Arrays
An array is a collection of elements of the same type. You can create and use array methods by using the `ArrayTrait` trait from the core library.
An important thing to note is that arrays have limited modification options. Arrays are, in fact, queues whose values can't be modified.
This has to do with the fact that once a memory slot is written to, it cannot be overwritten, but only read from it. You can only append items to the end of an array and remove items from the front.
Creating an Array
Creating an array is done with the `ArrayTrait::new()` call. Here's an example of creating an array and appending 3 elements to it::
```cairo
{{#include ../listings/ch03-common-collections/no_listing_01_array_new_append/src/lib.cairo}}
```
When required, you can pass the expected type of items inside the array when instantiating the array like this, or explicitly define the type of the variable.
```cairo, noplayground
let mut arr = ArrayTrait::<u128>::new();
```
```cairo, noplayground
let mut arr:Array<u128> = ArrayTrait::new();
```
Updating an Array
Adding Elements
To add an element to the end of an array, you can use the `append()` method:
```cairo
{{#rustdoc_include ../listings/ch03-common-collections/no_listing_01_array_new_append/src/lib.cairo:5}}
```
Removing Elements
You can only remove elements from the front of an array by using the `pop_front()` method.
This method returns an `Option` that can be unwrapped, containing the removed element, or `Option::None` if the array is empty.
```cairo
{{#include ../listings/ch03-common-collections/no_listing_02_array_pop_front/src/lib.cairo}}
```
The above code will print `The first value is 10` as we remove the first element that was added.
In Cairo, memory is immutable, which means that it is not possible to modify the elements of an array once they've been added. You can only add elements to the end of an array and remove elements from the front of an array. These operations do not require memory mutation, as they involve updating pointers rather than directly modifying the memory cells.

Reading Elements from an Array
To access array elements, you can use `get()` or `at()` array methods that return different types. Using `arr.at(index)` is equivalent to using the subscripting operator `arr[index]`.
`get()` Method
The `get` function returns an `Option<Box<@T>>`, which means it returns an option to a Box type (Cairo's smart-pointer type) containing a snapshot to the element at the specified index if that element exists in the array. If the element doesn't exist, `get` returns `None`. This method is useful when you expect to access indices that may not be within the array's bounds and want to handle such cases gracefully without panics. Snapshots will be explained in more detail in the ["References and Snapshots"][snapshots] chapter.
Here is an example with the `get()` method:
```cairo
{{#include ../listings/ch03-common-collections/no_listing_03_array_get/src/lib.cairo}}
```
[snapshots]: ./ch04-02-references-and-snapshots.md#snapshots
`at()` Method
The `at` function, and its equivalent the subscripting operator, on the other hand, directly return a snapshot to the element at the specified index using the `unbox()` operator to extract the value stored in a box. If the index is out of bounds, a panic error occurs. You should only use `at` when you want the program to panic if the provided index is out of the array's bounds, which can prevent unexpected behavior.
```cairo
{{#include ../listings/ch03-common-collections/no_listing_04_array_at/src/lib.cairo}}
```
In this example, the variable named `first` will get the value `0` because that
is the value at index `0` in the array. The variable named `second` will get
the value `1` from index `1` in the array.
In summary, use `at` when you want to panic on out-of-bounds access attempts, and use `get` when you prefer to handle such cases gracefully without panicking.
Size-related Methods
To determine the number of elements in an array, use the `len()` method. The return value is of type `usize`.
If you want to check if an array is empty or not, you can use the `is_empty()` method, which returns `true` if the array is empty and `false` otherwise.
`array!` Macro
Sometimes, we need to create arrays with values that are already known at compile time. The basic way of doing that is redundant. You would first declare the array and then append each value one by one. `array!` is a simpler way of doing this task by combining the two steps.
At compile-time, the compiler will expand the macro to generate the code that appends the items sequentially.
Without `array!`:
```cairo
{{#include ../listings/ch03-common-collections/no_listing_06_array_macro/src/
lib.cairo:no_macro}}
```
With `array!`:
```cairo
{{#include ../listings/ch03-common-collections/no_listing_06_array_macro/src/
lib.cairo:array_macro}}
```
Storing Multiple Types with Enums
If you want to store elements of different types in an array, you can use an `Enum` to define a custom data type that can hold multiple types. Enums will be explained in more detail in the ["Enums and Pattern Matching"][enums] chapter.
```cairo
{{#include ../listings/ch03-common-collections/no_listing_07_array_with_enums/src/
lib.cairo}}
```
[enums]: ./ch06-00-enums-and-pattern-matching.md
Span
`Span` is a struct that represents a snapshot of an `Array`. It is designed to provide safe and controlled access to the elements of an array without modifying the original array. Span is particularly useful for ensuring data integrity and avoiding borrowing issues when passing arrays between functions or when performing read-only operations, as introduced in ["References and Snapshots"][references].
All methods provided by `Array` can also be used with `Span`, except for the `append()` method.
[references]: ./ch04-02-references-and-snapshots.md
Turning an Array into Span
To create a `Span` of an `Array`, call the `span()` method:
```cairo
{{#rustdoc_include ../listings/ch03-common-collections/no_listing_08_array_span/src/
lib.cairo:3}}
```
{{#quiz ../quizzes/ch03-01-arrays.toml}}