Appendix C - Derivable Traits
In various places in the book, we’ve discussed the `derive` attribute, which you can apply to a struct or enum definition. The `derive` attribute generates code to implement a default trait on the type you’ve annotated with the `derive` syntax.
In this appendix, we provide a comprehensive reference detailing all the traits in the standard library compatible with the `derive` attribute.
These traits listed here are the only ones defined by the core library that can be implemented on your types using `derive`. Other traits defined in the standard library don’t have sensible default behavior, so it’s up to you to implement them in a way that makes sense for what you’re trying to accomplish.
Drop and Destruct
When moving out of scope, variables need to be moved first. This is where the `Drop` trait intervenes. You can find more details about its usage [here](ch04-01-what-is-ownership.md#no-op-destruction-the-drop-trait).
Moreover, Dictionaries need to be squashed before going out of scope. Calling the `squash` method on each of them manually can quickly become redundant. `Destruct` trait allows Dictionaries to be automatically squashed when they get out of scope. You can also find more information about `Destruct` [here](ch04-01-what-is-ownership.md#destruction-with-a-side-effect-the-destruct-trait).
`Clone` and `Copy` for Duplicating Values
The `Clone` trait provides the functionality to explicitly create a deep copy of a value. Deriving `Clone` implements the `clone` method, which, in turn, calls clone on each of the type's components. This means all the fields or values in the type must also implement `Clone` to derive `Clone`.
Here is a simple example:
```cairo
{{#include ../listings/appendix/listing_01_clone/src/lib.cairo}}
```
The `Copy` trait allows for the duplication of values. You can derive `Copy` on any type whose parts all implement `Copy`.
Example:
```cairo
{{#include ../listings/appendix/listing_02_copy/src/lib.cairo}}
```
`Debug` for Printing and Debugging
The `Debug` trait enables debug formatting in format strings, which you indicate by adding `:?` within `{}` placeholders.
It allows you to print instances of a type for debugging purposes, so you and other programmers using this type can inspect an instance at a particular point in a program’s execution.
For example, if you want to print the value of a variable of type `Point`, you can do it as follows:
```cairo
{{#include ../listings/appendix/listing_03_debug/src/lib.cairo}}
```
```shell
```

scarb cairo-run
Point { x: 1, y: 3 }
~~~
The `Debug` trait is required, for example, when using the `assert_xx!` macros in tests. Theses macros print the values of instances given as arguments if the equality or comparison assertion fails so programmers can see why the two instances weren’t equal.
## `Default` for Default Values
The `Default` trait allows creation of a default value of a type. The most common default value is zero. All primitive types in the standard library implement `Default`.
If you want to derive `Default` on a composite type, each of its elements must already implement `Default`. If you have an [`enum`](ch06-01-enums.md) type, you must declare its default value by using the `#[default]` attribute on one of its variants.
An example:
```cairo
{{#include ../listings/appendix/listing_07_default/src/lib.cairo}}
~~~
`PartialEq` for Equality Comparisons
The `PartialEq` trait allows for comparison between instances of a type for equality, thereby enabling the `==` and `!=` operators.
When `PartialEq` is derived on structs, two instances are equal only if all their fields are equal; they are not equal if any field is different. When derived for enums, each variant is equal to itself and not equal to the other variants.
You can write your own implementation of the `PartialEq` trait for your type, if you can't derive it or if you want to implement your custom rules. In the following example, we write an implementation for `PartialEq` in which we consider that two rectangles are equal if they have the same area:
```cairo
{{#include ../listings/appendix/listing_04_implpartialeq/src/lib.cairo}}
```
The `PartialEq` trait is required when using the `assert_eq!` macro in tests, which needs to be able to compare two instances of a type for equality.
Here is an example:
```cairo
{{#include ../listings/appendix/listing_05_partialeq/src/lib.cairo}}
```
Serializing with `Serde`
`Serde` provides trait implementations for `serialize` and `deserialize` functions for data structures defined in your crate. It allows you to transform your structure into an array (or the opposite).
> **[Serialization](https://en.wikipedia.org/wiki/Serialization)** is a process of transforming data structures into a format that can be easily stored or transmitted. Let's say you are running a program and would like to persist its state to be able to resume it later. To do this, you could take each of the objects your program is using and save their information, for example in a file. This is a simplified version of serialization. Now if you want to resume your program with this saved state, you would perform

deserialization, which means loading the state of the objects from the saved source.
For example:
```cairo
{{#include ../listings/appendix/listing_06_serialize/src/lib.cairo}}
```
If you run the `main` function, the output will be:
```shell
Run panicked with [2, 99 ('c'), ].
```
We can see here that our struct `A` has been serialized into the output array. Note that the `serialize` function takes as argument a snapshot of the type you want to convert into an array. This is why deriving `Drop` for `A` is required here, as the `main` function keeps ownership of the `first_struct` struct.
Also, we can use the `deserialize` function to convert the serialized array back into our `A` struct.
Here is an example:
```cairo
{{#include ../listings/appendix/listing_07_deserialize/src/lib.cairo}}
```
Here we are converting a serialized array span back to the struct `A`. `deserialize` returns an `Option` so we need to unwrap it. When using `deserialize` we also need to specify the type we want to deserialize into.
Hashing with `Hash`
It is possible to derive the `Hash` trait on structs and enums. This allows them to be hashed easily using any available hash function. For a struct or an enum to derive the `Hash` attribute, all fields or variants need to be hashable themselves.
You can refer to the [Hashes section](ch11-04-hash.md) to get more information about how to hash complex data types.
Starknet Storage with `starknet::Store`
The `starknet::Store` trait is relevant only when building on [Starknet](ch13-00-introduction-to-starknet-smart-contracts.md). It allows for a type to be used in smart contract storage by automatically implementing the necessary read and write functions.
You can find detailed information about the inner workings of Starknet storage in the [Contract storage section](ch14-01-00-contract-storage.md).