Control Flow
The ability to run some code depending on whether a condition is true and to run some code repeatedly while a condition is true are basic building blocks in most programming languages. The most common constructs that let you control the flow of execution of Cairo code are if expressions and loops.
`if` Expressions
An if expression allows you to branch your code depending on conditions. You provide a condition and then state, “If this condition is met, run this block of code. If the condition is not met, do not run this block of code.”
Create a new project called _branches_ in your _cairo_projects_ directory to explore the `if` expression. In the _src/lib.cairo_ file, input the following:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_27_if/src/lib.cairo}}
```
All `if` expressions start with the keyword `if`, followed by a condition. In this case, the condition checks whether or not the variable `number` has a value equal to 5. We place the block of code to execute if the condition is `true` immediately after the condition inside curly brackets.
Optionally, we can also include an `else` expression, which we chose to do here, to give the program an alternative block of code to execute should the condition evaluate to `false`. If you don’t provide an `else` expression and the condition is `false`, the program will just skip the `if` block and move on to the next bit of code.
Try running this code; you should see the following output:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_27_if/output.txt}}
```
Let’s try changing the value of `number` to a value that makes the condition `true` to see what happens:
```cairo, noplayground
    let number = 5;
```
```shell
$ scarb cairo-run
condition was true and number = 5
Run completed successfully, returning []
```
It’s also worth noting that the condition in this code must be a `bool`. If the condition isn’t a `bool`, we’ll get an error. For example, try running the following code:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_28_bis_if_not_bool/src/lib.cairo}}
```
The `if` condition evaluates to a value of 3 this time, and Cairo throws an error:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_28_bis_if_not_bool/output.txt}}
```

~~~
The error indicates that Cairo inferred the type of `number` to be a `bool` based on its later use as a condition of the `if` statement. It tries to create a `bool` from the value `3`, but Cairo doesn't support instantiating a `bool` from a numeric literal anyway - you can only use `true` or `false` to create a `bool`. Unlike languages such as Ruby and JavaScript, Cairo will not automatically try to convert non-Boolean types to a Boolean. If we want the `if` code block to run only when a number is not equal to 0, for example, we can change the if expression to the following:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_29_ter_if_not_equal_zero/src/lib.cairo}}
~~~
Running this code will print `number was something other than zero`.
Handling Multiple Conditions with `else if`
You can use multiple conditions by combining `if` and `else` in an `else if` expression. For example:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_30_else_if/src/
lib.cairo}}
```
This program has four possible paths it can take. After running it, you should see the following output:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_30_else_if/
output.txt}}
```
When this program executes, it checks each `if` expression in turn and executes the first body for which the condition evaluates to `true`. Note that even though `number - 2 == 1` is `true`, we don’t see the output `number minus 2 is 1` nor do we see the `number not found` text from the `else` block. That’s because Cairo only executes the block for the first true condition, and once it finds one, it doesn’t even check the rest. Using too many `else if` expressions can clutter your code, so if you have more than one, you might want to refactor your code. [Chapter 6][match] describes a powerful Cairo branching construct called `match` for these cases.
[match]: ./ch06-02-the-match-control-flow-construct.md
Using `if` in a `let` Statement
Because `if` is an expression, we can use it on the right side of a `let` statement to assign the outcome to a variable.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_31_if_let/src/
lib.cairo}}
```
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_31_if_let/
output.txt}}
```

~~~
The `number` variable will be bound to a value based on the outcome of the `if` expression, which will be 5 here.
## Repetition with Loops
It’s often useful to execute a block of code more than once. For this task, Cairo provides a simple loop syntax, which will run through the code inside the loop body to the end and then start immediately back at the beginning. To experiment with loops, let’s create a new project called _loops_.
Cairo has three kinds of loops: `loop`, `while`, and `for`. Let’s try each one.
### Repeating Code with `loop`
The `loop` keyword tells Cairo to execute a block of code over and over again forever or until you explicitly tell it to stop.
As an example, change the _src/lib.cairo_ file in your _loops_ directory to look like this:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_32_infinite_loop/src/lib.cairo}}
~~~
When we run this program, we’ll see `again!` printed over and over continuously until either the program runs out of gas or we stop the program manually. Most terminals support the keyboard shortcut ctrl-c to interrupt a program that is stuck in a continual loop. Give it a try:
```shell
$ scarb cairo-run --available-gas=20000000
   Compiling loops v0.1.0 (file:///projects/loops)
    Finished release target(s) in 0 seconds
     Running loops
again!
again!
again!
^Cagain!
```
The symbol `^C` represents where you pressed ctrl-c. You may or may not see the word `again!` printed after the ^C, depending on where the code was in the loop when it received the interrupt signal.
> Note: Cairo prevents us from running program with infinite loops by including a gas meter. The gas meter is a mechanism that limits the amount of computation that can be done in a program. By setting a value to the `--available-gas` flag, we can set the maximum amount of gas available to the program. Gas is a unit of measurement that expresses the computation cost of an instruction. When the gas meter runs out, the program will stop. In the previous case, we set the gas limit high enough for the program to run for quite some time.
> It is particularly important in the context of smart contracts deployed on Starknet, as it prevents from running infinite loops on the network.
> If you're writing a program that needs to run a loop, you will need to execute it with the `--available-gas` flag set to a value that is large enough to run the program.
Now, try running the same program again, but this time with the `--available-gas` flag

set to `200000` instead of `2000000000000`. You will see the program only prints
`again!` 3 times before it stops, as it ran out of gas to keep executing the loop.
Fortunately, Cairo also provides a way to break out of a loop using code. You can place
the `break` keyword within the loop to tell the program when to stop executing the loop.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_33_loop_break/
src/lib.cairo}}
```
The `continue` keyword tells the program to go to the next iteration of the loop and to
skip the rest of the code in this iteration.
Let's add a `continue` statement to our loop to skip the `println!` statement when `i` is
equal to `5`.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_34_loop_continue/
src/lib.cairo}}
```
Executing this program will not print the value of `i` when it is equal to `5`.
Returning Values from Loops
One of the uses of a `loop` is to retry an operation you know might fail, such
as checking whether an operation has succeeded. You might also need to pass
the result of that operation out of the loop to the rest of your code. To do
this, you can add the value you want returned after the `break` expression you
use to stop the loop; that value will be returned out of the loop so you can
use it, as shown here:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_35_loop_return_values/src/lib.cairo}}
```
Before the loop, we declare a variable named `counter` and initialize it to
`0`. Then we declare a variable named `result` to hold the value returned from
the loop. On every iteration of the loop, we check whether the `counter` is equal to `10`,
and then add `1` to the `counter` variable.
When the condition is met, we use the `break` keyword with the value `counter * 2`.
After the loop, we use a
semicolon to end the statement that assigns the value to `result`. Finally, we
print the value in `result`, which in this case is `20`.
Conditional Loops with `while`
A program will often need to evaluate a condition within a loop.
While the condition is `true`, the loop runs.
When the condition ceases to be `true`, the program calls `break`, stopping the loop.
It's possible to implement behavior like this using a combination of `loop`, `if`, `else`,
and `break`; you could try that now in a program, if you'd like.
However, this pattern is so common that Cairo has a built-in language construct for it,
called a `while` loop.
In Listing {{#ref while-true}}, we use `while` to loop the program three times, counting
down each time after printing the value of `number`, and then, after the loop, print a

message and exit.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_36_while_loop/
src/lib.cairo}}
```
{{#label while-true}}
<span class="caption">Listing {{#ref while-true}}: Using a `while` loop to run code while a condition holds `true`.</span>
This construct eliminates a lot of nesting that would be necessary if you used `loop`, `if`, `else`, and `break`, and it’s clearer.
While a condition evaluates to `true`, the code runs; otherwise, it exits the loop.
Looping Through a Collection with `for`
You can also use the while construct to loop over the elements of a collection, such as an array. For example, the loop in Listing {{#ref iter-while}} prints each element in the array `a`.
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_45_iter_loop_while/src/lib.cairo}}
```
{{#label iter-while}}
<span class="caption">Listing {{#ref iter-while}}: Looping through each element of a collection using a `while` loop</span>
Here, the code counts up through the elements in the array. It starts at index `0`, and then loops until it reaches the final index in the array (that is, when `index < 5` is no longer `true`). Running this code will print every element in the array:
```shell
{{#include ../listings/ch02-common-programming-concepts/
no_listing_45_iter_loop_while/output.txt}}
```
All five array values appear in the terminal, as expected. Even though `index` will reach a value of `5` at some point, the loop stops executing before trying to fetch a sixth value from the array.
However, this approach is error prone; we could cause the program to panic if the index value or test condition is incorrect. For example, if you changed the definition of the `a` array to have four elements but forgot to update the condition to `while index < 4`, the code would panic. It’s also slow, because the compiler adds runtime code to perform the conditional check of whether the index is within the bounds of the array on every iteration through the loop.
As a more concise alternative, you can use a `for` loop and execute some code for each item in a collection. A `for` loop looks like the code in Listing {{#ref iter-for}}.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_46_iter_loop_for/
src/lib.cairo}}
```
{{#label iter-for}}
<span class="caption">Listing {{#ref iter-for}}: Looping through each element of a

collection using a `for` loop</span>

When we run this code, we’ll see the same output as in Listing {{#ref iter-while}}. More importantly, we’ve now increased the safety of the code and eliminated the chance of bugs that might result from going beyond the end of the array or not going far enough and missing some items.

Using the `for` loop, you wouldn’t need to remember to change any other code if you changed the number of values in the array, as you would with the method used in Listing {{#ref iter-while}}.

The safety and conciseness of `for` loops make them the most commonly used loop construct in Cairo. Even in situations in which you want to run some code a certain number of times, as in the countdown example that used a while loop in Listing {{#ref while-true}}. Another way to run code a certain number of times would be to use a `Range`, provided by the core library, which generates all numbers in sequence starting from one number and ending before another number.

Here’s how you can use a `Range` to count from 1 to 3:

```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_47_for_range/src/lib.cairo}}
```

This code is a bit nicer, isn’t it?

Equivalence Between Loops and Recursive Functions

Loops and recursive functions are two common ways to repeat a block of code multiple times. The `loop` keyword is used to create an infinite loop that can be broken by using the `break` keyword.

```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_loop_recursion/src/examples/loop_example.cairo}}
```

Loops can be transformed into recursive functions by calling the function within itself. Here is an example of a recursive function that mimics the behavior of the `loop` example above.

```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_loop_recursion/src/examples/recursion_example.cairo}}
```

In both cases, the code block will run indefinitely until the condition `x == 2` is met, at which point the value of x will be displayed.

In Cairo, loops and recursions are not only conceptually equivalent: they are also compiled down to similar low-level representations. To understand this, we can compile both examples to Sierra, and analyze the Sierra Code generated by the Cairo compiler for both examples. Add the following in your `Scarb.toml` file:

```toml
[lib]
sierra-text = true
```

Then, run `scarb build` to compile both examples. You will find the Sierra code

generated by for both examples is extremely similar, as the loop is compiled to a recursive function in the Sierra statements.
> Note: For our example, our findings came from understanding the **statements** section in Sierra that shows the execution traces of the two programs. If you are curious to learn more about Sierra, check out [Exploring Sierra](https://medium.com/nethermind-eth/under-the-hood-of-cairo-1-0-exploring-sierra-7f32808421f5).
{{#quiz ../quizzes/ch02-05-control-flow.toml}}
Summary
You made it! This was a sizable chapter: you learned about variables, data types, functions, comments,
`if` expressions and loops! To practice with the concepts discussed in this chapter, try building programs to do the following:
- Generate the _n_-th Fibonacci number.
- Compute the factorial of a number _n_.
Now, we’ll review the common collection types in Cairo in the next chapter.