1)
    |> IO.inspect()
    |> Enum.map(&to_string(&1))
  end

end
PipePlay.lists()
#=> [1, 2, 3, 4, 5, 6]
#=> [10, 20, 30, 40, 50, 60]
#=> [11, 21, 31, 41, 51, 61]
#=> ["11", "21", "31", "41", "51", "61"]
```

## Many `Enum` convenience functions

There are many functions in the `Enum` module. Nearly all of them are convenience functions and you should turn to them first. Please note that many of the `Enum` functions are actually implemented using recursion. That's how powerful the recursive list processing pattern is.

The `Enum` module is an essential module to get comfortable with. Here are a few highlights beyond what we already covered:

- Enum.all?/2
- Enum.any?/2
- Enum.filter/3
- Enum.find/3
- Enum.sort/2

## Enum Recap

We covered a lot through Part 1 and 2! One important point to take is that we did everything here **using immutable data**! That's so cool! I hope you see that you can still accomplish what you need to do in a non-mutating way. You may need to change the way you think about things, but now you *know* you can do it!

Let's review the three `Enum` module functions that are commonly used when needing to loop.

- Enum.each/2 – You will likely end up using Enum.each/2 the least of the three. Since each has no meaningful return value and you can't mutate anything inside the loop, there aren't as many opportunities to actually use it. It is only used for creating other system side-effects.
- Enum.map/2 – When you want to "modify" a list of things, this is your go-to function. It returns a new list where each item had your function applied to it. It is probably the function you want if you are most familiar with mutating and imperative looping.
- Enum.reduce/3 – A real power house function. It iterates an enumerable applying a function. However, rather than returning a new list, it returns the accumulator. Your function defines what to accumulate and how to do it.

## Other topics covered

- **Ranges** – A range is a number sequence expressed like this: 1..10 . It is a good tool for looping a desired number of times. It works best when the bounds are developer defined.
- **Anonymous functions** – All the Enum functions we looked at take something to enumerate and a function to apply to each item. Anonymous functions are very commonly used for these. We covered more on how we can declare and use anonymous functions in Elixir.
  - **Shorthand syntax** – Reminder of the shorthand you can use for anonymous functions. fun = &IO.puts(&1)
  - **Pass by reference** – Can pass a function by it's reference fun = &IO.puts/1 . Works great for local functions that do more pattern matching or handle more complex logic.
  - **Multiple clauses** – Anonymous functions support multiple pattern matching clauses! Work great for a small number of clauses and simple logic.
- Enum.reduce/3 – Can accumulate more than a number. You can create lists, maps, and other things.
- **Enumerate non-lists** – You can enumerate over maps and more, not just lists.
- **Lists can be pipelined** – Enum.map/2 is a great tool for that.

# "for" comprehensions

The for comprehension is a powerful language feature. It is like a "Swiss Army Knife". It includes many features and options. At its simplest, it may remind you of Enum.map/2 .

```
for num <- [1, 2, 3, 4, 5] do
  num * 2
end
#=> [2, 4, 6, 8, 10]
```

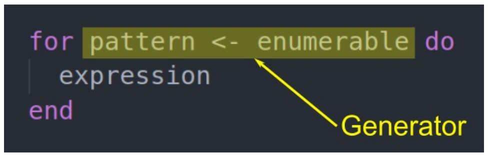
The above for comprehension returns a list where each element is the return value of the do/end block. However, instead of passing a function like in Enum.map/2 , you provide the function body code *inside* the do/end block.

## Basic for anatomy

The basic layout of the for comprehension is this:

```
for pattern <- enumerable do
  expression
end
```

The part `pattern <- enumerable` is called a “generator”.



```
for pattern <- enumerable do
  expression
end
```

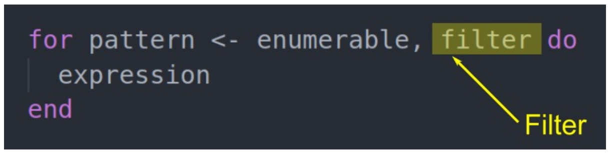
The fact that this has its own special name tells you that it might do something different. What happens when the pattern *doesn’t* match?

```
for num when is_number(num) <- ["abc", 123, "def", 456] do
  num
end
# [123, 456]
```

Interestingly, when the pattern *doesn’t match* an entry in the list it gets filtered out!

## Filter clause

The `for` comprehension also includes support for a *separate and dedicated “filter” clause*. What’s cool about the filter clause is that it can perform operations that are not allowed in a guard clause. Here’s where it goes in the comprehension.



```
for pattern <- enumerable, filter do
  expression
end
```

Similar to the pattern match, if the filter clause doesn’t match, the entry is filtered out. Let’s look at a working example to understand it better.

```
for value when is_binary(value) <- ["abc", 123, "def", 456], String.contains?(value, "e") do
  value
end
#=> ["def"]
```

You may be asking, “Why can’t I just conditionally add the result with the “expression” portion?” If you try that here, you see that the entry is included in the resulting list, but no explicit value was returned so it includes a `nil` value. **This isn’t the behavior you intended.**

```
for value when is_binary(value) <- ["abc", 123, "def", 456] do
  if String.contains?(value, "e") do
    value
  end
end
#=> [nil, "def"]
```

The filter clause is applied *before* the `do/end` block ever executes. Depending on the pattern match and any filter clauses, the `do/end` block may *never* execute.

Comprehensions ignore or discard all elements where the filter expression returns `false` or `nil`. All other values pass and are included.

Here’s the order that things happen:

1. Element of enumerable is checked against pattern
2. If pattern matches, filter constraint is applied
3. If filter passes, `do/end` block is executed

## Exercise #1 – Preferred user points

You are writing a game for a client and during game play, the client wants you to give preferential treatment to users that have characteristics important to them. Yes, it’s completely unfair… but it’s their game. The client really likes the letter combination “uc” in a name. They think of the words “…you see…” and just find it humorous. Yeah, weird client, I know!

So during game play, your function `Comprehension.award_unfair_points/2` will be called with a list of users. You should first filter out any users that are not active. An additional filter should be applied if their name contains the letters “uc”. So active users with the letter combination should have their points incremented by the given number of points and this privileged set of users should be returned in a list.

The tests for this exercise are here:

```
mix test test/comprehension_test.exs --only describe:"award_unfair_points/2"
```

**Showing Solution**

This solution performs the user update using the struct annotation. This is not necessary and it is valid to use `Map.put/3` as well.

```
  def award_unfair_points(users, points) do
    for %User{active: true} = user <- users, String.contains?(user.name, "uc") do
      %User{user | points: user.points + points}
    end
  end
```

## Multiple generators

The `for` comprehension supports using multiple generators. Remember the “generator” part is: `n <- [1, 2, 3]`. This is what it looks like with two generators. This example create a small multiplication table:

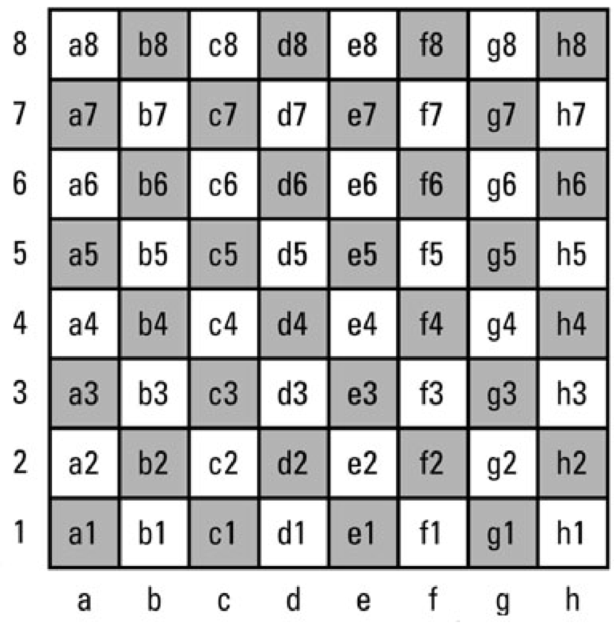
```
for x <- 1..3,
    y <- 1..3 do
  "#{x} * #{y} = #{x*y}"
end
#=> [
#=>   "1 * 1 = 1",
#=>   "1 * 2 = 2",
#=>   "1 * 3 = 3",
#=>   "2 * 1 = 2",
#=>   "2 * 2 = 4",
#=>   "2 * 3 = 6",
#=>   "3 * 1 = 3",
#=>   "3 * 2 = 6",
#=>   "3 * 3 = 9"
#=> ]
```

And yes, you *can* do more than 2 generators. Feel free to play with that.

Multiple generators create a Cartesian Product or a cross-join. It goes through each `x` value when `y` is `1`. Then it goes through each `x` value again when `y` is now `2` and so on.

## Exercise #2 – Build a chess board

In this example you need to build a Chessboard. A Chessboard is an 8×8 grid but each square has a unique name making them clearly addressable.



Using a `for` comprehension, build a list of the squares. The structure of each square will be a map that looks like this: `%{col: "a", row: 1, name: "a1"}` . The function to create is `Comprehension.build_chessboard/0` . It should return a list of 64 square entries. There are unit tests to check that the board is built correctly.

The tests for this exercise are here:

```
mix test test/comprehension_test.exs --only describe:"build_chessboard/0"
```

**Showing Solution**

You can solve this with either the numbers or the columns first. In my solution I did letters first because I wanted to see it output the first row fully.

```
  def build_chessboard() do
    for row <- 1..8,
        col <- ["a", "b", "c", "d", "e", "f", "g", "h"] do
      %{col: col, row: row, name: "#{col}#{row}"}
    end
  end
```

## Special options

The `for` comprehension has a few additional options that add some extra behavior. Let's cover those now.

## Into and Uniq

All of the examples up to now show the `for` comprehension returning a list as the result. Using the `:into` option, it can return any data structure that supports the Collectable protocol. The Enum.into/2 function does the work and you can read more about how it works on the documentation there. Just be aware that the `for` comprehension will do that for you when you use this option. This is what it looks like:

```
for n <- [a: 1, b: 2, c: 3], into: %{}, do: n
#=> %{a: 1, b: 2, c: 3}
```

This converts a keyword list into a map. The `into: %{}` tells it what to convert it into and provides an initial value.

The `uniq: true` option will guarantee the results are only added to the collection if they were not returned before.

You can read more about the `:into` and `:uniq` options in the documentation.

## Reduce

This option works like `Enum.reduce/3`. When used, the comprehension returns the accumulator, not a list. To use it, we pass in the initial value for the accumulator. Doing a reduce changes the syntax of the comprehension. Because we need to keep receiving the accumulator as it iterates through the enumerable, there is a special form that uses `->` to do it. Seeing it in action will help:

```
for n <- 1..5, reduce: 0 do
  acc ->
    acc + n
end
#=> 15
```

The option `reduce: 0` says we *want* to reduce and provides the **initial value**. In this case, we start with a `0`. Then the `acc ->` is how we receive the accumulator being passed in. Our return value will become the new accumulator value.

You can read more about the `:reduce` option in the documentation.

## Exercise #3 – Total team points

Our game has users who have received points. Users are part of a team and we need to be able to total up all the points awarded to the team. Create the function `Comprehension.team_points/1` which receives a list of users. The team may have lost users along the way and they became “inactive”. Filter out inactive users. Return the sum of all the active user’s points. Use the `for` comprehension and the `:reduce` option to do this.

The tests for this exercise are here:

```
mix test test/comprehension_test.exs --only describe:"team_points/1"
```

**Showing Solution**

```
  def team_points(users) do
    for %User{active: true} = user <- users, reduce: 0 do
      acc ->
        acc + user.points
    end
  end
```

## `for` comprehension recap

Using a `for` comprehension can be more elegant than the equivalent version using `Enum` functions. There are still functions and situations where other solutions work better so this isn’t the ultimate solution.

A few key points to remember:

- Basic usage is like `Enum.map/2`
- Generators with pattern matching filter out non-matches
- Filter clause can be added for additional filtering that doesn’t work in a guard clause
- Multiple generators can be used to create a Cartesian Product
- Options like `:into` and `:reduce` add extra data transformation behavior

The `for` comprehension is powerful and really is like a Swiss Army Knife. Like a Swiss Army Knife that includes a screwdriver feature, it can operate as a screwdriver but a dedicated or set of varied screwdrivers will work better at times. Still, it includes many features that work well together and it is a valuable addition to your toolbox.

The `for` comprehension includes the features and abilities of a number of `Enum` functions. Let’s review that list briefly and how they are used:

- `Enum.map/2` – By using a basic generator without any matching
- `Enum.filter/2` – Through both the pattern and the “filter” clause
- `Enum.into/2` – Through the in `into: collectable` option
- `Enum.uniq/1` – Through the `uniq: true` option
- `Enum.reduce/3` – Through the `reduce: initial_acc` option and the special `->` clause

# Enum vs Stream

The Enum and Stream modules have some of the same functions like `each/2`, `map/2`, `filter/2` and more. What's the difference? To help see the difference, let's look at a simple example. First we'll look at how `Enum` works and then compare it to `Stream`.

## Enum and piping a list

When we pipe a list using the `Enum.map/2` function, we can peek into the process by using `IO.inspect/2` to see the order of execution and what each step returns.

This example uses `IO.inspect/2` inside the functions being executed and between each step. With this we can better visualize what is happening.

```
defmodule Playing do
  def peeking_into_enum() do
    [1, 2, 3, 4, 5]
    |> IO.inspect(label: "ORIGINAL DATA")
    |> Enum.map(fn(num) -> IO.inspect(num + 10) end)
    |> IO.inspect(label: "STEP 1 RESULT")
    |> Enum.map(fn(num) -> IO.inspect(num * 2) end)
    |> IO.inspect(label: "STEP 2 RESULT")
    |> Enum.map(fn(num) -> IO.inspect(to_string(num)) end)
    |> IO.inspect(label: "STEP 3 RESULT")
  end
end
Playing.peeking_into_enum
#=> ORIGINAL DATA: [1, 2, 3, 4, 5]
#=> 11
#=> 12
#=> 13
#=> 14
#=> 15
#=> STEP 1 RESULT: [11, 12, 13, 14, 15]
#=> 22
#=> 24
#=> 26
#=> 28
#=> 30
#=> STEP 2 RESULT: [22, 24, 26, 28, 30]
#=> "22"
#=> "24"
#=> "26"
#=> "28"
#=> "30"
#=> STEP 3 RESULT: ["22", "24", "26", "28", "30"]
```

This makes it clear that it works probably exactly like you expected. It visits each element in the list and runs the function on the value. At the end of each step, we have a new list with the transformation applied.

This is called an **eager** evaluation.

> In eager evaluation, an expression is evaluated as soon as it is bound to a variable.
>
> https://en.wikipedia.org/wiki/Eager_evaluation

This was probably the behavior you expected because it is common in most traditional programming languages.

## When is "eager" a problem?

Eager evaluation is the default approach used in Elixir. Why would I want a different strategy? When would “eager” be a problem?

Eager evaluation causes problems when the data is very large, possibly even unbounded. Using `Enum` for each step that we evaluate makes everything happen in RAM. When working with large data sets this can be a problem.

The `Stream` module gives us an elegant way to do **lazy** evaluations.

> **Lazy evaluation**, or **call-by-need** is an evaluation strategy which delays the evaluation of an expression until its value is needed.
>
> https://en.wikipedia.org/wiki/Lazy_evaluation

---

> Eager evaluation causes problems when the data is very large, possibly even unbounded.

---

## Stream and piping a list

Let’s adapt the `Enum` code from above to now use `Stream` and see the difference.

```
defmodule Playing do
  def peeking_into_stream() do
    [1, 2, 3, 4, 5]
    |> IO.inspect(label: "ORIGINAL DATA")
    |> Stream.map(fn(num) -> IO.inspect(num + 10) end)
    |> IO.inspect(label: "STEP 1 RESULT")
    |> Stream.map(fn(num) -> IO.inspect(num * 2) end)
    |> IO.inspect(label: "STEP 2 RESULT")
    |> Stream.map(fn(num) -> IO.inspect(to_string(num)) end)
    |> IO.inspect(label: "STEP 3 RESULT")
  end
end
Playing.peeking_into_stream
#=> ORIGINAL DATA: [1, 2, 3, 4, 5]
#=> STEP 1 RESULT: #Stream<[
#=>   enum: [1, 2, 3, 4, 5],
#=>   funs: [#Function<48.51129937/1 in Stream.map/2>]
#=> ]>
#=> STEP 2 RESULT: #Stream<[
#=>   enum: [1, 2, 3, 4, 5],
#=>   funs: [#Function<48.51129937/1 in Stream.map/2>,
#=>    #Function<48.51129937/1 in Stream.map/2>]
#=> ]>
#=> STEP 3 RESULT: #Stream<[
#=>   enum: [1, 2, 3, 4, 5],
#=>   funs: [#Function<48.51129937/1 in Stream.map/2>,
#=>    #Function<48.51129937/1 in Stream.map/2>,
#=>    #Function<48.51129937/1 in Stream.map/2>]
#=> ]>
```

Wow. That output looks really different! A big thing to note is it hasn’t actually executed any of the functions!

At each step, instead of returning a transformed list where each element had the function applied to it, we get a data structure.

The Stream data structure contains the thing to enumerate as `enum` and it builds up a list of functions that we want applied in `funs` . Notice that at each step, the only thing that changes is that another function was added to the list.

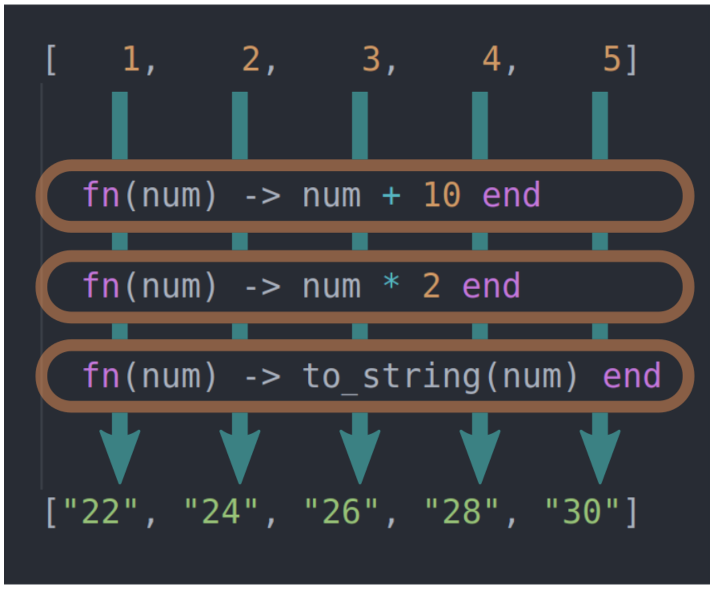
How do we make a Stream actually do work?

The Stream data structure also implements the Enumerable protocol. This means we can use the `Enum` module to use our **lazy** definition and actually apply some demand to the stream making it evaluate. To do this, we can add one more pipe that pipes our stream into `Enum.to_list/1` .

```
defmodule Playing do
  def peeking_into_stream() do
    [1, 2, 3, 4, 5]
    |> IO.inspect(label: "ORIGINAL DATA")
    |> Stream.map(fn(num) -> IO.inspect(num + 10) end)
    |> IO.inspect(label: "STEP 1 RESULT")
    |> Stream.map(fn(num) -> IO.inspect(num * 2) end)
    |> IO.inspect(label: "STEP 2 RESULT")
    |> Stream.map(fn(num) -> IO.inspect(to_string(num)) end)
    |> IO.inspect(label: "STEP 3 RESULT")
    |> Enum.to_list()
  end
end
Playing.peeking_into_stream
#=> ORIGINAL DATA: [1, 2, 3, 4, 5]
#=> STEP 1 RESULT: #Stream<[
#=>   enum: [1, 2, 3, 4, 5],
#=>   funs: [#Function<48.51129937/1 in Stream.map/2>]
#=> ]>
#=> STEP 2 RESULT: #Stream<[
#=>   enum: [1, 2, 3, 4, 5],
#=>   funs: [#Function<48.51129937/1 in Stream.map/2>,
#=>    #Function<48.51129937/1 in Stream.map/2>]
#=> ]>
#=> STEP 3 RESULT: #Stream<[
#=>   enum: [1, 2, 3, 4, 5],
#=>   funs: [#Function<48.51129937/1 in Stream.map/2>,
#=>    #Function<48.51129937/1 in Stream.map/2>,
#=>    #Function<48.51129937/1 in Stream.map/2>]
#=> ]>
#=> 11
#=> 22
#=> "22"
#=> 12
#=> 24
#=> "24"
#=> 13
#=> 26
#=> "26"
#=> 14
#=> 28
#=> "28"
#=> 15
#=> 30
#=> "30"
#=> ["22", "24", "26", "28", "30"]
```

By adding a call to an `Enum` function, it forced our stream to evaluate. Notice that it performed *all* of the functions in the sequence to the first element in the list *before* moving to the second element in the list.

Here’s a different way to visualize what just happened.



```
[   1,    2,    3,    4,    5]
fn(num) -> num + 10 end
fn(num) -> num * 2 end
fn(num) -> to_string(num) end
["22", "24", "26", "28", "30"]
```

Each element in the list is piped through the sequence of functions we defined in our stream and the final value is used in the resulting list. Using this approach **we never create the intermediate lists**. Only 1 new list containing the final values is created.

When working with very large data sets, this can make a *big* difference to the memory consumption of your application!

## When to use Enum vs Stream?

An obvious question to ask is "when do I choose one approach over the other?" Unfortunately, the *answer* isn't always obvious. There are some clear occasions where `Stream` is the best option. Other times `Enum` performs better. There are plenty of scenarios where they are so similar that it doesn't matter. There isn't an absolute rule to follow here. As you play with it, you get a feel for it.

These next exercises aren't problems to "solve". They are opportunities to "play". I setup some scenarios that give you a chance to play in IEx and develop your own feel for how these approaches compare. I'll suggest some things to try and experiments to run. Feel free to tweak and run your own experiments as well!

Play time!

## The playground equipment

First, let's make sure you are comfortable with the playground equipment we will use here.

### Location of the playground

Our playground is located in the `lib/streams.ex` file. Open it in your editor and look around.

### Using IEx on the project

In a terminal window in the directory of the downloaded project file, enter the following:

```
iex -S mix
```

This starts an IEx session and loads the mix project into it so all the code is available to play with.

Executing a function and giving it some initial data looks like this:

```
CodeFlow.Streams.experiment_1_enum([1, 2, 3, 4, 5])

CodeFlow.Streams.experiment_1_enum(1..1_000)
```

## Tweaking the code

As you are playing and experimenting, you will likely have an “I wonder…” moment and want to tweak the code. Feel free! That’s what this is for!

Rather than killing and restarting the IEx session, you can trigger Elixir to pick up your code changes and recompile it while your IEx session is still running. Use the `recompile` function.

```
recompile
```

Try it out. When you run `recompile` and no code changes were made, it returns a `:noop`. Meaning “**noop**eration” was performed. If you made a code change, it returns `:ok`. Using `recompile` keeps you in the flow and experimenting.

## Measuring the experiments

Let’s take a look at a simple experiment to see what it’s made up of.

```
  def experiment_1_enum(data) do
    simple_measurements(fn ->
      data
      |> Enum.map(&(&1 * 2))
      |> Enum.map(&(&1 + 1))
      |> Enum.map(&(&1 + 2))
      |> Enum.map(&(&1 + 3))
      |> Enum.map(&(&1 + 4))
      |> Enum.map(&(&1 + 5))
      |> Enum.map(&(&1 + 6))
      |> Enum.map(&(&1 + 7))
      |> Enum.map(&(&1 + 8))
      |> Enum.map(&(&1 + 9))
      |> Enum.map(&(&1 - 10))
      |> Enum.to_list()
    end)
  end
```

This defines a series of `Enum.map/2` functions. Why so many? Each call to `Enum.map/2` creates an intermediate list with the results for that step. By having so many it helps exaggerate the differences. Tweaking that can be part of your experiments!

Note the experimental code is wrapped inside the `simple_measurements` function. An anonymous function passes in the experiment to run. The `simple_measurements` function does the following things for us:

- Forces a system-wide garbage collection to give us a standard baseline
- Prints out the amount of memory our process is consuming *before* running the experiment
- Tracks the time at start
- Runs the anonymous function that performs our experiment
- Tracks the time at stop
- Prints out the amount of memory our process is consuming *after* running the experiment
- Prints out the elapsed time in milliseconds

Here’s an example of the output.

```
CodeFlow.Streams.experiment_1_enum([1, 2, 3])
#=> 0.01 MB
#=> 0.01 MB
#=> 0 msec
#=> :ok
```

Executing the function runs the experiment. I passed in a simple 3 element list of `[1, 2, 3]`. It printed out the starting RAM and ending RAM followed by the elapsed time. With such a small list, it doesn't really even register. The experiments are designed to make it easy for you to play with different sizes of data and see the impact.

Also note that these are **not** proper scientific benchmarks. Run the same operation multiple times and you will see variations between the runs. The goal with this setup is to give you enough feedback from your experiments that you get a feel for how it behaves.

Now that you have been introduced to the playground, it's time to try something!

## Experiment #1

There are two functions for experiment #1. An `Enum` and a `Stream` version. Try running them both and compare.

```
CodeFlow.Streams.experiment_1_enum(1..1_000)
#=> 0.01 MB
#=> 0.14 MB
#=> 1 msec
#=> :ok

CodeFlow.Streams.experiment_1_stream(1..1_000)
#=> 0.01 MB
#=> 0.08 MB
#=> 1 msec
#=> :ok
```

With a 1,000 item list, they both perform very quickly. The `Enum` version creates intermediate lists with all those steps. The RAM difference is noticeable but minor.

What do you think? Does it matter at this point which approach you'd choose?

Now try it with a much larger lists. Try these out:

- 1..1_000_000
- 1..10_000_000

What differences did you observe? Notice that the elapsed time is about the same. What about the difference in RAM?

## Experiment #2

In the previous experiment, our function returns a full list of however many items you said should be in the list. What if the work we do doesn't return a list but a computed result? What does that do?

In experiment #2 we change the last line from `Enum.to_list/1` to `Enum.sum/1`. Instead of returning the full list it sums all the numbers and returns the summed value.

Working with a list of 1,000 elements might look like this:

```
CodeFlow.Streams.experiment_2_enum(1..1_000)
#=> 0.01 MB
#=> 0.05 MB
#=> 1 msec
#=> :ok

CodeFlow.Streams.experiment_2_stream(1..1_000)
#=> 0.01 MB
#=> 0.04 MB
#=> 1 msec
#=> :ok
```

Now try it again with very large lists. Some suggested sizes again.

- 1..1_000_000
- 1..10_000_000

How did returning a computed value differ from returning a list?

Try using a very small list `[1, 2]` and using `IO.inspect/2` *inside* the `Stream.map/2` calls to peer into what happens.

Reveal how to tweak the experiment if you'd like a shortcut.

**Showing experiment with IO.inspect**

```
  def experiment_2_stream(data) do
    simple_measurements(fn ->
      data
      |> Stream.map(&IO.inspect(&1 * 2))
      |> Stream.map(&IO.inspect(&1 + 1))
      |> Stream.map(&IO.inspect(&1 + 2))
      |> Stream.map(&IO.inspect(&1 + 3))
      |> Stream.map(&IO.inspect(&1 + 4))
      |> Stream.map(&IO.inspect(&1 + 5))
      |> Stream.map(&IO.inspect(&1 + 6))
      |> Stream.map(&IO.inspect(&1 + 7))
      |> Stream.map(&IO.inspect(&1 + 8))
      |> Stream.map(&IO.inspect(&1 + 9))
      |> Stream.map(&IO.inspect(&1 - 10))
      |> Enum.sum()
    end)
  end
```

You should see that the `Stream` version **never builds a list of the values at all!** That's how the memory difference can be so large when working with large lists.

## Experiment #3

Here's another function to play with. `Enum.take/2` takes a desired number of elements from an enumerable and returns them in a list.

What happens when I process a very large list but only want the **first 5** elements in the result? That's the idea we play with in this experiment.

Try it for yourself!

```
CodeFlow.Streams.experiment_3_enum(1..10_000_000)

CodeFlow.Streams.experiment_3_stream(1..10_000_000)
```

Did you see a difference? Do you know why they behaved differently? Think about that for a minute before showing more explanation about it.

**Showing more explanation**
`Enum` functions are *eager*. At every step they build a full list of the results. The last step is "take the first 5". All the work creating the millions of list entries was completely wasted.

`Stream` functions are *lazy*. They only compute *what* is requested *when* it is requested. Nothing in the stream is evaluated until the last line when the first 5 are requested. So it only ever computes the first 5.

To see this in action, try this out in IEx.

```
defmodule Testing do
  def take_5() do
    1..10_000_000
    |> Stream.map(&IO.inspect(&1 * 2, label: "Num #{&1} * 2"))
    |> Enum.take(5)
  end
end
Testing.take_5()
```

It *only* performs the operation on the first 5 elements.

A `Range` is a stream too. It doesn't expand to the full set of numbers when expressed like `1..100` .

This also shows that a `Stream` can be stopped before visiting all the possible elements.

## Experiment #4

The downloaded project includes a file in `test/support/lorem.txt` that contains 13MB of generated lorem-ipsum text.

The `lorem.txt` file gives us a chance to play with reading a large-ish file. Using the `Enum` approach, the entire file gets loaded into memory and operated on. Using the `Stream` approach we can read chunks of data or even just a line of text at a time.

This experiment takes each line of text and splits it into a list of words. It then gets a count of the number of words for each line. Finally, it sums the number of words on each line all together for a total number of words for the file.

Run both and see how they compare. Run it a few times until the times level out.

```
CodeFlow.Streams.experiment_4_enum()

CodeFlow.Streams.experiment_4_stream()
```

What did you see happen? Why do you think it came out that way?

**Showing more explanation**
Your times will vary! This is how my computer performed each.

```
CodeFlow.Streams.experiment_4_enum()
#=> 0.01 MB
#=> Total words counted: 1954512
#=> 13.08 MB
#=> 108 msec
#=> :ok

CodeFlow.Streams.experiment_4_stream()
#=> 0.01 MB
#=> Total words counted: 1954512
#=> 0.03 MB
#=> 219 msec
#=> :ok
```

Notice that the `Stream` version was 2x slower. Also notice that the `Enum` version used RAM is much larger. The exact size varies with the version of Erlang and your platform. It loads the full file into memory and operates on it there. The `Stream` version used *very little* memory.

Why is the `Stream` version slower? Disk access is slower than memory access. The time it takes to read from the disk (even when the OS caches it) is slower than RAM.

Which is better? It depends.

Which is better when you need to process 100 files all 5K in size? What if the files were all 250GB in size? Do you care more about processing multiple files in **parallel** or the raw speed on a **single** file at a time?

Which is *better* depends on the problem you are solving and how it will be used.

## Built-in ways to start a stream

After playing with the experiments, hopefully you have a better sense of when to use `Enum` vs `Stream`. The next question you might ask is “What can be a stream”? Elixir’s standard library comes with some built-in functions for creating streams.

Here are few built-in ways to create a stream without using the `Stream` module.

- `File.stream!/3` – Returns a stream for reading a file. Used in Experiment #4.
- `IO.stream/2` – Converts an IO device into a stream.
- `Task.async_stream/5` – Spawns a concurrent Task to process elements in the stream. Options let you tune how many concurrent tasks will run, etc. A good option when the task being performed is expensive and running in parallel makes sense.
- `Ecto.Repo.stream/2` – The Ecto database library can stream query results. Works well for processing potentially very large result sets.

## Anything can be a stream

Using `Stream.resource/3`, potentially *anything* can become a stream. The `resource/3` function takes in 3 functions to do the following:

- Setup the resource
- Get the next value from it
- Close or cleanup the resource

Examples of what you could use this to do:

- Fetch and process pages of JSON data from an external service. The “next” function can fetch the next page of data.
- Lazily parse a large CSV file – which is what the NimbleCSV library does.
- Stream and process very large files over the web – as this blog series demonstrates.

## Recap

You took a different approach here. Instead of making failing tests pass, you spent time playing and experimenting with `Enum` and `Stream` to get a feel for how a `Stream` is different.

In many ways, a `Stream` acts like `Enum`. Both are enumerable, have functions for processing data, and more. However a `Stream` is different because it uses *lazy evaluation*.

You may not need to use a `Stream` often, but knowing *what* it is, *how* to use it, *why* it is different, and getting a *feel* for the kinds of problems it helps solve was our goal.

Consider using a `Stream` when you are working with data that would otherwise consume a lot of memory to process. Examples are:

- large lists
- large files
- large database result sets
- processing a potentially unbounded source

# Handling Errors

Elixir has 3 varieties of errors.

- Exceptions
- Throws
- Exits

Each type serves a different purpose. Before we dig in, it is worth mentioning that these are used far less frequently in Elixir than in other languages. Expect that your usage of these will also be “exceptional”, meaning infrequent and only when required.

## Exceptions

Exceptions are not intended for managing Code Flow or control flow. They are intended for *actual errors*. Here’s an example of a `ArithmeticError` raised exception.

```
:customer + 10
#=> ** (ArithmeticError) bad argument in arithmetic expression: :customer + 10
#=>     :erlang.+(:customer, 10)
```

## Raising an exception

You can `raise` a `RuntimeError` using the `raise` keyword.

```
raise "something failed!"
#=> ** (RuntimeError) something failed!
```

You can also raise a specific error type using raise/2 . You provide the exception name/type and a message. Other attributes may be defined on an exception and using a keyword list you can set those as well.

```
raise ArgumentError, "you must provide a valid arg"
#=> ** (ArgumentError) you must provide a valid arg

raise ArgumentError, message: "you must provide a valid arg"
#=> ** (ArgumentError) you must provide a valid arg
```

## Defining custom exceptions

You can create custom exceptions as well. They are defined within a module this way:

```
defmodule MySpecialError do
  defexception message: "Something special blew up", extra: nil
end

# raising with default message
raise MySpecialError
#=> ** (MySpecialError) Something special blew up

# overriding the message
raise MySpecialError, "what just happened?"
#=> ** (MySpecialError) what just happened?

# setting :extra value requires keyword usage
raise MySpecialError, message: "custom message", extra: 123
#=> ** (MySpecialError) custom message
```

The `defexception` works like a `defstruct` where you provide the keys that your exception will have and optionally any default values.

The `:extra` value is set in the `MySpecialError` but a simplified error representation is written to the console. In order to access that data or handle the error, we need to use `try/rescue` .

## Handling with try/rescue

The `try/rescue` block is used to handle raised exceptions.

```
try do
  raise "Boom!"
rescue
  e in RuntimeError ->
    "Raised error: #{e.message}"
end
#=> "Raised error: Boom!"
```

If we wanted to handle the custom exception we defined before and access the data on the `:extra` field, we can do the following.

```
defmodule MySpecialError do
  defexception message: "Something special blew up", extra: nil
end

try do
  raise MySpecialError, message: "custom message", extra: 123
rescue
  e in MySpecialError ->
    IO.puts("Found MySpecialError.extra value #{e.extra}")
    {:error, "#{e.message}: #{e.extra}"}
end
#=> Found MySpecialError.extra value 123
#=> {:error, "custom message: 123"}
```

This example handles the raised custom error. The `rescue` block doesn't support full pattern matching on the exception. It expects the expression to be a variable, a module, or a `var in Module` format. This still lets us get basic matching enough to access the `:extra` information on the error and transform it to an `{:error, reason}` tuple.

If you don't care about the error type or even referencing the error, you can do this:

```
try do
  raise "Boom!"
rescue
  _ ->
    "Hit an error"
end
#=> "Hit an error"
```

The above matches on any exception type.

Specifying an exception's module name, you can match on specific error type if needed.

```
try do
  raise "Boom!"
rescue
  RuntimeError ->
    "A runtime error was hit"

  _ ->
    "Hit an error"
end
#=> "A runtime error was hit"
```

## Writing functions that raise exceptions

Sometimes you may want to create a function that raises an exception when something is really wrong. If your code explicitly raises exceptions, then the convention is to name it ending with a `!`. This is a **naming convention** used to convey the function may raise an exception. It typically means, "Perform the operation or raise an exception." This is also called a "bang". An example is the `Enum.fetch!/2` function. You would read this as the "Enum fetch bang" function. From the documentation, it is described as:

> Finds the element at the given `index` (zero-based). Raises `OutOfBoundsError` if the given `index` is outside the range of the `enumerable`.
>
> https://hexdocs.pm/elixir/Enum.html#fetch!/2

Many of the functions in the `File` module have `!` (bang) versions as a failure to read a file may be considered truly exceptional and a reason to fail entirely.

As mentioned previously, in Elixir it is far less common to deal with exceptions than what you find in other languages. For this reason, the `File` module also contains `{:ok, result}` and error responses for all the same functions. It is easier to pattern match and more common to respond to an `:ok/:error` tuple response than raised exceptions.

## Alternate Syntax

An alternate syntax exists for the `try/rescue` block. Within a function, you can just use the `rescue` at the end of the function block without including the `try`.

```
defmodule Testing do
  def full_try do
    try
      # operation that might raise exception
    rescue
      # handling logic
    end
  end

  def without_try do
    # operation that might raise exception
  rescue
    # handling logic
  end
end
```

The above are equivalent. When your `try/rescue` block would encompass the **full function body**, you can use the abbreviated version. It is cleaner and has less indentation noise.

## Exercise #1 – Conditionally raise an exception

The project has an existing function `CodeFlow.Fake.Users.find/1`. It finds a user and returns an `{:ok, user}` or `{:error, reason}` result. Your team wants you to create a `find_user!/1` function that calls the existing `Users.find/1` function but *unwraps* the tuples. When found, return the user not in a tuple. When *not* found, raise an exception where the message is the reason from the `:error` tuple.

The tests for this exercise are here:

```
mix test test/errors_test.exs:18
```

**Showing Solution**

```
  def find_user!(id) do
    case Users.find(id) do
      {:ok, user} ->
        user

      {:error, reason} ->
        raise reason
    end
  end
```

## Exercise #2 – Transforming an exception

As your team has gotten more comfortable with Elixir and has started “thinking Elixir”, they realize they have too many functions raising exceptions. They brought habits from other languages with them when they were new to Elixir. Now they want to add more functions that return tuples but not break all the existing code using the exception versions.

The project has a function named `CodeFlow.Fake.Orders.find!/1` that raises an exception when an order is not found. Your team wants you to create a `CodeFlow.Errors.find_order/1` function that uses the existing `CodeFlow.Fake.Orders.find!/1` function to do the work but transforms a raised exception into a tuple response. Capture the exception’s message and use that as the reason for the failure. When an order is found, wrap it in an `{:ok, order}` tuple.

The tests for this exercise are here:

```
mix test test/errors_test.exs:35
```

**Showing Solution**

This uses the `rescue` block built into the `def` function. It is equally valid to use the full `try/rescue` block version.

```
  def find_order(id) do
    {:ok, Orders.find!(id)}
  rescue
    e in RuntimeError ->
      {:error, e.message}
  end
```

## Throw and Catch

The `throw` and `catch` keywords *are* designed for Code Flow or control flow. It is reserved for situations where it is the *only way* you can get a value back. Again, like `raise` and `rescue`, `throw` and `catch` should be used sparingly.

The times where you might need to use `throw` and `catch` are when working with poorly designed libraries. So, truly, you don't use these much. However, here's what it looks like if you did need it.

This example stops at the first number evenly divisible by 17 but is not 17. It throws the matching value which is caught outside the loop.

```
try do
  Enum.each 1..1_000_000, fn(n) ->
    if rem(n, 17) == 0 && n != 17 do
      throw n
    end
  end
catch
  found -> "Caught #{found}"
end
```

I've never encountered an actual need for the `throw` keyword. You aren't likely to either.

## Exit

Elixir has a great concurrency story and that all happens because of green thread processes. We aren't going to get into those at this point. However, this is relevant to an `exit`. An `exit` is a signal sent to the process running the code telling it to die.

```
exit("things went BOOM!")
#=> ** (exit) "things went BOOM!"
```

An `exit` signal is like a special thrown value which can be handled using `catch`.

```
try do
  exit "Abort! Abort!"
catch
  :exit, reason ->
    IO.inspect reason
    "Phew... caught that."
end
#=> "Abort! Abort!"
#=> "Phew... caught that."
```

**Thinking Tip: Don't actually catch exits**

Exits deal with killing a process. In Elixir, processes are typically supervised. The `exit` is caught and handled by the supervisor who's job is to restart a new process in a known good state to keep the system running smooth. Exit signals are an important part of a resilient system.

If using `throw` and `catch` are uncommon, then catching an `exit` is even *less* common! There may be cases where you want to trigger an `exit`, but you'll be wanting a supervisor to handle it.

## After

The `after` keyword is used to help ensure some cleanup operations happen either if the `try` block succeeds or blows up.

```
try do
  IO.puts "try block succeeded"
after
  IO.puts "after performed"
end
#=> try block succeeded
#=> after performed

try do
  raise "try block failed"
after
  IO.puts "after performed"
end
#=> after performed
#=> ** (RuntimeError) try block failed
```

An important point to realize about the `after` block is that it only offers a *soft* guarantee. The documentation on this is helpful to review.

> Note that the process will exit as usual when receiving an exit signal that causes it to exit abruptly and so the `after` clause is not guaranteed to be executed. Luckily, most resources in Elixir (such as open files, ETS tables, ports, sockets, and so on) are linked to or monitor the owning process and will automatically clean themselves up if that process exits.
>
> https://hexdocs.pm/elixir/Kernel.SpecialForms.html?#try/1-after-clauses

We learn that when a process accessing a resource dies, the resources linked to it are automatically closed. Typically in other languages you might use an `after` to close open files or resources. However, in Elixir you probably don't need that case either. This means there aren't likely to be many cases where you actually *need* an `after`.

## Else

The `else` block is optional. If present, it provides the ability to pattern match on the result of the `try` block when it **does not fail**.

```
try do
  :success
rescue
  _ ->
    :error
else
  :success ->
    "Yay, it worked!"
end
#=> "Yay, it worked!"
```

Note: any errors or failures in the `else` block code are *not* caught.

## Variables and Scope

An important point to make with the `try/rescue/catch/after` blocks is that any variables bound inside those block do not leak out. For instance, when the code in the try block below fails, would you expect some variables to be bound but others not?

```
try do
  raise "boom!"
  status = "no error"
rescue
  _ ->
    status = "rescued"
end
status
#=> ** (CompileError) iex:9: undefined function status/0
```

Instead, bind the result of the `try/rescue` block to the value.

```
status =
  try do
    raise "boom!"
    "no error"
  rescue
    _ ->
      "rescued"
  end
status
#=> "rescued"
```

## Recap

Putting all of these pieces together looks like this:

```
try do
  # code that might raise, throw or exit
rescue
  # handle exceptions
catch
  # catch any "thrown" values
  # catch any "exit" signals
after
  # code to always run for cleanup
else
  # code that runs when nothing errors in the try block
end
```

Within a function body, using a `rescue/catch/after` will cause the compiler to include a `try` block for you. This helps the syntax stay a bit cleaner. It looks like this:

```
def do_work() do
  # code that might raise, throw or exit
rescue
  # handle exceptions
catch
  # catch any "thrown" values
  # catch any "exit" signals
after
  # code to always run for cleanup
end
```

Some key points to remember:

- All of these error handling options are not encountered often. They are available but seldom used.
- The most common one you are likely to use use is `raise` and `rescue`. Other libraries may raise exceptions that you must deal with.
- Pattern matching is preferred to exceptions.
- Functions that *do* raise an exception end with a `!` bang as a naming convention.

# If and Cond statements

The last language features being discussed for managing Code Flow are the `if` and `cond` statements. The reason the `if` is introduced at this late stage is to emphasize how less important it is in Elixir. The `if` statement works like you might expect from other languages. However, an `if` statement **does not support pattern matching**. By now you've seen that pattern matching is everywhere in Elixir! This similarity to other languages is a weakness for the `if` statement.

## `if` statements

There are valid uses for `if` conditions like needing to test a single value. Typically, if you find yourself reaching for an `if` by default, realize there is probably a better way to write the code. In general, consider the writing of an `if` statement as a possible anti-pattern. With that said, let's look at the syntax and talk about how it works.

```
if condition do
  # executed when condition is truthy
else
  # executed when condition is falsy
end
```

A **falsy** value is `nil` or `false` and a **truthy** value is everything else.

If you want to test that a value *is* `false`, you would write it as `if value == false do...` A better way is to use pattern matching.

## Variables and leaking

By now you've seen that variables bound inside a clause don't leak out. This is also true for the `if`. It is a common pattern in other languages to assign variables inside an `if` statement. Beneficially, the compiler recognizes this common mistake and points you in the right direction.

```
# This doesn't work!
do_thing = false

if some_condition do
  do_thing = true
end

#=> warning: variable "do_thing" is unused
#=>
#=> Note variables defined inside case, cond, fn, if and similar do not leak. If you want to conditionall
y override an existing variable "do_thing", you will have to explicitly return the variable. For example:
#=>
#=>     if some_condition? do
#=>       atom = :one
#=>     else
#=>       atom = :two
#=>     end
#=>
#=> should be written as
#=>
#=>     atom =
#=>       if some_condition? do
#=>         :one
#=>       else
#=>         :two
#=>       end
```

That's a pretty helpful error message! This shows that the `if` statement returns a value and that's how you get conditional values out of it.

Note that you aren't able to bind multiple results like this. That is an extra signal that an `if` isn't the preferred way to manage Code Flow.

## Ternary statements

In languages like Javascript, there is a ternary operator. This doesn't exist in Elixir. However, the `if` statement supports being expressed as a Keyword list putting it inline. Note that it doesn't use the `end`.

```
result = if true, do: "true", else: "false"
result
#=> "true"
```

## Nested `if` statements

When you see a nested `if` statement, that is *definitely* a code smell and an anti-pattern in Elixir. This is an opportunity to refactor into one or more of the following alternative structures:

- multiple function clauses with pattern matching
- a `case` statement
- possibly a `cond` statement (up next)

## `unless` statement

If usage of an `if` clause is a code smell of a possible anti-pattern then usage of the `unless` statement is even more so.

```
unless condition do
  # executed when condition is falsy
end
```

The `unless` clause exists, but is negative logic. This is harder to reason about and is best avoided.

## No “else if” conditions

Note that the `if` statement does not have an `elsif` or built-in `else if` style clause. If you need that, then the `cond` statement is a better fit.

## `cond` statements

The `cond` statement is well suited for replacing the `if ... else if` clauses you find in many other languages. However, because Elixir provides many more powerful features for controlling Code Flow, `cond` is used much less frequently.

It evaluates a series of conditions and stops at the **first truthy** one.

```
value = 123

cond do
  value > 200 ->
    "Greater than 200"
  value > 100 ->
    "Greater than 100"
  value > 50 ->
    "Greater than 100"
end
#=> "Greater than 100"
```

If *nothing* matches, it raises an exception.

```
value = 123

cond do
  value in 1..10 ->
    "Between 1 an 10"
  value > 200 ->
    "Greater than 200"
  value < 100 ->
    "Less than 100"
end
#=> ** (CondClauseError) no cond clause evaluated to a truthy value
```

To ensure *something* matches like the last `else` clause of an `if ... else if` statement, use `true` as your condition.

```
value = 123

cond do
  value in 1..10 ->
    "Between 1 an 10"
  value > 200 ->
    "Greater than 200"
  value < 100 ->
    "Less than 100"
  true ->
    "It was something else..."
end
#=> "It was something else..."
```

The final `true` provides a “default” or grand `else` clause to ensure you have a match.

## Recap

- “falsy” is when a condition is `nil` or `false`
- “truthy” is anything not falsy
- `if` statements do not support pattern matching
- Nested `if` statements are an anti-pattern
- You can do simple inline statements: `if 1 == 1, do: "One!", else: "...uh..."`

- Don't use `unless` statements. They are negative `if` statements.
- `cond` handles the `if ... else if` statements you may be familiar with
- `cond` statements can end with `true` to ensure a match

# Code Flow Summary

Congratulations! We covered a lot of ground together! If you took the time to do the exercises then you've built up excellent hands-on experience applying these new concepts.

Let's briefly review what we've covered here and stress some take-away points.

## What we covered

- The Pipe Operator `|>` does a simple job but it enables pipelines which are more elegant and readable.
- A `case` statement supports guard clauses and they can be piped into.
- A Keyword list is a list of tuples. It is frequently used for passing options to a function.
- The Railway Pattern adds pattern matching clauses and pipelines to create a "happy path" and "failure path". Works well on functions defined in a module used as a workflow.
- The "with" statement uses pattern matching to define a "happy path". It replaces nested `case` statements and works well for pulling functions together from different parts of the system.
- Looping in Elixir is done using pattern matching and recursive function calls.
- Tail recursion is a feature of many functional languages that makes recursive calls able to "loop" forever without a stack overflow.
- The `Enum` module has functions that help with processing a list. They include `Enum.each/2`, `Enum.map/2`, and `Enum.reduce/3`.
- `for` comprehensions provide a simple way to loop. It is a Swiss Army Knife feature that can do many things. A great tool to have on-hand.
- Error handling is used less in Elixir than other languages. A `raise` needs a `rescue`. A `throw` needs a `catch` and an `exit` is a special type of `throw`.
- The `if` statement exists but is used far less frequently.
- A nested `if` is an anti-pattern.
- The `cond` statement replaces the `if ... else if` structures from other languages.

## Extra things we picked up on the way

- `IO.inspect/2` is a great debugging and insight tool. It is "pipe friendly" and can be added liberally. Using the `label: "My Label"` option really helps to identify which inspect call we are looking at.
- When defining an `alias`, we can use the `alias MyApp.Thing, as: OtherName` option to override what it is called.
- We talked about the Range type (ex: `1..10`) and how they can be used for looping.
- Anonymous functions support multiple clauses. This gives added flexibility when working with `Enum` list processing functions.
- Elixir has "truthy" and "falsy" evaluations. A "falsy" value is `nil` or `false`. A "truthy" values is anything that is *not* `nil` or `false`.

## Lessons learned

Elixir has immutable data, is a "functional language", uses recursion, doesn't have "objects". This can feel very unfamiliar and frankly a little scary. It is exactly these characteristics that are the foundation for the powerful concurrency model and the resilience of Elixir systems. This requires you to learn to do old things in a new way. Even the way you control the flow of your code changes.

Having completed this course, you have experienced for yourself that "you can do this".

Feeling *comfortable* with Elixir takes time. You know the techniques, you've used the control patterns, and you know when to use them.

Now it is up to you to continue with it. Look for opportunities to use these Code Flow patterns. Don't be afraid to refactor your code from one pattern into another as you realize something else is a better fit. This is where you really build experience and skill.

You got here on your own. I know you can do this. We did cover a lot and you can return to review anything you need. I think the most important thing you have learned from this is that ***you know you can do this***.

You got this.

## Download reference resource

Now that you have completed the course, as a special “thank you”, I want you to have a ready, portable, handy reference as a resource of everything we covered together. This is a PDF download of the course information. It is indexed and searchable so you can easily jump around and find something you want to refer back to.