Recoverable Errors with `Result`
Most errors aren’t serious enough to require the program to stop entirely. Sometimes, when a function fails, it’s for a reason that you can easily interpret and respond to. For example, if you try to add two large integers and the operation overflows because the sum exceeds the maximum representable value, you might want to return an error or a wrapped result instead of causing undefined behavior or terminating the process.
The `Result` Enum
Recall from [Generic data types][generic enums] section in Chapter 8 that the `Result` enum is defined as having two variants, `Ok` and `Err`, as follows:
```cairo,noplayground
{{#include ../listings/ch09-error-handling/no_listing_07_result_enum/src/lib.cairo}}
```
The `Result<T, E>` enum has two generic types, `T` and `E`, and two variants: `Ok` which holds the value of type `T` and `Err` which holds the value of type `E`. This definition makes it convenient to use the `Result` enum anywhere we have an operation that might succeed (by returning a value of type `T`) or fail (by returning a value of type `E`).
[generic enums]: ./ch08-01-generic-data-types.md#enums
The `ResultTrait`
The `ResultTrait` trait provides methods for working with the `Result<T, E>` enum, such as unwrapping values, checking whether the `Result` is `Ok` or `Err`, and panicking with a custom message. The `ResultTraitImpl` implementation defines the logic of these methods.
```cairo,noplayground
{{#include ../listings/ch09-error-handling/no_listing_08_result_trait/src/lib.cairo}}
```
The `expect` and `unwrap` methods are similar in that they both attempt to extract the value of type `T` from a `Result<T, E>` when it is in the `Ok` variant. If the `Result` is `Ok(x)`, both methods return the value `x`. However, the key difference between the two methods lies in their behavior when the `Result` is in the `Err` variant. The `expect` method allows you to provide a custom error message (as a `felt252` value) that will be used when panicking, giving you more control and context over the panic. On the other hand, the `unwrap` method panics with a default error message, providing less information about the cause of the panic.
The `expect_err` and `unwrap_err` methods have the exact opposite behavior. If the `Result` is `Err(x)`, both methods return the value `x`. However, the key difference between the two methods is in case of `Result::Ok()`. The `expect_err` method allows you to provide a custom error message (as a `felt252` value) that will be used when panicking, giving you more control and context over the panic. On the other hand, the `unwrap_err` method panics with a default error message, providing less information about the cause of the panic.
A careful reader may have noticed the `<+Drop<T>>` and `<+Drop<E>>` in the first four methods signatures. This syntax represents generic type constraints in the Cairo language, as seen in the previous chapter. These constraints indicate that the associated functions require an implementation of the `Drop` trait for the generic types `T` and `E`, respectively.

Finally, the `is_ok` and `is_err` methods are utility functions provided by the `ResultTrait` trait to check the variant of a `Result` enum value.

- `is_ok` takes a snapshot of a `Result<T, E>` value and returns `true` if the `Result` is the `Ok` variant, meaning the operation was successful. If the `Result` is the `Err` variant, it returns `false`.
- `is_err` takes a snapshot of a `Result<T, E>` value and returns `true` if the `Result` is the `Err` variant, meaning the operation encountered an error. If the `Result` is the `Ok` variant, it returns `false`.

These methods are helpful when you want to check the success or failure of an operation without consuming the `Result` value, allowing you to perform additional operations or make decisions based on the variant without unwrapping it.

You can find the implementation of the `ResultTrait` [here][result corelib].

It is always easier to understand with examples. Have a look at this function signature:

```cairo,noplayground
{{#include ../listings/ch09-error-handling/no_listing_09_result_example/src/
lib.cairo:overflow}}
```

It takes two `u128` integers, `a` and `b`, and returns a `Result<u128, u128>` where the `Ok` variant holds the sum if the addition does not overflow, and the `Err` variant holds the overflowed value if the addition does overflow.

Now, we can use this function elsewhere. For instance:

```cairo,noplayground
{{#include ../listings/ch09-error-handling/no_listing_09_result_example/src/
lib.cairo:checked-add}}
```

Here, it accepts two `u128` integers, `a` and `b`, and returns an `Option<u128>`. It uses the `Result` returned by `u128_overflowing_add` to determine the success or failure of the addition operation. The `match` expression checks the `Result` from `u128_overflowing_add`. If the result is `Ok(r)`, it returns `Option::Some(r)` containing the sum. If the result is `Err(r)`, it returns `Option::None` to indicate that the operation has failed due to overflow. The function does not panic in case of an overflow.

Let's take another example:

```cairo,noplayground
{{#include ../listings/ch09-error-handling/listing_09_01/src/lib.cairo:function}}
```

In this example, the `parse_u8` function takes a `felt252` and tries to convert it into a `u8` integer using the `try_into` method. If successful, it returns `Result::Ok(value)`, otherwise it returns `Result::Err('Invalid integer')`.

Our two test cases are:

```cairo,noplayground
{{#rustdoc_include ../listings/ch09-error-handling/listing_09_01/src/lib.cairo:tests}}
```

Don't worry about the `#[cfg(test)]` attribute for now. We'll explain in more detail its meaning in the next [Testing Cairo Programs][tests] chapter.

`#[test]` attribute means the function is a test function, and `#[should_panic]` attribute means this test will pass if the test execution panics.

The first one tests a valid conversion from `felt252` to `u8`, expecting the `unwrap` method not to panic. The second test function attempts to convert a value that is out of the `u8` range, expecting the `unwrap` method to panic with the error message `Invalid integer`.
[result corelib]: https://github.com/starkware-libs/cairo/blob/main/corelib/src/result.cairo#L20
[tests]: ./ch10-01-how-to-write-tests.md
The `?` Operator
The last operator we will talk about is the `?` operator. The `?` operator is used for more idiomatic and concise error handling. When you use the `?` operator on a `Result` or `Option` type, it will do the following:
- If the value is `Result::Ok(x)` or `Option::Some(x)`, it will return the inner value `x` directly.
- If the value is `Result::Err(e)` or `Option::None`, it will propagate the error or `None` by immediately returning from the function.

The `?` operator is useful when you want to handle errors implicitly and let the calling function deal with them.
Here is an example:
```cairo,noplayground
{{#include ../listings/ch09-error-handling/listing_09_02/src/lib.cairo:function}}
```
We can see that `do_something_with_parse_u8` function takes a `felt252` value as input and calls `parse_u8` function. The `?` operator is used to propagate the error, if any, or unwrap the successful value.
And with a little test case:
```cairo,noplayground
{{#rustdoc_include ../listings/ch09-error-handling/listing_09_02/src/lib.cairo:tests}}
```
The console will print the error `Invalid Integer`.
Summary
We saw that recoverable errors can be handled in Cairo using the `Result` enum, which has two variants: `Ok` and `Err`. The `Result<T, E>` enum is generic, with types `T` and `E` representing the successful and error values, respectively. The `ResultTrait` provides methods for working with `Result<T, E>`, such as unwrapping values, checking if the result is `Ok` or `Err`, and panicking with custom messages.
To handle recoverable errors, a function can return a `Result` type and use pattern matching to handle the success or failure of an operation. The `?` operator can be used to implicitly handle errors by propagating the error or unwrapping the successful value. This allows for more concise and clear error handling, where the caller is responsible for managing errors raised by the called function.
{{#quiz ../quizzes/ch09-02-error-handling-result.toml}}