Generic Types and Traits

Every programming language has tools for effectively handling the duplication of concepts. In Cairo, one such tool is generics: abstract stand-ins for concrete types or other properties. We can express the behavior of generics or how they relate to other generics without knowing what will be in their place when compiling and running the code.

Functions can take parameters of some generic type, instead of a concrete type like `u32` or `bool`, in the same way a function takes parameters with unknown values to run the same code on multiple concrete values. In fact, we've already used generics in [Chapter 6][option enum] with `Option<T>`.

In this chapter, you'll explore how to define your own types, functions, and traits with generics.

Generics allow us to replace specific types with a placeholder that represents multiple types to remove code duplication. Upon compilation, the compiler creates a new definition for each concrete type that replaces a generic type, reducing development time for the programmer, but code duplication at compile level still exists. This may be of importance if you are writing Starknet contracts and using a generic for multiple types which will cause contract size to increment.

Then you'll learn how to use traits to define behavior in a generic way. You can combine traits with generic types to constrain a generic type to accept only those types that have a particular behavior, as opposed to just any type.

[option enum]: ./ch06-01-enums.md#the-option-enum-and-its-advantages

Removing Duplication by Extracting a Function

Generics allow us to replace specific types with a placeholder that represents multiple types to remove code duplication. Before diving into generics syntax, let's first look at how to remove duplication in a way that doesn't involve generic types by extracting a function that replaces specific values with a placeholder that represents multiple values. Then we'll apply the same technique to extract a generic function! By learning how to identify duplicated code that can be extracted into a function, you'll start to recognize instances where generics can be used to reduce duplication.

We begin with a short program that finds the largest number in an array of `u8`:

```cairo
{{#include ../listings/ch08-generic-types-and-traits/listing_08_01_extracting_function_01/src/lib.cairo}}
```

We store an array of `u8` in the variable `number_list` and extract the first number in the array in a variable named `largest`. We then iterate through all the numbers in the array, and if the current number is greater than the number stored in `largest`, we update the value of `largest`. However, if the current number is less than or equal to the largest number seen so far, the variable doesn't change, and the code moves on to the next number in the list. After considering all the numbers in the array, `largest` should contain the largest number, which in this case is 100.

We've now been tasked with finding the largest number in two different arrays of numbers. To do so, we can choose to duplicate the previous code and use the same logic at two different places in the program, as follows:

```cairo
```

{{#include ../listings/ch08-generic-types-and-traits/listing_08_01_extracting_function_02/src/lib.cairo}}
```
Although this code works, duplicating code is tedious and error-prone. We also have to remember to update the code in multiple places when we want to change it.
To eliminate this duplication, we'll create an abstraction by defining a function that operates on any array of `u8` passed in a parameter. This solution makes our code clearer and lets us express the concept of finding the largest number in an array abstractly.
To do that, we extract the code that finds the largest number into a function named `largest`. Then we call the function to find the largest number in the two arrays. We could also use the function on any other array of `u8` values we might have in the future.
```cairo
{{#include ../listings/ch08-generic-types-and-traits/listing_08_01_extracting_function_03/src/lib.cairo}}
```
The largest function has a parameter called `number_list`, passed by reference, which represents any concrete array of `u8` values we might pass into the function. As a result, when we call the function, the code runs on the specific values that we pass in.
In summary, here are the steps we took to change the code:
- Identify duplicate code.
- Extract the duplicate code into the body of the function and specify the inputs and return values of that code in the function signature.
- Update the two instances of duplicated code to call the function instead.
Next, we'll use these same steps with generics to reduce code duplication. In the same way that the function body can operate on an abstract `Array<T>` instead of specific `u8` values, generics allow code to operate on abstract types.
```