Enums
Enums, short for "enumerations," are a way to define a custom data type that consists of a fixed set of named values, called _variants_. Enums are useful for representing a collection of related values where each value is distinct and has a specific meaning.
Enum Variants and Values
Here's a simple example of an enum:
```cairo, noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_01_enum_example/src/lib.cairo:enum_example}}
```
In this example, we've defined an enum called `Direction` with four variants: `North`, `East`, `South`, and `West`. The naming convention is to use PascalCase for enum variants. Each variant represents a distinct value of the `Direction` type. In this particular example, variants don't have any associated value. One variant can be instantiated using this syntax:
```cairo, noplayground
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/no_listing_01_enum_example/src/lib.cairo:here}}
```
Now let's imagine that our variants have associated values, that store the exact degree of the direction. We can define a new `Direction` enum:
```cairo, noplayground
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/no_listing_02_enum_with_values_example/src/lib.cairo:enum_example}}
```
and instantiate it as follows:
```cairo, noplayground
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/no_listing_02_enum_with_values_example/src/lib.cairo:here}}
```
In this code, each variant is associated with a `u128` value, representing the direction in degrees. In the next example, we will see that it is also possible to associate different data types with each variant.
It's easy to write code that acts differently depending on the variant of an enum instance, in this example to run specific code according to a direction. You can learn more about it in the [Match Control Flow Construct][match] section.
[match]: ./ch06-02-the-match-control-flow-construct.md
Enums Combined with Custom Types
Enums can also be used to store more interesting custom data associated with each variant. For example:
```cairo, noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_03_enum_message/src/lib.cairo:message}}
```
In this example, the `Message` enum has three variants: `Quit`, `Echo`, and `Move`, all with different types:

- `Quit` doesn't have any associated value.
- `Echo` is a single `felt252`.
- `Move` is a tuple of two `u128` values.

You could even use a Struct or another enum you defined inside one of your enum variants.

Trait Implementations for Enums

In Cairo, you can define traits and implement them for your custom enums. This allows you to define methods and behaviors associated with the enum. Here's an example of defining a trait and implementing it for the previous `Message` enum:

```cairo, noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_03_enum_message/src/lib.cairo:trait_impl}}
```

In this example, we implemented the `Processing` trait for `Message`. Here is how it could be used to process a `Quit` message:

```cairo
{{#rustdoc_include ../listings/ch06-enums-and-pattern-matching/no_listing_03_enum_message/src/lib.cairo:main}}
```

The `Option` Enum and Its Advantages

The `Option` enum is a standard Cairo enum that represents the concept of an optional value. It has two variants: `Some: T` and `None`. `Some: T` indicates that there's a value of type `T`, while `None` represents the absence of a value.

```cairo,noplayground
enum Option<T> {
    Some: T,
    None,
}
```

The `Option` enum is helpful because it allows you to explicitly represent the possibility of a value being absent, making your code more expressive and easier to reason about. Using `Option` can also help prevent bugs caused by using uninitialized or unexpected `null` values.

To give you an example, here is a function which returns the index of the first element of an array with a given value, or `None` if the element is not present.

We are demonstrating two approaches for the above function:

- Recursive approach with `find_value_recursive`.
- Iterative approach with `find_value_iterative`.

```cairo,noplayground
{{#include ../listings/ch06-enums-and-pattern-matching/no_listing_04_enum_option/src/lib.cairo}}
```

Enums can be useful in many situations, especially when using the `match` flow construct that we just used. We will describe it in the next section.

Other enums are used very often, such as the `Result` enum, allowing to handle errors gracefully. We will explain the `Result` enum in detail in the ["Error Handling"][result

enum] chapter.
{{#quiz ../quizzes/ch06-01-enums.toml}}
[result enum]: ./ch09-02-recoverable-errors.md#the-result-enum