Concise Control Flow with `if let` and `while let`
`if let`
The `if let` syntax lets you combine `if` and `let` into a less verbose way to handle values that match one pattern while ignoring the rest. Consider the program in Listing {{#ref config_max}} that matches on an `Option::Some<u8>` value in the `config_max` variable but only wants to execute code if the value is `Option::Some` variant.
```cairo
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/
no_listing_14_if_let_match_one/src/lib.cairo:match}}
```
{{#label config_max}}
Listing {{#ref config_max}}: A `match` that only cares about executing
code when the value is `Option::Some`
If the value is `Option::Some`, we print out the value in the `Option::Some` variant by binding
the value to the variable `max` in the pattern. We don’t want to do anything with the `None` value. To satisfy the `match` expression, we have to add `_ => ()` after processing just one variant, which is annoying boilerplate code to add.
Instead, we could write this in a shorter way using `if let`. The following code behaves the same as the `match` in Listing {{#ref config_max}}:
```cairo
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/no_listing_15_if_let/src/
lib.cairo:here}}
```
The syntax `if let` takes a pattern and an expression separated by an equal sign. It works the same way as a `match`, where the expression is given to the `match` and the pattern is its first arm. In this case, the pattern is `Option::Some(max)`, and `max` binds to the value inside `Option::Some`. We can then use `max` in the body of the `if let` block in the same way we used `max` in the corresponding `match` arm. The code in the `if let` block isn’t run if the value doesn’t match the pattern.
Using `if let` means less typing, less indentation, and less boilerplate code. However, you lose the exhaustive checking that `match` enforces. Choosing between `match` and `if let` depends on what you’re doing in your particular situation and whether gaining conciseness is an appropriate trade-off for losing exhaustive checking.
In other words, you can think of `if let` as syntactic sugar for a `match` that runs code when the value matches one pattern and then ignores all other values.
We can include an `else` with an `if let`. The block of code that goes with `else` is the same as the block of code that would go with the `_` case in the `match` expression. Recall the `Coin` enum definition in Listing {{#ref match-pattern-bind}}, where the `Quarter` variant also held a `UsState` value. If we wanted to count all non-quarter coins we see while also announcing the state of the quarters, we could do that with a `match` expression, like this:

```cairo
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/
no_listing_16_if_let_coiner_match/src/lib.cairo:here}}
```
Or we could use an `if let` and `else` expression, like this:
```cairo
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/
no_listing_17_if_let_coiner/src/lib.cairo:here}}
```
If you have a situation in which your program has logic that is too verbose to express using `match`, remember that `if let` is in your Cairo toolbox as well.
`while let`
The `while let` syntax is similar to the `if let` syntax, but it allows you to loop over a collection of values and execute a block of code for each value that matches a specified pattern. In the case below, the pattern is `Option::Some(x)`, which matches any `Some` variant of the `Option` enum.
```cairo
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/no_listing_18_while_let/
src/lib.cairo}}
```
Using `while let` provides a more concise and idiomatic way of writing this loop compared to a traditional `while` loop with explicit pattern matching or handling of the `Option` type. However, as with `if let`, you lose the exhaustive checking that a `match` expression provides, so you need to be careful to handle any remaining cases outside the `while let` loop if necessary.