References and Snapshots
The issue with the tuple code in previous Listing {{#ref return-multiple-values}} is that we have to return the
`Array` to the calling function so we can still use the `Array` after the
call to `calculate_length`, because the `Array` was moved into
`calculate_length`.
Snapshots
In the previous chapter, we talked about how Cairo's ownership system prevents us from using a variable after we've moved it, protecting us from potentially writing twice to the same memory cell. However, it's not very convenient.
Let's see how we can retain ownership of the variable in the calling function using snapshots.
In Cairo, a snapshot is an immutable view of a value at a certain point in time.
Recall that memory is immutable, so modifying a value actually creates a new memory cell.
The old memory cell still exists, and snapshots are variables that refer to that "old" value.
In this sense, snapshots are a view "into the past".
Here is how you would define and use a `calculate_length` function that takes a snapshot of an array as a parameter instead of taking ownership of the underlying value. In this example,
the `calculate_length` function returns the length of the array passed as a parameter.
As we're passing it as a snapshot, which is an immutable view of the array, we can be sure that
the `calculate_length` function will not mutate the array, and ownership of the array is kept in the `main` function.
Filename: src/lib.cairo
```cairo
{{#include ../listings/ch04-understanding-ownership/no_listing_09_snapshots/src/lib.cairo}}
```
> Note: it is only possible to call the `len()` method on an array snapshot because it is defined as such in the `ArrayTrait` trait. If you try to call a method that is not defined for snapshots on a snapshot, you will get a compilation error. However, you can call methods expecting a snapshot on non-snapshot types.
The output of this program is:
```shell
{{#include ../listings/ch04-understanding-ownership/no_listing_09_snapshots/output.txt}}
```
First, notice that all the tuple code in the variable declaration and the function return value is gone. Second, note
that we pass `@arr1` into `calculate_length` and, in its definition, we take
`@Array<u128>` rather than `Array<u128>`.
Let’s take a closer look at the function call here:
```cairo
let second_length = calculate_length(@arr1); // Calculate the current length of the array
```

```
The `@arr1` syntax lets us create a snapshot of the value in `arr1`. Because a snapshot is an immutable view of a value at a specific point in time, the usual rules of the linear type system are not enforced. In particular, snapshot variables always implement the `Drop` trait, never the `Destruct` trait, even dictionary snapshots. Similarly, the signature of the function uses `@` to indicate that the type of the parameter `arr` is a snapshot. Let’s add some explanatory annotations:
```cairo, noplayground
fn calculate_length(
    array_snapshot: @Array<u128> // array_snapshot is a snapshot of an Array
) -> usize {
    array_snapshot.len()
} // Here, array_snapshot goes out of scope and is dropped.
// However, because it is only a view of what the original array `arr` contains, the original `arr` can still be used.
```
The scope in which the variable `array_snapshot` is valid is the same as any function parameter’s scope, but the underlying value of the snapshot is not dropped when `array_snapshot` stops being used. When functions have snapshots as parameters instead of the actual values, we won’t need to return the values in order to give back ownership of the original value, because we never had it.
Desnap Operator
To convert a snapshot back into a regular variable, you can use the `desnap` operator `*`, which serves as the opposite of the `@` operator.
Only `Copy` types can be desnapped. However, in the general case, because the value is not modified, the new variable created by the `desnap` operator reuses the old value, and so desnapping is a completely free operation, just like `Copy`.
In the following example, we want to calculate the area of a rectangle, but we don't want to take ownership of the rectangle in the `calculate_area` function, because we might want to use the rectangle again after the function call. Since our function doesn't mutate the rectangle instance, we can pass the snapshot of the rectangle to the function, and then transform the snapshots back into values using the `desnap` operator `*`.
```cairo
{{#include ../listings/ch04-understanding-ownership/no_listing_10_desnap/src/lib.cairo}}
```
But, what happens if we try to modify something we’re passing as a snapshot? Try the code in
Listing {{#ref modify-snapshot}}. Spoiler alert: it doesn’t work!
Filename: src/lib.cairo
```cairo,does_not_compile
{{#include ../listings/ch04-understanding-ownership/
listing_04_attempt_modifying_snapshot/src/lib.cairo}}
```
{{#label modify-snapshot}}
Listing {{#ref modify-snapshot}}: Attempting to modify a snapshot value
```

Here’s the error:
```shell
{{#include ../listings/ch04-understanding-ownership/
listing_04_attempt_modifying_snapshot/output.txt}}
```
The compiler prevents us from modifying values associated to snapshots.
Mutable References
We can achieve the behavior we want in Listing {{#ref modify-snapshot}} by using a _mutable reference_ instead of a snapshot. Mutable references are actually mutable values passed to a function that are implicitly returned at the end of the function, returning ownership to the calling context. By doing so, they allow you to mutate the value passed while keeping ownership of it by returning it automatically at the end of the execution.
In Cairo, a parameter can be passed as _mutable reference_ using the `ref` modifier.
> **Note**: In Cairo, a parameter can only be passed as _mutable reference_ using the `ref` modifier if the variable is declared as mutable with `mut`.
In Listing 4-5, we use a mutable reference to modify the value of the `height` and `width` fields of the `Rectangle` instance in the `flip` function.
```cairo
{{#include ../listings/ch04-understanding-ownership/listing_05_mutable_reference/src/
lib.cairo}}
```
Listing 4-5: Use of a mutable reference to modify a value
First, we change `rec` to be `mut`. Then we pass a mutable reference of `rec` into `flip` with `ref rec`, and update the function signature to accept a mutable reference with `ref rec: Rectangle`. This makes it very clear that the `flip` function will mutate the value of the `Rectangle` instance passed as parameter.
The output of the program is:
```shell
{{#include ../listings/ch04-understanding-ownership/listing_05_mutable_reference/
output.txt}}
```
As expected, the `height` and `width` fields of the `rec` variable have been swapped.
{{#quiz ../quizzes/ch04-02-references-and-snapshots.toml}}
Small Recap
Let’s recap what we’ve discussed about the linear type system, ownership, snapshots, and references:
- At any given time, a variable can only have one owner.
- You can pass a variable by-value, by-snapshot, or by-reference to a function.
- If you pass-by-value, ownership of the variable is transferred to the function.
- If you want to keep ownership of the variable and know that your function won’t mutate it, you can pass it as a snapshot with `@`.
- If you want to keep ownership of the variable and know that your function will mutate it, you can pass it as a mutable reference with `ref`.