The Move Book

This is the Move Book - a comprehensive guide to the Move programming language and the Sui blockchain. The book is intended for developers who are interested in learning about Move and building on Sui.

ⓘ The book is in active development and a work in progress. If you have any feedback or suggestions, feel free to open an issue or a pull request on the GitHub repository.

> If you're looking for The Move Reference, you can find it here.

Foreword

This book is dedicated to Move, a smart contract language that captures the essence of safe programming with digital assets. Move is designed around the following values:

1. Secure by default: Insecure languages are a serious barrier both to accessible smart contract development and to mainstream adoption of digital assets. The first duty of a smart contract language is to prevent as many potential safety issues as possible (e.g. re-entrancy, missing access control checks, arithmetic overflow, ...) by construction. Any changes to Move should preserve or enhance its existing security guarantees.

2. Expressive by nature: Move must enable programmers to write any smart contract they can imagine. But we care as much about the way it *feels* to write Move as we do about what Move allows you to do - the language should be rich enough that the features needed for a task are available, and minimal enough that the choice is obvious. The Move toolchain should be a productivity enhancer and a thought partner.

3. Intuitive for all: Smart contracts are only one part of a useful application. Move should understand the broader context of its usage and design with both the smart contract developer and the application developer in mind. It should be easy for developers to learn how to read Move-managed state, build Move powered transactions, and write new Move code.

The core technical elements of Move are:

- Safe, familiar, and flexible abstractions for digital assets via programmable *objects*.
- A rich *ability* system (inspired by linear types) that gives programmers extreme control of how values are created, destroyed, stored, copied, and transferred.
- A *module* system with strong encapsulation features to enable code reuse while maintaining this control.
- *Dynamic fields* for creating hierarchical relationships between objects.
- *Programmable transaction blocks* (PTBs) to enable atomic client-side composition of Move-powered APIs.

Move was born in 2018 as part of Facebook's Libra project. It was publicly revealed in 2019, the first Move-powered network launched in 2020. As of April 2024, there are numerous Move-powered chains in production with several more in the works. Move is an embedded language with a platform-agnostic core, which means it takes on a slightly different personality in each chain that uses it.

Creating a new programming language and bootstrapping a community around it is an ambitious, long term project. A language has to be an order of magnitude better than alternatives in relevant ways to have a chance, but even then the quality of the community matters more than the

technical fundamentals. Move is a young language, but it's off to a good start in terms of both differentiation and community. A small, but fanatical group of smart contract programmers and core contributors united by the Move values are pushing the boundaries of what smart contracts can do, the applications they can enable, and who can (safely) write them. If that inspires you, read on!

— Sam Blackshear, creator of Move

Before we begin

Move requires an environment to run and develop applications, and in this small chapter we will cover the prerequisites for the Move language: how to set up your IDE, how to install the compiler and what is Move 2024. If you are already familiar with these topics or have a CLI installed, you can skip this chapter and proceed to the next one.

Install Sui

Move is a compiled language, so you need to install a compiler to be able to write and run Move programs. The compiler is included into the Sui binary, which can be installed or downloaded using one of the methods below.

Download Binary

You can download the latest Sui binary from the releases page. The binary is available for macOS, Linux and Windows. For education purposes and development, we recommend using the `mainnet` version.

Install using Homebrew (MacOS)

You can install Sui using the Homebrew package manager.

```
brew install sui
```

Install using Chocolatey (Windows)

You can install Sui using the Chocolatey package manager for Windows.

```
choco install sui
```

Build using Cargo (MacOS, Linux)

You can install and build Sui locally by using the Cargo package manager (requires Rust)

```
cargo install --git https://github.com/MystenLabs/sui.git --bin sui --branch
mainnet
```

Make sure that your system has the latest Rust versions with the command below.

```
rustup update stable
```

Troubleshooting

For troubleshooting the installation process, please refer to the Install Sui Guide.

Set up your IDE

There are two most popular IDEs for Move development: VSCode and IntelliJ IDEA. Both of them provide basic features like syntax highlighting and error messages, though they differ in their additional features. Whatever IDE you choose, you'll need to use the terminal to run the Move CLI.

> IntelliJ Plugin does not support Move 2024 edition, some syntax won't get highlighted.

VSCode

- VSCode is a free and open source IDE from Microsoft.
- Move (Extension) is a language server extension for Move maintained by MystenLabs.
- Move Syntax a simple syntax highlighting extension for Move by Damir Shamanaev.

IntelliJ IDEA

- IntelliJ IDEA is a commercial IDE from JetBrains.
- Move Language Plugin provides a Move language extension for IntelliJ IDEA by Pontem Network.

Emacs

- Emacs is a free and open source text editor.
- move-mode is a Move mode for Emacs by Ashok Menon.

Github Codespaces

The Web-based IDE from Github can be run right in the browser and provides almost a full-featured VSCode experience.

- Github Codespaces

- Move Syntax is also available in the extensions marketplace.

Move 2024

Move 2024 is the new edition of the Move language maintained by Mysten Labs. All of the examples in this book are written in Move 2024. If you're used to the pre-2024 version of Move, please, refer to the Move 2024 Migration Guide to learn about the changes and improvements in the new edition.

Hello, World!

In this chapter, you will learn how to create a new package, write a simple module, compile it, and run tests with the Move CLI. Make sure you have installed Sui and set up your IDE environment. Run the command below to test if Sui has been installed correctly.

```
# It should print the client version. E.g. sui-client 1.22.0-036299745.
sui client --version
```

> Move CLI is a command-line interface for the Move language; it is built into the Sui binary and provides a set of commands to manage packages, compile and test code.

The structure of the chapter is as follows:

- Create a New Package
- Directory Structure
- Compiling the Package
- Running Tests

Create a New Package

To create a new program, we will use the `sui move new` command followed by the name of the application. Our first program will be called `hello_world`.

> Note: In this and other chapters, if you see code blocks with lines starting with `$` (dollar sign), it means that the following command should be run in a terminal. The sign should not be included. It's a common way of showing commands in terminal environments.

```
$ sui move new hello_world
```

The `sui move` command gives access to the Move CLI - a built-in compiler, test runner and a utility for all things Move. The `new` command followed by the name of the package will create a new package in a new folder. In our case, the folder name is "hello_world".

We can view the contents of the folder to see that the package was created successfully.

```
$ ls -l hello_world
Move.toml
sources
tests
```

Directory Structure

Move CLI will create a scaffold of the application and pre-create the directory structure and all necessary files. Let's see what's inside.

```
hello_world
├── Move.toml
├── sources
│   └── hello_world.move
└── tests
    └── hello_world_tests.move
```

Manifest

The `Move.toml` file, known as the package manifest, contains definitions and configuration settings for the package. It is used by the Move Compiler to manage package metadata, fetch dependencies, and register named addresses. We will explain it in detail in the Concepts chapter.

> By default, the package features one named address - the name of the package.

```
[addresses]
hello_world = "0x0"
```

Sources

The `sources/` directory contains the source files. Move source files have *.move* extension, and are typically named after the module defined in the file. For example, in our case, the file name is *hello_world.move* and the Move CLI has already placed commented out code inside:

```
/*
/// Module: hello_world
module hello_world::hello_world {

}
*/
```

> The `/*` and `*/` are the comment delimiters in Move. Everything in between is ignored by the compiler and can be used for documentation or notes. We explain all ways to comment the code in the Basic Syntax.

The commented out code is a module definition, it starts with the keyword `module` followed by a named address (or an address literal), and the module name. The module name is a unique identifier for the module and has to be unique within the package. The module name is used to reference the module from other modules or transactions.

Tests

The `tests/` directory contains package tests. The compiler excludes these files in the regular build process but uses them in *test* and *dev* modes. The tests are written in Move and are marked with the `#[test]` attribute. Tests can be grouped in a separate module (then it's usually called *module_name_tests.move*), or inside the module they're testing.

Modules, imports, constants and functions can be annotated with `#[test_only]`. This attribute is used to exclude modules, functions or imports from the build process. This is useful when you want to add helpers for your tests without including them in the code that will be published on chain.

The *hello_world_tests.move* file contains a commented out test module template:

```
/*
#[test_only]
module hello_world::hello_world_tests {
    // uncomment this line to import the module
    // use hello_world::hello_world;

    const ENotImplemented: u64 = 0;

    #[test]
    fun test_hello_world() {
        // pass
    }

    #[test, expected_failure(abort_code =
hello_world::hello_world_tests::ENotImplemented)]
    fun test_hello_world_fail() {
        abort ENotImplemented
    }
}
*/
```

Other Folders

Additionally, Move CLI supports the `examples/` folder. The files there are treated similarly to the ones placed under the `tests/` folder - they're only built in the *test* and *dev* modes. They are to be examples of how to use the package or how to integrate it with other packages. The most popular use case is for documentation purposes and library packages.

Compiling the Package

Move is a compiled language, and as such, it requires the compilation of source files into Move Bytecode. It contains only necessary information about the module, its members, and types, and excludes comments and some identifiers (for example, for constants).

To demonstrate these features, let's replace the contents of the *sources/hello_world.move* file with the following:

```
/// The module `hello_world` under named address `hello_world`.
/// The named address is set in the `Move.toml`.
module hello_world::hello_world {
    // Imports the `String` type from the Standard Library
    use std::string::String;

    /// Returns the "Hello, World!" as a `String`.
    public fun hello_world(): String {
        b"Hello, World!".to_string()
    }
}
```

During compilation, the code is built, but not run. A compiled package only includes functions that can be called by other modules or in a transaction. We will explain these concepts in the Concepts chapter. But now, let's see what happens when we run the *sui move build*.

```
# run from the `hello_world` folder
$ sui move build

# alternatively, if you didn't `cd` into it
$ sui move build --path hello_world
```

It should output the following message on your console.

```
UPDATING GIT DEPENDENCY https://github.com/MystenLabs/sui.git
INCLUDING DEPENDENCY Sui
INCLUDING DEPENDENCY MoveStdlib
BUILDING hello_world
```

During the compilation, Move Compiler automatically creates a build folder where it places all fetched and compiled dependencies as well as the bytecode for the modules of the current package.

> If you're using a versioning system, such as Git, build folder should be ignored. For example, you should use a `.gitignore` file and add `build` to it.

Running Tests

Before we get to testing, we should add a test. Move Compiler supports tests written in Move and provides the execution environment. The tests can be placed in both the source files and in the `tests/` folder. Tests are marked with the `#[test]` attribute and are automatically discovered by the compiler. We explain tests in depth in the Testing section.

Replace the contents of the `tests/hello_world_tests.move` with the following content:

```
#[test_only]
module hello_world::hello_world_tests {
    use hello_world::hello_world;

    #[test]
    fun test_hello_world() {
        assert!(hello_world::hello_world() == b"Hello, World!".to_string(), 0);
    }
}
```

Here we import the `hello_world` module, and call its `hello_world` function to test that the output is indeed the string "Hello, World!". Now, that we have tests in place, let's compile the package in the test mode and run tests. Move CLI has the `test` command for this:

```
$ sui move test
```

The output should be similar to the following:

```
INCLUDING DEPENDENCY Sui
INCLUDING DEPENDENCY MoveStdlib
BUILDING hello_world
Running Move unit tests
[ PASS    ] 0x0::hello_world_tests::test_hello_world
Test result: OK. Total tests: 1; passed: 1; failed: 0
```

If you're running the tests outside of the package folder, you can specify the path to the package:

```
$ sui move test --path hello_world
```

You can also run a single or multiple tests at once by specifying a string. All the tests names containing the string will be run:

```
$ sui move test test_hello
```

Next Steps

In this section, we explained the basics of a Move package: its structure, the manifest, the build, and test flows. On the next page, we will write an application and see how the code is structured and what the language can do.

Further Reading

- Package Manifest section
- Package in The Move Reference

Hello, Sui!

In the previous section we created a new package and demonstrated the basic flow of creating, building, and testing a Move package. In this section, we will write a simple application that uses the storage model and can be interacted with. To do this, we will create a simple todo list application.

Create a New Package

Following the same flow as in Hello, World!, we will create a new package called `todo_list`.

```
$ sui move new todo_list
```

Add the code

To speed things up and focus on the application logic, we will provide the code for the todo list application. Replace the contents of the *sources/todo_list.move* file with the following code:

> Note: while the contents may seem overwhelming at first, we will break it down in the following sections. Try to focus on what's at hand right now.

```
/// Module: todo_list
module todo_list::todo_list {
    use std::string::String;

    /// List of todos. Can be managed by the owner and shared with others.
    public struct TodoList has key, store {
        id: UID,
        items: vector<String>
    }

    /// Create a new todo list.
    public fun new(ctx: &mut TxContext): TodoList {
        let list = TodoList {
            id: object::new(ctx),
            items: vector[]
        };

        (list)
    }

    /// Add a new todo item to the list.
    public fun add(list: &mut TodoList, item: String) {
        list.items.push_back(item);
    }

    /// Remove a todo item from the list by index.
    public fun remove(list: &mut TodoList, index: u64): String {
        list.items.remove(index)
    }

    /// Delete the list and the capability to manage it.
    public fun delete(list: TodoList) {
        let TodoList { id, items: _ } = list;
        id.delete();
    }

    /// Get the number of items in the list.
    public fun length(list: &TodoList): u64 {
        list.items.length()
    }
}
```

Build the package

To make sure that we did everything correctly, let's build the package by running the `sui move build` command. If everything is correct, you should see the output similar to the following:

```
$ sui move build
UPDATING GIT DEPENDENCY https://github.com/MystenLabs/sui.git
INCLUDING DEPENDENCY Sui
INCLUDING DEPENDENCY MoveStdlib
BUILDING todo_list
```

If there are no errors following this output, you have successfully built the package. If there are errors, make sure that:

- The code is copied correctly
- The file name and the package name is correct

There are not many other reasons for the code to fail at this stage. But if you are still having issues, try looking up the structure of the package in this location.

Set up an account

To publish and interact with the package, we need to set up an account. For the sake of simplicity and demonstration purposes, we will use *sui devnet* environment.

> If you already have an account set up, you can skip this step.

If you are doing it for the first time, you will need to create a new account. To do this, run the `sui client` command, then the CLI will prompt you with multiple questions. The answers are marked below with `>`:

```
$ sui client
Config file ["/path/to/home/.sui/sui_config/client.yaml"] doesn't exist, do you
want to connect to a Sui Full node server [y/N]?
> y
Sui Full node server URL (Defaults to Sui Testnet if not specified) :
>
Select key scheme to generate keypair (0 for ed25519, 1 for secp256k1, 2: for
secp256r1):
> 0
```

After you have answered the questions, the CLI will generate a new keypair and save it to the configuration file. You can now use this account to interact with the network.

To check that we have the account set up correctly, run the `sui client active-address` command:

```
$ sui client active-address
0x....
```

The command will output the address of your account, it starts with `0x` followed by 64 characters.

Requesting Coins

In *devnet* and *testnet* environments, the CLI provides a way to request coins to your account, so you can interact with the network. To request coins, run the `sui client faucet` command:

```
$ sui client faucet
Request successful. It can take up to 1 minute to get the coin. Run sui client gas
to check your gas coins.
```

After waiting a little bit, you can check that the Coin object was sent to your account by running the `sui client balance` command:

```
$ sui client balance
╭─────────────────────────────────────────────╮
│ Balance of coins owned by this address      │
├─────────────────────────────────────────────┤
│  ╭──────────────────────────────────────╮   │
│  │ coin  balance (raw)  balance         │   │
│  ├──────────────────────────────────────┤   │
│  │ Sui   1000000000    1.00 SUI         │   │
│  ╰──────────────────────────────────────╯   │
╰─────────────────────────────────────────────╯
```

Alternatively, you can query *objects* owned by your account, by running the `sui client objects` command. The actual output will be different, because the object ID is unique, and so is digest, but the structure will be similar:

```
$ sui client objects
╭──────────────────────────────────────────────────────────────────────────────────────────────────────────╮
│                                                                                                          │
├──────────────┬───────────────────────────────────────────────────────────────────────────────────────────┤
│ objectId     │ 0x4ea1303e4f5e2f65fc3709bc0fb70a3035fdd2d53dbcff33e026a50a742ce0de                        │
│ version      │ 4                                                                                         │
│ digest       │ nA68oa8gab/CdIRw+240wze8u0P+sRe4vcisbENcR4U=                                               │
│ objectType   │ 0x0000..0002::coin::Coin                                                                  │
│                                                                                                          │
╰──────────────┴───────────────────────────────────────────────────────────────────────────────────────────╯
```

Now that we have the account set up and the coins in the account, we can interact with the network. We will start by publishing the package to the network.

Publish

To publish the package to the network, we will use the `sui client publish` command. The command will automatically build the package and use its bytecode to publish in a single transaction.

> We are using the `--gas-budget` argument during publishing. It specifies how much gas we are willing to spend on the transaction. We won't touch on this topic in this section, but it's important to know that every transaction in Sui costs gas, and the gas is paid in SUI coins.

The `gas-budget` is specified in *MISTs*. 1 SUI equals 10^9 MISTs. For the sake of demonstration, we will use 100,000,000 MISTs, which is 0.1 SUI.

```
# run this from the `todo_list` folder
$ sui client publish --gas-budget 100000000

# alternatively, you can specify path to the package
$ sui client publish --gas-budget 100000000 todo_list
```

The output of the publish command is rather lengthy, so we will show and explain it in parts.

```
$ sui client publish --gas-budget 100000000
UPDATING GIT DEPENDENCY https://github.com/MystenLabs/sui.git
INCLUDING DEPENDENCY Sui
INCLUDING DEPENDENCY MoveStdlib
BUILDING todo_list
Successfully verified dependencies on-chain against source.
Transaction Digest: GpcDV6JjjGQMRwHpEz582qsd5MpCYgSwrDAq1JXcpFjW
```

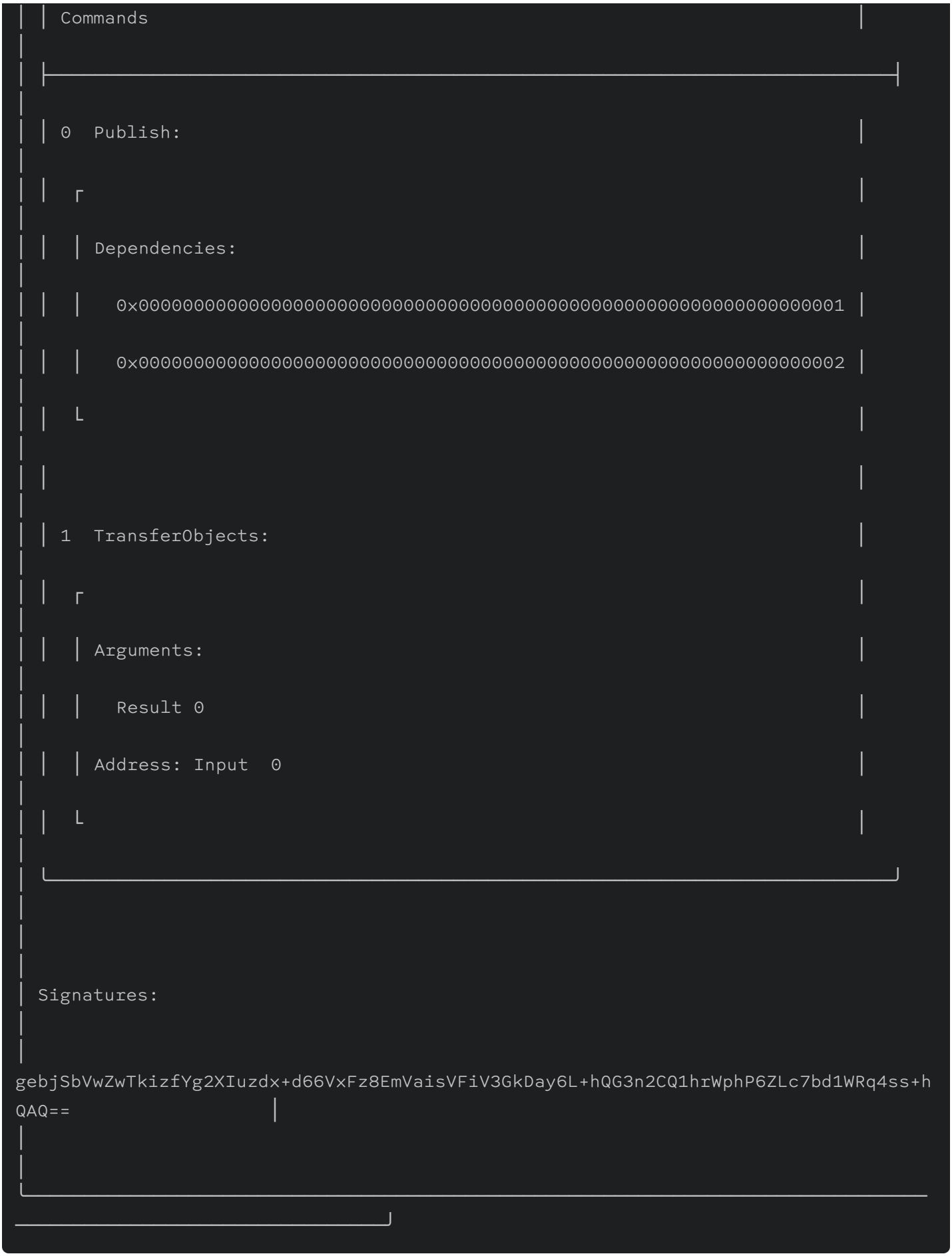
As you can see, when we run the `publish` command, the CLI first builds the package, then verifies the dependencies on-chain, and finally publishes the package. The output of the command is the transaction digest, which is a unique identifier of the transaction and can be used to query the transaction status.

Transaction Data

The section titled `TransactionData` contains the information about the transaction we just sent. It features fields like `sender`, which is your address, the `gas_budget` set with the `--gas-budget` argument, and the Coin we used for payment. It also prints the Commands that were run by the

CLI. In this example, the commands `Publish` and `TransferObject` were run - the latter transfers a special object `UpgradeCap` to the sender.


Transaction Data
Sender: 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1
Gas Owner: 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1
Gas Budget: 100000000 MIST
Gas Price: 1000 MIST
Gas Payment:
ID: 0x4ea1303e4f5e2f65fc3709bc0fb70a3035fdd2d53dbcff33e026a50a742ce0de
Version: 7
Digest: AXYPnups8A5J6pkvLa6RekX2ye3qur66EZ88mEbaUDQ1
Transaction Kind: Programmable
Input Objects
0 Pure Arg: Type: address, Value:
"0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1"


Commands
0 Publish:
Dependencies:
0x0000000000000000000000000000000000000000000000000000000000000001
0x0000000000000000000000000000000000000000000000000000000000000002
1 TransferObjects:
Arguments:
Result 0
Address: Input 0
Signatures:
gebjSbVwZwTkizfYg2XIuzdx+d66VxFz8EmVaisVFiV3GkDay6L+hQG3n2CQ1hrWphP6ZLc7bd1WRq4ss+hQAQ==

Transaction Effects

Transaction Effects contains the status of the transaction, the changes that the transaction made to the state of the network and the objects involved in the transaction.

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┐
│ Transaction Effects
│
├──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┤
│ Digest: GpcDV6JjjGQMRwHpEz582qsd5MpCYgSwrDAq1JXcpFjW
│
│ Status: Success
│
│ Executed Epoch: 411
│
│
│
│ Created Objects:
│
│  ┌──
│
│  │ ID: 0x160f7856e13b27e5a025112f361370f4efc2c2659cb0023f1e99a8a84d1652f3
│
│  │ Owner: Account Address (
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 )  │
│  │ Version: 8
│
│  │ Digest: 8y6bhwvQrGJHDckUZmj2HDAjfkyVqHohhvY1Fvzyj7ec
│
│  └──
│
│  ┌──
│
│  │ ID: 0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe
│
│  │ Owner: Immutable
│
│  │ Version: 1
│
│  │ Digest: Ein91NF2hc3qC4XYoMUFMfin9U23xQmDAdEMSHLae7MK
│
│  └──
│
│ Mutated Objects:
│
│  ┌──
│
│  │ ID: 0x4ea1303e4f5e2f65fc3709bc0fb70a3035fdd2d53dbcff33e026a50a742ce0de
```

```
│
│   │ Owner: Account Address (
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 )  │
│   │ Version: 8
│
│   │ Digest: 7ydahjaM47Gyb33PB4qnW2ZAGqZvDuWScV6sWPiv7LTc
│
│   └──
│
│ Gas Object:
│
│   ┌──
│
│   │ ID: 0x4ea1303e4f5e2f65fc3709bc0fb70a3035fdd2d53dbcff33e026a50a742ce0de
│
│   │ Owner: Account Address (
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 )  │
│   │ Version: 8
│
│   │ Digest: 7ydahjaM47Gyb33PB4qnW2ZAGqZvDuWScV6sWPiv7LTc
│
│   └──
│
│ Gas Cost Summary:
│
│    Storage Cost: 10404400 MIST
│
│    Computation Cost: 1000000 MIST
│
│    Storage Rebate: 978120 MIST
│
│    Non-refundable Storage Fee: 9880 MIST
│
│
│
│ Transaction Dependencies:
│
│    7Ukrc5GqdFqTA41wvWgreCdHn2vRLfgQ3YMFkdks72Vk
│
│    7d4amuHGhjtYKujEs9YkJARzNEn4mRbWWv3fn4cdKdyh
│
└──────────────────────────────────────────────────────────────────────────
──────────────────────┘
```

Events

If there were any *events* emitted, you would see them in this section. Our package does not use events, so the section is empty.

```
┌──────────────────────────┐
│ No transaction block events │
└──────────────────────────┘
```

Object Changes

These are the changes to *objects* that transaction has made. In our case, we have *created* a new `UpgradeCap` object which is a special object that allows the sender to upgrade the package in the future, *mutated* the Gas object, and *published* a new package. Packages are also objects on Sui.

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┐
│ Object Changes
│
├──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────┤
│ Created Objects:
│
│  ┌──
│
│  │ ObjectID: 0x160f7856e13b27e5a025112f361370f4efc2c2659cb0023f1e99a8a84d1652f3
│
│  │ Sender: 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1
│
│  │ Owner: Account Address (
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 ) │
│  │ ObjectType: 0x2::package::UpgradeCap
│
│  │ Version: 8
│
│  │ Digest: 8y6bhwvQrGJHDckUZmj2HDAjfkyVqHohhvY1Fvzyj7ec
│
│  └──
│
│ Mutated Objects:
│
│  ┌──
│
│  │ ObjectID: 0x4ea1303e4f5e2f65fc3709bc0fb70a3035fdd2d53dbcff33e026a50a742ce0de
│
│  │ Sender: 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1
│
│  │ Owner: Account Address (
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 ) │
│  │ ObjectType: 0x2::coin::Coin<0x2::sui::SUI>
│
│  │ Version: 8
│
│  │ Digest: 7ydahjaM47Gyb33PB4qnW2ZAGqZvDuWScV6sWPiv7LTc
│
│  └──
│
│ Published Objects:
│
│  ┌──
```

```
│
│   │ PackageID: 0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe
│
│   │ Version: 1
│
│   │ Digest: Ein91NF2hc3qC4XYoMUFMfin9U23xQmDAdEMSHLae7MK
│
│   │ Modules: todo_list
│
│   └──
│
└──────────────────────────────────────────────────────────────────────────────
──────────────────┘
```

Balance Changes

This last section contains changes to SUI Coins, in our case, we have *spent* around 0.015 SUI, which in MIST is 10,500,000. You can see it under the *amount* field in the output.

```
┌──────────────────────────────────────────────────────────────────────────────
──────────────────┐
│ Balance Changes
│
├──────────────────────────────────────────────────────────────────────────────
──────────────────┤
│   ┌──
│
│   │ Owner: Account Address (
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 )  │
│   │ CoinType: 0x2::sui::SUI
│
│   │ Amount: -10426280
│
│   └──
│
└──────────────────────────────────────────────────────────────────────────────
──────────────────┘
```

Alternative Output

It is possible to specify the `--json` flag during publishing to get the output in JSON format. This is useful if you want to parse the output programmatically or store it for later use.

```
$ sui client publish --gas-budget 100000000 --json
```

Using the Results

After the package is published on chain, we can interact with it. To do this, we need to find the address (object ID) of the package. It's under the `Published Objects` section of the `Object Changes` output. The address is unique for each package, so you will need to copy it from the output.

In this example, the address is:

```
0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe
```

Now that we have the address, we can interact with the package. In the next section, we will show how to interact with the package by sending transactions.

Sending Transactions

To demonstrate the interaction with the `todo_list` package, we will send a transaction to create a new list and add an item to it. Transactions are sent via the `sui client ptb` command, it allows using the Transaction Blocks at full capacity. The command may look big and complex, but we go through it step by step.

Prepare the Variables

Before we construct the command, let's store the values we will use in the transaction. Replace the `0x4....` with the address of the package you have published. And `MY_ADDRESS` variable will be automatically set to your address from the CLI output.

```
$ export
PACKAGE_ID=0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe
$ export MY_ADDRESS=$(sui client active-address)
```

Building the Transaction in CLI

Now to building an actual transaction. The transaction will consist of two parts: we will call the `new` function in the `todo_list` package to create a new list, and then we will transfer the list object to our account. The transaction will look like this:

```
$ sui client ptb \
--gas-budget 100000000 \
--assign sender @$MY_ADDRESS \
--move-call $PACKAGE_ID::todo_list::new \
--assign list \
--transfer-objects "[list]" sender
```

In this command, we are using the `ptb` subcommand to build a transaction. Parameters that follow it define the actual commands and actions that the transaction will perform. The first two calls we make are utility calls to set the sender address to the command inputs and set the gas budget for the transaction.

```
# sets the gas budget for the transaction
--gas-budget 100000000 \n
# registers a variable "sender=@..."
--assign sender @$MY_ADDRESS \n
```

Then we perform the actual call to a function in the package. We use the `--move-call` followed by the package ID, the module name, and the function name. In this case, we are calling the `new` function in the `todo_list` package.

```
# calls the "new" function in the "todo_list" package under the $PACKAGE_ID address
--move-call $PACKAGE_ID::todo_list::new
```

The function that we defined actually returns a value, which we want need to store. We use the `--assign` command to give a name to the returned value. In this case, we are calling it `list`. And then we transfer the object to our account.

```
--move-call $PACKAGE_ID::todo_list::new \
# assigns the result of the "new" function to the "list" variable (from the
previous step)
--assign list \
# transfers the object to the sender
--transfer-objects "[list]" sender
```

Once the command is constructed, you can run it in the terminal. If everything is correct, you should see the output similar to the one we had in previous sections. The output will contain the transaction digest, the transaction data, and the transaction effects.

▶ Spoiler: Full transaction output

The section that we want to focus on is the "Object Changes". More specifically, the "Created Objects" part of it. It contains the object ID, the type and the version of the `TodoList` that you have created. We will use this object ID to interact with the list.

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│ Object Changes                                                                                               │
├──────────────────────────────────────────────────────────────────────────────────────────────────────────────┤
│ Created Objects:                                                                                             │
│  ┌──                                                                                                         │
│  │ ObjectID: 0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553                              │
│  │ Sender: 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1                                │
│  │ Owner: Account Address ( 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 )             │
│  │ ObjectType: 0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::TodoList       │
│  │ Version: 22                                                                                               │
│  │ Digest: HyWdUpjuhjLY38dLpg6KPHQ3bt4BqQAbdF5gB8HQdEqG                                                      │
│  └──                                                                                                         │
│ Mutated Objects:                                                                                             │
│  ┌──                                                                                                         │
│  │ ObjectID: 0xe5ddeb874a8d7ead328e9f2dd2ad8d25383ab40781a5f1aefa75600973b02bc4                              │
│  │ Sender: 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1                                │
│  │ Owner: Account Address ( 0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1 )             │
│  │ ObjectType: 0x2::coin::Coin<0x2::sui::SUI>                                                                │
│  │ Version: 22                                                                                               │
│  │ Digest: DiBrBMshDiD9cThpaEgpcYSF76uV4hCoE1qRyQ3rnYCB                                                      │
│  └──                                                                                                         │
└──────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

```
└──────────────────────────────────────────────────────────────────────────────────────────────────────
──────────────────────────┘
```

In this example the object ID is `0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553`. And the owner should be your account address. We achieved this by transferring the object to the sender in the last command of the transaction.

Another way to test that you have successfully created the list is to check the account objects.

```
$ sui client objects
```

It should have an object that looks similar to this:

```
┌──  ...
┐
│
┌──────────────┬──────────────────────────────────────────────────────────────────────────────────────
─────┐ │
│ │ objectId   │
0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553  │ │
│ │ version    │  22
│ │
│ │ digest     │  /DUEiCLkaNSgzpZSq2vSV0auQQEQhyH9occq9grMBZM=
│ │
│ │ objectType │  0x468d..29fe::todo_list::TodoList
│ │
│
└──────────────┴──────────────────────────────────────────────────────────────────────────────────────
─────┘ │
│  ...
│
```

Passing Objects to Functions

The TodoList that we created in the previous step is an object that you can interact with as its owner. You can call functions defined in the `todo_list` module on this object. To demonstrate this, we will add an item to the list. First, we will add just one item, and in the second transaction we will add 3 and remove another one.

Double check that you have variables set up from the previous step, and then add one more variable for the list object.

```
$ export LIST_ID=0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553
```

Now we can construct the transaction to add an item to the list. The command will look like this:

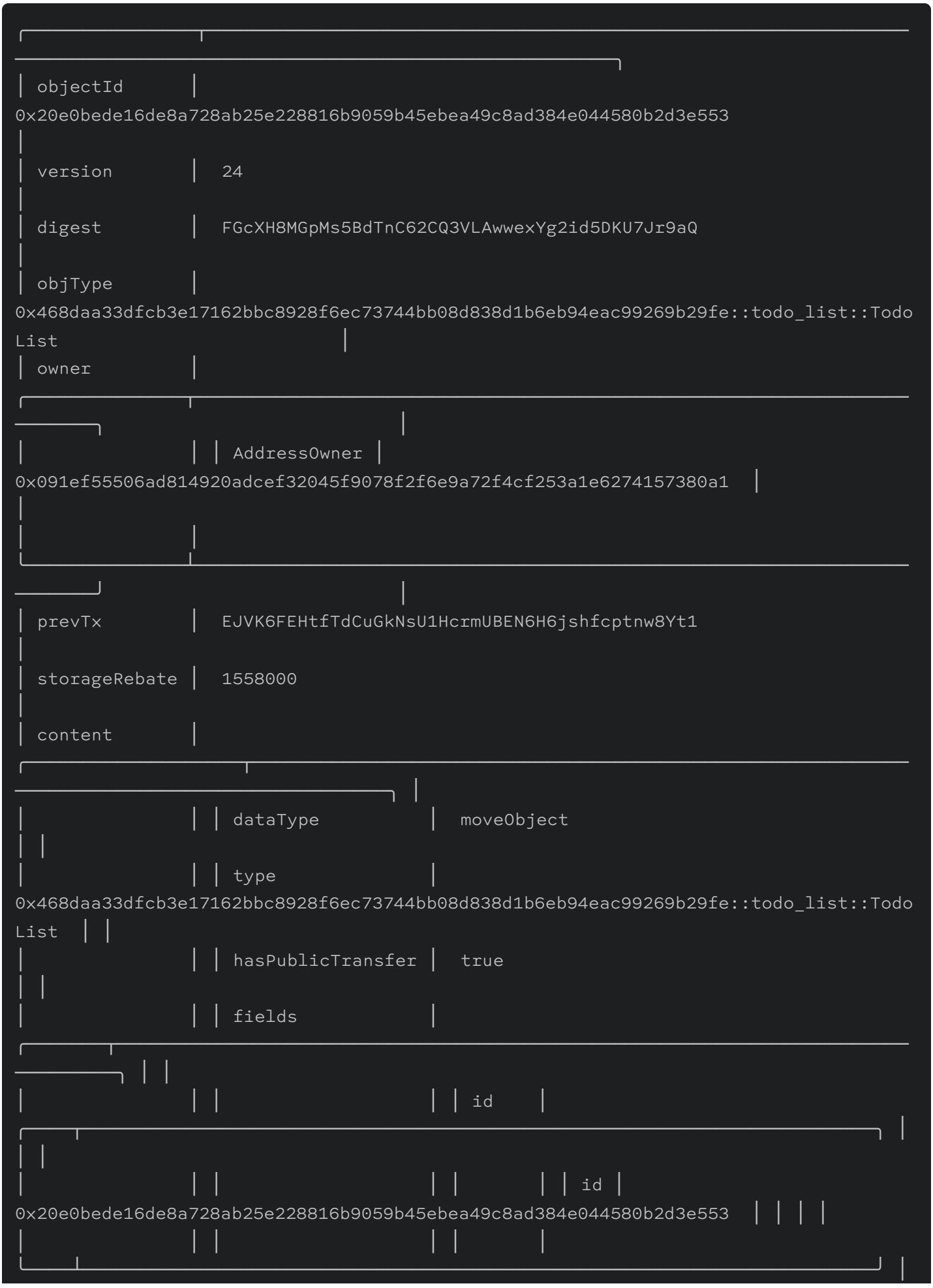
```
$ sui client ptb \
--gas-budget 100000000 \
--move-call $PACKAGE_ID::todo_list::add @$LIST_ID "'Finish the Hello, Sui chapter'"
```

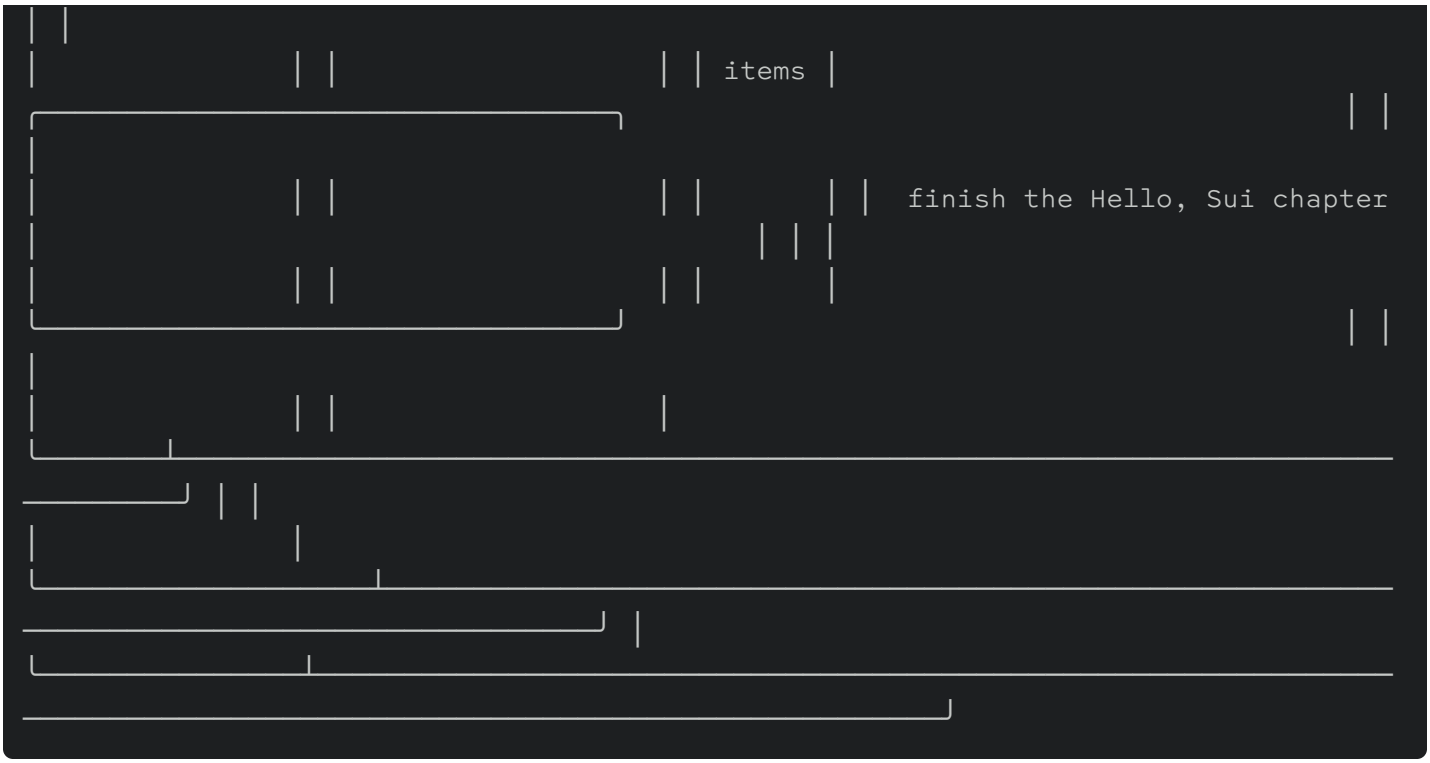
In this command, we are calling the `add` function in the `todo_list` package. The function takes two arguments: the list object and the item to add. The item is a string, so we need to wrap it in single quotes. The command will add the item to the list.

If everything is correct, you should see the output similar to the one we had in previous sections. Now you can check the list object to see if the item was added.

```
$ sui client object $LIST_ID
```

The output should contain the item that you have added.


objectId
0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553
version
24
digest
FGcXH8MGpMs5BdTnC62CQ3VLAwwexYg2id5DKU7Jr9aQ
objType
0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::TodoList
owner
AddressOwner
0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1
prevTx
EJVK6FEHtfTdCuGkNsU1HcrmUBEN6H6jshfcptnw8Yt1
storageRebate
1558000
content
dataType
moveObject
type
0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::TodoList
hasPublicTransfer
true
fields
id
id
0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553



A JSON representation of the object can be obtained by adding the `--json` flag to the command.

```
$ sui client object $LIST_ID --json
```

```
{
  "objectId": "0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553",
  "version": "24",
  "digest": "FGcXH8MGpMs5BdTnC62CQ3VLAwwexYg2id5DKU7Jr9aQ",
  "type":
"0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::Tod
oList",
  "owner": {
    "AddressOwner":
"0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1"
  },
  "previousTransaction": "EJVK6FEHtfTdCuGkNsU1HcrmUBEN6H6jshfcptnw8Yt1",
  "storageRebate": "1558000",
  "content": {
    "dataType": "moveObject",
    "type":
"0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::Tod
oList",
    "hasPublicTransfer": true,
    "fields": {
      "id": {
        "id": "0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553"
      },
      "items": ["Finish the Hello, Sui chapter"]
    }
  }
}
```


Chaining Commands

You can chain multiple commands in a single transaction. This shows the power of Transaction Blocks! Using the same list object, we will add three more items and remove one. The command will look like this:

```
$ sui client ptb \
--gas-budget 100000000 \
--move-call $PACKAGE_ID::todo_list::add @$LIST_ID "'Finish Concepts chapter'" \
--move-call $PACKAGE_ID::todo_list::add @$LIST_ID "'Read the Move Basics chapter'"
\
--move-call $PACKAGE_ID::todo_list::add @$LIST_ID "'Learn about Object Model'" \
--move-call $PACKAGE_ID::todo_list::remove @$LIST_ID 0
```

If previous commands were successful, this one should not be any different. You can check the list object to see if the items were added and removed. The JSON representation is a bit more readable!

```
sui client object $LIST_ID --json
```


```
{
  "objectId": "0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553",
  "version": "25",
  "digest": "EDTXDsteqPGAGu4zFAj5bbQGTkucWk4hhuUquk39enGA",
  "type":
"0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::Tod
oList",
  "owner": {
    "AddressOwner":
"0x091ef55506ad814920adcef32045f9078f2f6e9a72f4cf253a1e6274157380a1"
  },
  "previousTransaction": "7SXLGBSh31jv8G7okQ9mEgnw5MnTfvzzHEHpWf3Sa9gY",
  "storageRebate": "1922800",
  "content": {
    "dataType": "moveObject",
    "type":
"0x468daa33dfcb3e17162bbc8928f6ec73744bb08d838d1b6eb94eac99269b29fe::todo_list::Tod
oList",
    "hasPublicTransfer": true,
    "fields": {
      "id": {
        "id": "0x20e0bede16de8a728ab25e228816b9059b45ebea49c8ad384e044580b2d3e553"
      },
      "items": [
        "Finish Concepts chapter",
        "Read the Move Basics chapter",
        "Learn about Object Model"
      ]
    }
  }
}
```


Commands don't have to be in the same package or operate on the same object. Within a single transaction block, you can interact with multiple packages and objects. This is a powerful feature that allows you to build complex interactions on-chain!

Conclusion

In this guide, we have shown how to publish a package on the Move blockchain and interact with it using the Sui CLI. We have demonstrated how to create a new list object, add items to it, and remove them. We have also shown how to chain multiple commands in a single transaction block. This guide should give you a good starting point for building your own applications on the Sui blockchain!

Concepts

In this chapter you will learn about the basic concepts of Sui and Move. You will learn what is a package, how to interact with it, what is an account and a transaction, and how data is stored on Sui. While this chapter is not a complete reference, and you should refer to the Sui Documentation for that, it will give you a good understanding of the basic concepts required to write Move programs on Sui.

Package

Move is a language for writing smart contracts - programs that are stored and run on the blockchain. A single program is organized into a package. A package is published on the blockchain and is identified by an address. A published package can be interacted with by sending transactions calling its functions. It can also act as a dependency for other packages.

> To create a new package, use the `sui move new` command. To learn more about the command, run `sui move new --help`.

Package consists of modules - separate scopes that contain functions, types, and other items.

```
package 0x...
    module a
        struct A1
        fun hello_world()
    module b
        struct B1
        fun hello_package()
```

Package Structure

Locally, a package is a directory with a `Move.toml` file and a `sources` directory. The `Move.toml` file - called the "package manifest" - contains metadata about the package, and the `sources` directory contains the source code for the modules. Package usually looks like this:

```
sources/
    my_module.move
    another_module.move
    ...
tests/
    ...
examples/
    using_my_module.move
Move.toml
```

The `tests` directory is optional and contains tests for the package. Code placed into the `tests` directory is not published on-chain and is only availably in tests. The `examples` directory can be used for code examples, and is also not published on-chain.

Published Package

During development, package doesn't have an address and it needs to be set to `0x0`. Once a package is published, it gets a single unique address on the blockchain containing its modules' bytecode. A published package becomes *immutable* and can be interacted with by sending transactions.

```
0x...
    my_module: <bytecode>
    another_module: <bytecode>
```

Links

- Package Manifest
- Address
- Packages in the Move Reference.

Package Manifest

The `Move.toml` is a manifest file that describes the package and its dependencies. It is written in TOML format and contains multiple sections, the most important of which are `[package]`, `[dependencies]` and `[addresses]`.

```
[package]
name = "my_project"
version = "0.0.0"
edition = "2024"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir = "crates/sui-framework/packages/sui-framework", rev = "framework/testnet" }

[addresses]
std =  "0x1"
alice = "0xA11CE"

[dev-addresses]
alice = "0xB0B"
```

Sections

Package

The `[package]` section is used to describe the package. None of the fields in this section are published on chain, but they are used in tooling and release management; they also specify the Move edition for the compiler.

- `name` - the name of the package when it is imported;
- `version` - the version of the package, can be used in release management;
- `edition` - the edition of the Move language; currently, the only valid value is `2024`.

Dependencies

The `[dependencies]` section is used to specify the dependencies of the project. Each dependency is specified as a key-value pair, where the key is the name of the dependency, and the

value is the dependency specification. The dependency specification can be a git repository URL or a path to the local directory.

```
# git repository
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir = "crates/sui-framework/packages/sui-framework", rev = "framework/testnet" }

# local directory
MyPackage = { local = "../my-package" }
```

Packages also import addresses from other packages. For example, the Sui dependency adds the `std` and `sui` addresses to the project. These addresses can be used in the code as aliases for the addresses.

Resolving version conflicts with override

Sometimes dependencies have conflicting versions of the same package. For example, if you have two dependencies that use different versions of the Sui package, you can override the dependency in the `[dependencies]` section. To do so, add the `override` field to the dependency. The version of the dependency specified in the `[dependencies]` section will be used instead of the one specified in the dependency itself.

```
[dependencies]
Sui = { override = true, git = "https://github.com/MystenLabs/sui.git", subdir = "crates/sui-framework/packages/sui-framework", rev = "framework/testnet" }
```

Dev-dependencies

It is possible to add `[dev-dependencies]` section to the manifest. It is used to override dependencies in the dev and test modes. For example, if you want to use a different version of the Sui package in the dev mode, you can add a custom dependency specification to the `[dev-dependencies]` section.

Addresses

The `[addresses]` section is used to add aliases for the addresses. Any address can be specified in this section, and then used in the code as an alias. For example, if you add `alice = "0xA11CE"` to this section, you can use `alice` as `0xA11CE` in the code.

Dev-addresses

The `[dev-addresses]` section is the same as `[addresses]`, but only works for the test and dev modes. Important to note that it is impossible to introduce new aliases in this section, only override the existing ones. So in the example above, if you add `alice = "0xB0B"` to this section, the `alice` address will be `0xB0B` in the test and dev modes, and `0xA11CE` in the regular build.

TOML styles

The TOML format supports two styles for tables: inline and multiline. The examples above are using the inline style, but it is also possible to use the multiline style. You wouldn't want to use it for the `[package]` section, but it can be useful for the dependencies.

```
# Inline style
[dependencies]
Sui = { override = true, git = "", subdir = "crates/sui-framework/packages/sui-framework", rev = "framework/testnet" }
MyPackage = { local = "../my-package" }
```

```
# Multiline style
[dependencies.Sui]
override = true
git = "https://github.com/MystenLabs/sui.git"
subdir = "crates/sui-framework/packages/sui-framework"
rev = "framework/testnet"

[dependencies.MyPackage]
local = "../my-package"
```

Further Reading

- Packages in the Move Reference.

Address

Address is a unique identifier of a location on the blockchain. It is used to identify packages, accounts, and objects. Address has a fixed size of 32 bytes and is usually represented as a hexadecimal string prefixed with `0x`. Addresses are case insensitive.

```
0xe51ff5cd221a81c3d6e22b9e670ddf99004d71de4f769b0312b68c7c4872e2f1
```

The address above is an example of a valid address. It is 64 characters long (32 bytes) and prefixed with `0x`.

Sui also has reserved addresses that are used to identify standard packages and objects. Reserved addresses are typically simple values that are easy to remember and type. For example, the address of the Standard Library is `0x1`. Addresses, shorter than 32 bytes, are padded with zeros to the left.

```
0x1 = 0x0000000000000000000000000000000000000000000000000000000000000001
```

Here are some examples of reserved addresses:

- `0x1` - address of the Sui Standard Library (alias `std`)
- `0x2` - address of the Sui Framework (alias `sui`)
- `0x6` - address of the system `Clock` object

> You can find all reserved addresses in the Appendix B.

Further reading

- Address type in Move

Account

An account is a way to identify a user. An account is generated from a private key, and is identified by an address. An account can own objects, and can send transactions. Every transaction has a sender, and the sender is identified by an address.

Sui supports multiple cryptographic algorithms for account generation. The two supported curves are ed25519, secp256k1, and there is also a special way of generating an account - zklogin. The cryptographic agility - the unique feature of Sui - allows for flexibility in the account generation.

Further Reading

- Cryptography in Sui in the Sui Blog
- Keys and Addresses in the Sui Docs
- Signatures in the Sui Docs

Transaction

Transaction is a fundamental concept in the blockchain world. It is a way to interact with a blockchain. Transactions are used to change the state of the blockchain, and they are the only way to do so. In Move, transactions are used to call functions in a package, deploy new packages, and upgrade existing ones.

Transaction Structure

> Every transaction explicitly specifies the objects it operates on!

Transactions consist of:

- a sender - the account that *signs* the transaction;
- a list (or a chain) of commands - the operations to be executed;
- command inputs - the arguments for the commands: either `pure` - simple values like numbers or strings, or `object` - objects that the transaction will access;
- a gas object - the `Coin` object used to pay for the transaction;
- gas price and budget - the cost of the transaction;

Inputs

Transaction inputs are the arguments for the transaction and are split between 2 types:

- Pure arguments: These are mostly primitive types with some extra additions. A pure argument can be:
  - `bool`.
  - unsigned integer (`u8`, `u16`, `u32`, `u64`, `u128`, `u256`).
  - `address`.
  - `std::string::String`, UTF8 strings.
  - `std::ascii::String`, ASCII strings.
  - `vector<T>`, where `T` is a pure type.
  - `std::option::Option<T>`, where `T` is a pure type.
  - `std::object::ID`, typically points to an object. See also What is an Object.
- Object arguments: These are objects or references of objects that the transaction will access. An object argument needs to be either a shared object, a frozen object, or an object that the

transaction sender owns, in order for the transaction to be successfull. For more see Object Model.

Commands

Sui transactions may consist of multiple commands. Each command is a single built-in command (like publishing a package) or a call to a function in an already published package. The commands are executed in the order they are listed in the transaction, and they can use the results of the previous commands, forming a chain. Transaction either succeeds or fails as a whole.

Schematically, a transaction looks like this (in pseudo-code):

```
Inputs:
- sender = 0xa11ce

Commands:
- payment = SplitCoins(Gas, [ 1000 ])
- item = MoveCall(0xAAA::market::purchase, [ payment ])
- TransferObjects(item, sender)
```

In this example, the transaction consists of three commands:

1. `SplitCoins` - a built-in command that splits a new coin from the passed object, in this case, the `Gas` object;
2. `MoveCall` - a command that calls a function `purchase` in a package `0xAAA`, module `market` with the given arguments - the `payment` object;
3. `TransferObjects` - a built-in command that transfers the object to the recipient.

Transaction Effects

Transaction effects are the changes that a transaction makes to the blockchain state. More specifically, a transaction can change the state in the following ways:

- use the gas object to pay for the transaction;
- create, update, or delete objects;
- emit events;

The result of the executed transaction consists of different parts:

- Transaction Digest - the hash of the transaction which is used to identify the transaction;

- Transaction Data - the inputs, commands and gas object used in the transaction;
- Transaction Effects - the status and the "effects" of the transaction, more specifically: the status of the transaction, updates to objects and their new versions, the gas object used, the gas cost of the transaction, and the events emitted by the transaction;
- Events - the custom events emitted by the transaction;
- Object Changes - the changes made to the objects, including the *change of ownership*;
- Balance Changes - the changes made to the aggregate balances of the account involved in the transaction;

Move Basics

This chapter is all about the basic syntax of the Move language. It covers the basics of the language, such as types, modules, functions, and control flow. It focuses on the language without a storage model or a blockchain, and explains the essential concepts of the language. To learn features specific to Sui, such as storage functions and abilities, refer to the Using Objects chapter, however, it is recommended to start with this chapter first.

Module

Module is the base unit of code organization in Move. Modules are used to group and isolate code, and all of the members of the module are private to the module by default. In this section you will learn how to define a module, how to declare its members and how to access them from other modules.

Module declaration

Modules are declared using the `module` keyword followed by the package address, module name and the module body inside the curly braces `{}`. The module name should be in `snake_case` - all lowercase letters with underscores between words. Modules names must be unique in the package.

Usually, a single file in the `sources/` folder contains a single module. The file name should match the module name - for example, a `donut_shop` module should be stored in the `donut_shop.move` file. You can read more about coding conventions in the Coding Conventions section.

```
module book::my_module {
    // module body
}
```

Structs, functions, constants and imports all part of the module:

- Structs
- Functions
- Constants
- Imports
- Struct Methods

Address / Named address

Module address can be specified as both: an address *literal* (does not require the `@` prefix) or a named address specified in the Package Manifest. In the example below, both are identical because there's a `book = "0x0"` record in the `[addresses]` section of the `Move.toml`.

```
module 0x0::address_literal { /* ... */ }
module book::named_address { /* ... */ }
```

Addresses section in the Move.toml:

```
# Move.toml
[addresses]
book = "0x0"
```

Module members

Module members are declared inside the module body. To illustrate that, let's define a simple module with a struct, a function and a constant:

```
module book::my_module_with_members {
    // import
    use book::my_module;

    // a constant
    const CONST: u8 = 0;

    // a struct
    public struct Struct {}

    // method alias
    public use fun function as Struct.struct_fun;

    // function
    fun function(_: &Struct) { /* function body */ }
}
```

Further reading

- Modules in the Move Reference.

Comments

Comments are a way to add notes or document your code. They are ignored by the compiler and don't result in the Move bytecode. You can use comments to explain what your code does, to add notes to yourself or other developers, to temporarily remove a part of your code, or to generate documentation. There are three types of comments in Move: line comment, block comment, and doc comment.

Line comment

```
module book::comments_line {
    fun some_function() {
        // this is a comment line
    }
}
```

You can use double slash `//` to comment out the rest of the line. Everything after `//` will be ignored by the compiler.

```
module book::comments_line_2 {
    // let's add a note to everything!
    fun some_function_with_numbers() {
        let a = 10;
        // let b = 10 this line is commented and won't be executed
        let b = 5; // here comment is placed after code
        a + b; // result is 15, not 10!
    }
}
```

Block comment

Block comments are used to comment out a block of code. They start with `/*` and end with `*/`. Everything between `/*` and `*/` will be ignored by the compiler. You can use block comments to comment out a single line or multiple lines. You can even use them to comment out a part of a line.

```
module book::comments_block {
    fun /* you can comment everywhere */ go_wild() {
        /* here
           there
           everywhere */ let a = 10;
        let b = /* even here */ 10; /* and again */
        a + b;
    }
    /* you can use it to remove certain expressions or definitions
    fun empty_commented_out() {

    }
    */
}
```

This example is a bit extreme, but it shows how you can use block comments to comment out a part of a line.

Doc comment

Documentation comments are special comments that are used to generate documentation for your code. They are similar to block comments, but they start with three slashes `///` and are placed before the definition of the item they document.

```
/// Module has documentation!
module book::comments_doc {

    /// This is a 0x0 address constant!
    const AN_ADDRESS: address = @0x0;

    /// This is a struct!
    public struct AStruct {
        /// This is a field of a struct!
        a_field: u8,
    }

    /// This function does something!
    /// And it's documented!
    fun do_something() {}
}
```

Primitive Types

For simple values, Move has a number of built-in primitive types. They're the base that makes up all other types. The primitive types are:

- Booleans
- Unsigned Integers
- Address - covered in the next section

However, before we get to the types, let's first look at how to declare and assign variables in Move.

Variables and assignment

Variables are declared using the `let` keyword. They are immutable by default, but can be made mutable using the `let mut` keyword. The syntax for the `let mut` statement is:

```
let <variable_name>[: <type>]  = <expression>;
let mut <variable_name>[: <type>] = <expression>;
```

Where:

- `<variable_name>` - the name of the variable
- `<type>` - the type of the variable, optional
- `<expression>` - the value to be assigned to the variable

```
let x: bool = true;
let mut y: u8 = 42;
```

A mutable variable can be reassigned using the `=` operator.

```
y = 43;
```

Variables can also be shadowed by re-declaring.

```
let x: u8 = 42;
let x: u16 = 42;
```

Booleans

The `bool` type represents a boolean value - yes or no, true or false. It has two possible values: `true` and `false` which are keywords in Move. For booleans, there's no need to explicitly specify the type - the compiler can infer it from the value.

```
let x = true;
let y = false;
```

Booleans are often used to store flags and to control the flow of the program. Please, refer to the Control Flow section for more information.

Integer Types

Move supports unsigned integers of various sizes: from 8-bit to 256-bit. The integer types are:

- `u8` - 8-bit
- `u16` - 16-bit
- `u32` - 32-bit
- `u64` - 64-bit
- `u128` - 128-bit
- `u256` - 256-bit

```
let x: u8 = 42;
let y: u16 = 42;
// ...
let z: u256 = 42;
```

Unlike booleans, integer types need to be inferred. In most of the cases, the compiler will infer the type from the value, usually defaulting to `u64`. However, sometimes the compiler is unable to infer the type and will require an explicit type annotation. It can either be provided during assignment or by using a type suffix.

```
// Both are equivalent
let x: u8 = 42;
let x = 42u8;
```

Operations

Move supports the standard arithmetic operations for integers: addition, subtraction, multiplication, division, and remainder. The syntax for these operations is:

Syntax	Operation	Aborts If
+	addition	Result is too large for the integer type
-	subtraction	Result is less than zero
*	multiplication	Result is too large for the integer type
%	modular division	The divisor is 0
/	truncating division	The divisor is 0

> For more operations, including bitwise operations, please refer to the Move Reference.

The type of the operands *must match*, otherwise, the compiler will raise an error. The result of the operation will be of the same type as the operands. To perform operations on different types, the operands need to be cast to the same type.

Casting with `as`

Move supports explicit casting between integer types. The syntax for it is:

```
<expression> as <type>
```

Note, that it may require parentheses around the expression to prevent ambiguity.

```
let x: u8 = 42;
let y: u16 = x as u16;
let z = 2 * (x as u16); // ambiguous, requires parentheses
```

A more complex example, preventing overflow:

```
let x: u8 = 255;
let y: u8 = 255;
let z: u16 = (x as u16) + ((y as u16) * 2);
```

Overflow

Move does not support overflow / underflow, an operation that results in a value outside the range of the type will raise a runtime error. This is a safety feature to prevent unexpected behavior.

```
let x = 255u8;
let y = 1u8;

// This will raise an error
let z = x + y;
```

Further reading

- Bool in the Move Reference.
- Integer in the Move Reference.

Address Type

To represent addresses, Move uses a special type called `address`. It is a 32 byte value that can be used to represent any address on the blockchain. Addresses are used in two syntax forms: hexadecimal addresses prefixed with `0x` and named addresses.

```
// address literal
let value: address = @0x1;

// named address registered in Move.toml
let value = @std;
let other = @sui;
```

An address literal starts with the `@` symbol followed by a hexadecimal number or an identifier. The hexadecimal number is interpreted as a 32 byte value. The identifier is looked up in the Move.toml file and replaced with the corresponding address by the compiler. If the identifier is not found in the Move.toml file, the compiler will throw an error.

Conversion

Sui Framework offers a set of helper functions to work with addresses. Given that the address type is a 32 byte value, it can be converted to a `u256` type and vice versa. It can also be converted to and from a `vector<u8>` type.

Example: Convert an address to a `u256` type and back.

```
use sui::address;

let addr_as_u256: u256 = address::to_u256(@0x1);
let addr = address::from_u256(addr_as_u256);
```

Example: Convert an address to a `vector<u8>` type and back.

```
use sui::address;

let addr_as_u8: vector<u8> = address::to_bytes(@0x1);
let addr = address::from_bytes(addr_as_u8);
```

Example: Convert an address into a string.

```
use sui::address;
use std::string::String;

let addr_as_string: String = address::to_string(@0x1);
```

Further reading

- Address in the Move Reference.

Expression

In programming languages expression is a unit of code which returns a value, in Move, almost everything is an expression, - with the sole exception of `let` statement which is a declaration. In this section, we cover the types of expressions and introduce the concept of scope.

> Expressions are sequenced with semicolons `;`. If there's "no expression" after the semicolon, the compiler will insert a unit `()` - an empty expression.

Literals

In the Primitive Types section, we introduced the basic types of Move. And to illustrate them, we used literals. A literal is a notation for representing a fixed value in the source code. Literals are used to initialize variables and to pass arguments to functions. Move has the following literals:

- `true` and `false` for boolean values
- `0`, `1`, `123123` or other numeric for integer values
- `0x0`, `0x1`, `0x123` or other hexadecimal for integer values
- `b"bytes_vector"` for byte vector values
- `x"0A"` HEX literal for byte values

```
let b = true;     // true is a literal
let n = 1000;     // 1000 is a literal
let h = 0x0A;     // 0x0A is a literal
let v = b"hello"; // b'hello' is a byte vector literal
let x = x"0A";    // x'0A' is a byte vector literal
let c = vector[1, 2, 3]; // vector[] is a vector literal
```

Operators

Arithmetic, logical, and bitwise operators are used to perform operations on values. The result of an operation is a value, so operators are also expressions.

```
let sum = 1 + 2;   // 1 + 2 is an expression
let sum = (1 + 2); // the same expression with parentheses
let is_true = true && false; // true && false is an expression
let is_true = (true && false); // the same expression with parentheses
```

Blocks

A block is a sequence of statements and expressions, and it returns the value of the last expression in the block. A block is written as a pair of curly braces `{}`. A block is an expression, so it can be used anywhere an expression is expected.

```
// block with an empty expression, however, the compiler will
// insert an empty expression automatically: `let none = { () }`
// let none = {};

// block with let statements and an expression.
let sum = {
    let a = 1;
    let b = 2;
    a + b // last expression is the value of the block
};

// block is an expression, so it can be used in an expression and
// doesn't have to be assigned to a variable.
{
    let a = 1;
    let b = 2;
    a + b; // not returned - semicolon.
    // compiler automatically inserts an empty expression `()`
};
```

Function Calls

We go into detail about functions in the Functions section. However, we already used function calls in the previous sections, so it's worth mentioning them here. A function call is an expression that calls a function and returns the value of the last expression in the function body.

```
fun add(a: u8, b: u8): u8 {
    a + b
}

#[test]
fun some_other() {
    let sum = add(1, 2); // add(1, 2) is an expression with type u8
}
```

Control Flow Expressions

Control flow expressions are used to control the flow of the program. They are also expressions, so they return a value. We cover control flow expressions in the Control Flow section. Here's a very brief overview:

```
// if is an expression, so it returns a value; if there are 2 branches,
// the types of the branches must match.
if (bool_expr) expr1 else expr2;

// while is an expression, but it returns `()`.
while (bool_expr) { expr; };

// loop is an expression, but returns `()` as well.
loop { expr; break };
```

Custom Types with Struct

Move's type system shines when it comes to defining custom types. User defined types can be custom tailored to the specific needs of the application. Not just on the data level, but also in its behavior. In this section we introduce the struct definition and how to use it.

Struct

To define a custom type, you can use the `struct` keyword followed by the name of the type. After the name, you can define the fields of the struct. Each field is defined with the `field_name: field_type` syntax. Field definitions must be separated by commas. The fields can be of any type, including other structs.

> Move does not support recursive structs, meaning a struct cannot contain itself as a field.

```
/// A struct representing an artist.
public struct Artist {
    /// The name of the artist.
    name: String,
}

/// A struct representing a music record.
public struct Record {
    /// The title of the record.
    title: String,
    /// The artist of the record. Uses the `Artist` type.
    artist: Artist,
    /// The year the record was released.
    year: u16,
    /// Whether the record is a debut album.
    is_debut: bool,
    /// The edition of the record.
    edition: Option<u16>,
}
```

In the example above, we define a `Record` struct with five fields. The `title` field is of type `String`, the `artist` field is of type `Artist`, the `year` field is of type `u16`, the `is_debut` field is

of type `bool`, and the `edition` field is of type `Option<u16>`. The `edition` field is of type `Option<u16>` to represent that the edition is optional.

Structs are private by default, meaning they cannot be imported and used outside of the module they are defined in. Their fields are also private and can't be accessed from outside the module. See visibility for more information on different visibility modifiers.

> Fields of a struct are private and can only be accessed by the module defining the struct. Reading and writing the fields of a struct in other modules is only possible if the module defining the struct provides public functions to access the fields.

Create and use an instance

We described how struct *definition* works. Now let's see how to initialize a struct and use it. A struct can be initialized using the `struct_name { field1: value1, field2: value2, ... }` syntax. The fields can be initialized in any order, and all of the fields must be set.

```
let mut artist = Artist {
    name: b"The Beatles".to_string()
};
```

In the example above, we create an instance of the `Artist` struct and set the `name` field to a string "The Beatles".

To access the fields of a struct, you can use the `.` operator followed by the field name.

```
// Access the `name` field of the `Artist` struct.
let artist_name = artist.name;

// Access a field of the `Artist` struct.
assert!(artist.name == string::utf8(b"The Beatles"), 0);

// Mutate the `name` field of the `Artist` struct.
artist.name = string::utf8(b"Led Zeppelin");

// Check that the `name` field has been mutated.
assert!(artist.name == string::utf8(b"Led Zeppelin"), 1);
```

Only module defining the struct can access its fields (both mutably and immutably). So the above code should be in the same module as the `Artist` struct.

Unpacking a struct

Structs are non-discardable by default, meaning that the initiated struct value must be used: either stored or *unpacked*. Unpacking a struct means deconstructing it into its fields. This is done using the `let` keyword followed by the struct name and the field names.

```
// Unpack the `Artist` struct and create a new variable `name`
// with the value of the `name` field.
let Artist { name } = artist;
```

In the example above we unpack the `Artist` struct and create a new variable `name` with the value of the `name` field. Because the variable is not used, the compiler will raise a warning. To suppress the warning, you can use the underscore `_` to indicate that the variable is intentionally unused.

```
// Unpack the `Artist` struct and ignore the `name` field.
let Artist { name: _ } = artist;
```

Further reading

- Structs in the Move Reference.

Abilities: Introduction

Move has a unique type system which allows customizing *type abilities*. In the previous section, we introduced the `struct` definition and how to use it. However, the instances of the `Artist` and `Record` structs had to be unpacked for the code to compile. This is default behavior of a struct without *abilities*.

> Throughout the book you will see chapters with name `Ability: <name>`, where `<name>` is the name of the ability. These chapters will cover the ability in detail, how it works, and how to use it in Move.

What are Abilities?

Abilities are a way to allow certain behaviors for a type. They are a part of the struct declaration and define which behaviours are allowed for the instances of the struct.

Abilities syntax

Abilities are set in the struct definition using the `has` keyword followed by a list of abilities. The abilities are separated by commas. Move supports 4 abilities: `copy`, `drop`, `key`, and `store`, each of them is used to define a specific behaviour for the struct instances.

```
/// This struct has the `copy` and `drop` abilities.
struct VeryAble has copy, drop {
    // field: Type1,
    // field2: Type2,
    // ...
}
```

Overview

A quick overview of the abilities:

> All of the built-in types, except references, have `copy`, `drop` and `store` abilities. References have `copy` and `drop`.

- `copy` - allows the struct to be *copied*. Explained in the Ability: Copy chapter.
- `drop` - allows the struct to be *dropped* or *discarded*. Explained in the Ability: Drop chapter.
- `key` - allows the struct to be used as a *key* in a storage. Explained in the Ability: Key chapter.
- `store` - allows the struct to be *stored* in structs with the *key* ability. Explained in the Ability: Store chapter.

While it is important to mention them here, we will go in detail about each ability in the following chapters and give a proper context on how to use them.

No abilities

A struct without abilities cannot be discarded, or copied, or stored in the storage. We call such a struct a *Hot Potato*. It is a joke, but it is also a good way to remember that a struct without abilities is like a hot potato - it can only be passed around and requires special handling. Hot Potato is one of the most powerful patterns in Move, we go in detail about it in the Hot Potato chapter.

Further reading

- Type Abilities in the Move Reference.

Abilities: Drop

The `drop` ability - the simplest of them - allows the instance of a struct to be *ignored* or *discarded*. In many programming languages this behavior is considered default. However, in Move, a struct without the `drop` ability is not allowed to be ignored. This is a safety feature of the Move language, which ensures that all assets are properly handled. An attempt to ignore a struct without the `drop` ability will result in a compilation error.

```
module book::drop_ability {

    /// This struct has the `drop` ability.
    public struct IgnoreMe has drop {
        a: u8,
        b: u8,
    }

    /// This struct does not have the `drop` ability.
    public struct NoDrop {}

    #[test]
    // Create an instance of the `IgnoreMe` struct and ignore it.
    // Even though we constructed the instance, we don't need to unpack it.
    fun test_ignore() {
        let no_drop = NoDrop {};
        let _ = IgnoreMe { a: 1, b: 2 }; // no need to unpack

        // The value must be unpacked for the code to compile.
        let NoDrop {} = no_drop; // OK
    }
}
```

The `drop` ability is often used on custom collection types to eliminate the need for special handling of the collection when it is no longer needed. For example, a `vector` type has the `drop` ability, which allows the vector to be ignored when it is no longer needed. However, the biggest feature of Move's type system is the ability to not have `drop`. This ensures that the assets are properly handled and not ignored.

A struct with a single `drop` ability is called a *Witness*. We explain the concept of a *Witness* in the Witness and Abstract Implementation section.

Types with the `drop` Ability

All native types in Move have the `drop` ability. This includes:

- bool
- unsigned integers
- vector
- address

All of the types defined in the standard library have the `drop` ability as well. This includes:

- Option
- String
- TypeName

Further reading

- Type Abilities in the Move Reference.

Importing Modules

Move achieves high modularity and code reuse by allowing module imports. Modules within the same package can import each other, and a new package can depend on already existing packages and use their modules too. This section will cover the basics of importing modules and how to use them in your own code.

Importing a Module

Modules defined in the same package can import each other. The `use` keyword is followed by the module path, which consists of the package address (or alias) and the module name separated by `::`.

```
// File: sources/module_one.move
module book::module_one {
    /// Struct defined in the same module.
    public struct Character has drop {}

    /// Simple function that creates a new `Character` instance.
    public fun new(): Character { Character {} }
}
```

Another module defined in the same package can import the first module using the `use` keyword.

```
// File: sources/module_two.move
module book::module_two {
    use book::module_one; // importing module_one from the same package

    /// Calls the `new` function from the `module_one` module.
    public fun create_and_ignore() {
        let _ = module_one::new();
    }
}
```

Importing Members

You can also import specific members from a module. This is useful when you only need a single function or a single type from a module. The syntax is the same as for importing a module, but you add the member name after the module path.

```
module book::more_imports {
    use book::module_one::new;       // imports the `new` function from the
`module_one` module
    use book::module_one::Character; // importing the `Character` struct from the
`module_one` module

    /// Calls the `new` function from the `module_one` module.
    public fun create_character(): Character {
        new()
    }
}
```

Grouping Imports

Imports can be grouped into a single `use` statement using the curly braces `{}`. This is useful when you need to import multiple members from the same module. Move allows grouping imports from the same module and from the same package.

```
module book::grouped_imports {
    // imports the `new` function and the `Character` struct from
    /// the `module_one` module
    use book::module_one::{new, Character};

    /// Calls the `new` function from the `module_one` module.
    public fun create_character(): Character {
        new()
    }
}
```

Single function imports are less common in Move, since the function names can overlap and cause confusion. A recommended practice is to import the entire module and use the module path to access the function. Types have unique names and should be imported individually.

To import members and the module itself in the group import, you can use the `Self` keyword. The `Self` keyword refers to the module itself and can be used to import the module and its members.

```
module book::self_imports {
    // imports the `Character` struct, and the `module_one` module
    use book::module_one::{Self, Character};

    /// Calls the `new` function from the `module_one` module.
    public fun create_character(): Character {
        module_one::new()
    }
}
```

Resolving Name Conflicts

When importing multiple members from different modules, it is possible to have name conflicts. For example, if you import two modules that both have a function with the same name, you will need to use the module path to access the function. It is also possible to have modules with the same name in different packages. To resolve the conflict and avoid ambiguity, Move offers the `as` keyword to rename the imported member.

```
module book::conflict_resolution {
    // `as` can be placed after any import, including group imports
    use book::module_one::{Self as mod, Character as Char};

    /// Calls the `new` function from the `module_one` module.
    public fun create(): Char {
        mod::new()
    }
}
```

Adding an External Dependency

Every new package generated via the `sui` binary features a `Move.toml` file with a single dependency on the *Sui Framework* package. The Sui Framework depends on the *Standard Library* package. And both of these packages are available in default configuration. Package dependencies are defined in the Package Manifest as follows:

```
[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir = "crates/sui-framework/packages/sui-framework", rev = "framework/testnet" }
Local = { local = "../my_other_package" }
```

The `dependencies` section contains a list of package dependencies. The key is the name of the package, and the value is either a git import table or a local path. The git import contains the URL of the package, the subdirectory where the package is located, and the revision of the package. The local path is a relative path to the package directory.

If a dependency is added to the `Move.toml` file, the compiler will automatically fetch (and later refetch) the dependencies when building the package.

Importing a Module from Another Package

Normally, packages define their addresses in the `[addresses]` section, so you can use the alias instead of the address. For example, instead of `0x2::coin` module, you would use `sui::coin`. The `sui` alias is defined in the Sui Framework package. Similarly, the `std` alias is defined in the Standard Library package and can be used to access the standard library modules.

To import a module from another package, you use the `use` keyword followed by the module path. The module path consists of the package address (or alias) and the module name separated by `::`.

```
module book::imports {
    use std::string; // std = 0x1, string is a module in the standard library
    use sui::coin;   // sui = 0x2, coin is a module in the Sui Framework
}
```

Standard Library

The Move Standard Library provides functionality for native types and operations. It is a standard collection of modules which do not interact with the storage, but provide basic tools for working and manipulating the data. It is the only dependency of the Sui Framework, and is imported together with it.

Most Common Modules

In this book we go into detail about most of the modules in the Standard Library, however, it is also helpful to give an overview of the features, so that you can get a sense of what is available and which module implements it.

Module	Description	Chapter
std::string	Provides basic string operations	String
std::ascii	Provides basic ASCII operations	String
std::option	Implements an `Option<T>`	Option
std::vector	Native operations on the vector type	Vector
std::bcs	Contains the `bcs::to_bytes()` function	BCS
std::address	Contains a single `address::length` function	Address
std::type_name	Allows runtime *type reflection*	Type Reflection
std::hash	Hashing functions: `sha2_256` and `sha3_256`	Cryptography and Hashing
std::debug	Contains debugging functions, which are available in only in test mode	Debugging
std::bit_vector	Provides operations on bit vectors	-

Module	Description	Chapter
std::fixed_point32	Provides the `FixedPoint32` type	-

Exported Addresses

Standard Library exports one named address - `std = 0x1`.

```
[addresses]
std = "0x1"
```

Implicit Imports

Some of the modules are imported implicitly, and are available in the module without explicit `use` import. For Standard Library, these modules and types are:

- std::vector
- std::option
- std::option::Option

Importing std without Sui Framework

The Move Standard Library can be imported to the package directly. However, `std` alone is not enough to build a meaningful application, as it does not provide any storage capabilities, and can't interact with the on-chain state.

```
MoveStdlib = { git = "https://github.com/MystenLabs/sui.git", subdir = "crates/sui-framework/packages/move-stdlib", rev = "framework/mainnet" }
```

Source Code

The source code of the Move Standard Library is available in the Sui repository.

Vector

Vectors are a native way to store collections of elements in Move. They are similar to arrays in other programming languages, but with a few differences. In this section, we introduce the `vector` type and its operations.

Vector syntax

The `vector` type is defined using the `vector` keyword followed by the type of the elements in angle brackets. The type of the elements can be any valid Move type, including other vectors. Move has a vector literal syntax that allows you to create vectors using the `vector` keyword followed by square brackets containing the elements (or no elements for an empty vector).

```
// An empty vector of bool elements.
let empty: vector<bool> = vector[];

// A vector of u8 elements.
let v: vector<u8> = vector[10, 20, 30];

// A vector of vector<u8> elements.
let vv: vector<vector<u8>> = vector[
    vector[10, 20],
    vector[30, 40]
];
```

The `vector` type is a built-in type in Move, and does not need to be imported from a module. However, vector operations are defined in the `std::vector` module, and you need to import the module to use them.

Vector operations

The standard library provides methods to manipulate vectors. The following are some of the most commonly used operations:

- `push_back`: Adds an element to the end of the vector.
- `pop_back`: Removes the last element from the vector.
- `length`: Returns the number of elements in the vector.
- `is_empty`: Returns true if the vector is empty.

- `remove`: Removes an element at a given index.

```
let mut v = vector[10u8, 20, 30];

assert!(v.length() == 3, 0);
assert!(!v.is_empty(), 1);

v.push_back(40);
let last_value = v.pop_back();

assert!(last_value == 40, 2);
```

Destroying a Vector of non-droppable types

A vector of non-droppable types cannot be discarded. If you define a vector of types without `drop` ability, the vector value cannot be ignored. However, if the vector is empty, compiler requires an explicit call to `destroy_empty` function.

```
    /// A struct without `drop` ability.
    public struct NoDrop {}

    #[test]
    fun test_destroy_empty() {
        // Initialize a vector of `NoDrop` elements.
        let v = vector<NoDrop>[];

        // While we know that `v` is empty, we still need to call
        // the explicit `destroy_empty` function to discard the vector.
        v.destroy_empty();
    }
```

Further reading

- Vector in the Move Reference.

Option

Option is a type that represents an optional value which may or may not exist. The concept of Option in Move is borrowed from Rust, and it is a very useful primitive in Move. `Option` is defined in the Standard Library, and is defined as follows:

File: move-stdlib/source/option.move

```
// File: move-stdlib/source/option.move
/// Abstraction of a value that may or may not be present.
struct Option<Element> has copy, drop, store {
    vec: vector<Element>
}
```

> The 'std::option' module is implicitly imported in every module, and you don't need to add an import.

The `Option` is a generic type which takes a type parameter `Element`. It has a single field `vec` which is a `vector` of `Element`. Vector can have length 0 or 1, and this is used to represent the presence or absence of a value.

Option type has two variants: `Some` and `None`. `Some` variant contains a value and `None` variant represents the absence of a value. The `Option` type is used to represent the absence of a value in a type-safe way, and it is used to avoid the need for empty or `undefined` values.

In Practice

To showcase why Option type is necessary, let's look at an example. Consider an application which takes a user input and stores it in a variable. Some fields are required, and some are optional. For example, a user's middle name is optional. While we could use an empty string to represent the absence of a middle name, it would require extra checks to differentiate between an empty string and a missing middle name. Instead, we can use the `Option` type to represent the middle name.

```
module book::user_registry {
    use std::string::String;

    /// A struct representing a user record.
    public struct User has drop {
        first_name: String,
        middle_name: Option<String>,
        last_name: String,
    }

    /// Create a new `User` struct with the given fields.
    public fun register(
        first_name: String,
        middle_name: Option<String>,
        last_name: String,
    ): User {
        User { first_name, middle_name, last_name }
    }
}
```

In the example above, the `middle_name` field is of type `Option<String>`. This means that the `middle_name` field can either contain a `String` value or be empty. This makes it clear that the middle name is optional, and it avoids the need for extra checks to differentiate between an empty string and a missing middle name.

Using Option

To use the `Option` type, you need to import the `std::option` module and use the `Option` type. You can then create an `Option` value using the `some` or `none` methods.

```
// `option::some` creates an `Option` value with a value.
let mut opt = option::some(b"Alice");

// `option.is_some()` returns true if option contains a value.
assert!(opt.is_some(), 1);

// internal value can be `borrow`ed and `borrow_mut`ed.
assert!(opt.borrow() == &b"Alice", 0);

// `option.extract` takes the value out of the option, leaving the option empty.
let inner = opt.extract();

// `option.is_none()` returns true if option is None.
assert!(opt.is_none(), 2);
```

String

While Move does not have a built-in type to represent strings, it does have two standard implementations for strings in the Standard Library. The `std::string` module defines a `String` type and methods for UTF-8 encoded strings, and the second module, `std::ascii`, provides an ASCII `String` type and its methods.

> Sui execution environment automatically converts bytevector into `String` in transaction inputs. So in many cases, a String does not need to be constructed in the Transaction Block.

Strings are bytes

No matter which type of string you use, it is important to know that strings are just bytes. The wrappers provided by the `string` and `ascii` modules are just that: wrappers. They do provide safety checks and methods to work with strings, but at the end of the day, they are just vectors of bytes.

```
module book::custom_string {
    /// Anyone can implement a custom string-like type by wrapping a vector.
    public struct MyString {
        bytes: vector<u8>,
    }

    /// Implement a `from_bytes` function to convert a vector of bytes to a string.
    public fun from_bytes(bytes: vector<u8>): MyString {
        MyString { bytes }
    }

    /// Implement a `bytes` function to convert a string to a vector of bytes.
    public fun bytes(self: &MyString): &vector<u8> {
        &self.bytes
    }
}
```

Working with UTF-8 Strings

While there are two types of strings in the standard library, the `string` module should be considered the default. It has native implementations of many common operations, and hence is more efficient than the `ascii` module, which is fully implemented in Move.

Definition

The `String` type in the `std::string` module is defined as follows:

```
// File: move-stdlib/sources/string.move
/// A `String` holds a sequence of bytes which is guaranteed to be in utf8 format.
public struct String has copy, drop, store {
    bytes: vector<u8>,
}
```

Creating a String

To create a new UTF-8 `String` instance, you can use the `string::utf8` method. The Standard Library provides an alias `.to_string()` on the `vector<u8>` for convenience.

```
// the module is `std::string` and the type is `String`
use std::string::{Self, String};

// strings are normally created using the `utf8` function
// type declaration is not necessary, we put it here for clarity
let hello: String = string::utf8(b"Hello");

// The `.to_string()` alias on the `vector<u8>` is more convenient
let hello = b"Hello".to_string();
```

Common Operations

UTF8 String provides a number of methods to work with strings. The most common operations on strings are: concatenation, slicing, and getting the length. Additionally, for custom string operations, the `bytes()` method can be used to get the underlying byte vector.

```
let mut str = b"Hello,".to_string();
let another = b" World!".to_string();

// append(String) adds the content to the end of the string
str.append(another);

// `sub_string(start, end)` copies a slice of the string
str.sub_string(0, 5); // "Hello"

// `length()` returns the number of bytes in the string
str.length(); // 12 (bytes)

// methods can also be chained! Get a length of a substring
str.sub_string(0, 5).length(); // 5 (bytes)

// whether the string is empty
str.is_empty(); // false

// get the underlying byte vector for custom operations
let bytes: &vector<u8> = str.bytes();
```

Safe UTF-8 Operations

The default `utf8` method may abort if the bytes passed into it are not valid UTF-8. If you are not sure that the bytes you are passing are valid, you should use the `try_utf8` method instead. It returns an `Option<String>`, which contains no value if the bytes are not valid UTF-8, and a string otherwise.

> Hint: the name that starts with `try_*` indicates that the function returns an Option with the expected result or `none` if the operation fails. It is a common naming convention borrowed from Rust.

```
// this is a valid UTF-8 string
let hello = b"Hello".try_to_string();

assert!(hello.is_some(), 0); // abort if the value is not valid UTF-8

// this is not a valid UTF-8 string
let invalid = b"\xFF".try_to_string();

assert!(invalid.is_none(), 0); // abort if the value is valid UTF-8
```

UTF-8 Limitations

The `string` module does not provide a way to access individual characters in a string. This is because UTF-8 is a variable-length encoding, and the length of a character can be anywhere from 1 to 4 bytes. Similarly, the `length()` method returns the number of bytes in the string, not the number of characters.

However, methods like `sub_string` and `insert` check character boundaries and will abort when the index is in the middle of a character.

ASCII Strings

This section is coming soon!

Control Flow

Control flow statements are used to control the flow of execution in a program. They are used to make decisions, to repeat a block of code, and to exit a block of code early. Move has the following control flow statements (explained in detail below):

- `if` and `if-else` - making decisions on whether to execute a block of code
- `loop` and `while` loops - repeating a block of code
- `break` and `continue` statements - exiting a loop early
- `return` statement - exiting a function early

Conditional Statements

The `if` expression is used to make decisions in a program. It evaluates a boolean expression and executes a block of code if the expression is true. Paired with `else`, it can execute a different block of code if the expression is false.

The syntax for the `if` expression is:

```
if (<bool_expression>) <expression>;
if (<bool_expression>) <expression> else <expression>;
```

Just like any other expression, `if` requires a semicolon, if there are other expressions following it. The `else` keyword is optional, except for the case when the resulting value is assigned to a variable. We will cover this below.

```
    #[test]
    fun test_if() {
        let x = 5;

        // `x > 0` is a boolean expression.
        if (x > 0) {
            std::debug::print(&b"X is bigger than 0".to_string())
        };
    }
```

Let's see how we can use `if` and `else` to assign a value to a variable:

```
    #[test]
    fun test_if_else() {
        let x = 5;
        let y = if (x > 0) {
            1
        } else {
            0
        };

        assert!(y == 1, 0);
    }
```

Here we assign the value of the `if` expression to the variable `y`. If `x` is greater than 0, `y` will be assigned the value 1, otherwise 0. The `else` block is necessary, because both branches must return a value of the same type. If we omit the `else` block, the compiler will throw an error.

Conditional expressions are one of the most important control flow statements in Move. They can use either user provided input or some already stored data to make decisions. In particular, they are used in the `assert!` macro to check if a condition is true, and if not, to abort execution. We will get to it very soon!

Repeating Statements with Loops

Loops are used to execute a block of code multiple times. Move has two built-in types of loops: `loop` and `while`. In many cases they can be used interchangeably, but usually `while` is used when the number of iterations is known in advance, and `loop` is used when the number of iterations is not known in advance or there are multiple exit points.

Loops are helpful when dealing with collections, such as vectors, or when we want to repeat a block of code until a certain condition is met. However, it is important to be careful with loops, as they can lead to infinite loops, which can lead to gas exhaustion and the transaction being aborted.

The `while` loop

The `while` statement is used to execute a block of code as long as a boolean expression is true. Just like we've seen with `if`, the boolean expression is evaluated before each iteration of the loop. Just like conditional statements, the `while` loop is an expression and requires a semicolon if there are other expressions following it.

The syntax for the `while` loop is:

```
while (<bool_expression>) { <expressions>; };
```

Here is an example of a `while` loop with a very simple condition:

```
    // This function iterates over the `x` variable until it reaches 10, the
    // return value is the number of iterations it took to reach 10.
    //
    // If `x` is 0, then the function will return 10.
    // If `x` is 5, then the function will return 5.
    fun while_loop(mut x: u8): u8 {
        let mut y = 0;

        // This will loop until `x` is 10.
        // And will never run if `x` is 10 or more.
        while (x < 10) {
            y = y + 1;
            x = x + 1;
        };

        y
    }

    #[test]
    fun test_while() {
        assert!(while_loop(0) == 10, 0); // 10 times
        assert!(while_loop(5) == 5, 0);  // 5 times
        assert!(while_loop(10) == 0, 0); // loop never executed
    }
```

Infinite `loop`

Now let's imagine a scenario where the boolean expression is always `true`. For example, if we literally passed `true` to the `while` condition. As you might expect, this would create an infinite loop, and this is almost what the `loop` statement works like.

```
    #[test, expected_failure(out_of_gas, location=Self)]
    fun test_infinite_while() {
        let mut x = 0;

        // This will loop forever.
        while (true) {
            x = x + 1;
        };

        // This line will never be executed.
        assert!(x == 5, 0);
    }
```

An infinite `while`, or `while` without a condition, is a `loop`. The syntax for it is simple:

```
loop { <expressions>; };
```

Let's rewrite the previous example using `loop` instead of `while`:

```
    #[test, expected_failure(out_of_gas, location=Self)]
    fun test_infinite_loop() {
        let mut x = 0;

        // This will loop forever.
        loop {
            x = x + 1;
        };

        // This line will never be executed.
        assert!(x == 5, 0);
    }
```

Infinite loops on their own are not very useful in Move, since every operation in Move costs gas, and an infinite loop will lead to gas exhaustion. However, they can be used in combination with `break` and `continue` statements to create more complex loops.

Exiting a Loop Early

As we already mentioned, infinite loops are rather useless on their own. And that's where we introduce the `break` and `continue` statements. They are used to exit a loop early, and to skip the rest of the current iteration, respectively.

Syntax for the `break` statement is (without a semicolon):

```
break
```

The `break` statement is used to stop the execution of a loop and exit it early. It is often used in combination with a conditional statement to exit the loop when a certain condition is met. To illustrate this point, let's turn the infinite `loop` from the previous example into something that looks and behaves more like a `while` loop:

```
    #[test]
    fun test_break_loop() {
        let mut x = 0;

        // This will loop until `x` is 5.
        loop {
            x = x + 1;

            // If `x` is 5, then exit the loop.
            if (x == 5) {
                break // Exit the loop.
            }
        };

        assert!(x == 5, 0);
    }
```

Almost identical to the `while` loop, right? The `break` statement is used to exit the loop when `x` is 5. If we remove the `break` statement, the loop will run forever, just like the previous example.

Skipping an Iteration

The `continue` statement is used to skip the rest of the current iteration and start the next one. Similarly to `break`, it is used in combination with a conditional statement to skip the rest of the iteration when a certain condition is met.

Syntax for the `continue` statement is (without a semicolon):

```
continue
```

The example below skips odd numbers and prints only even numbers from 0 to 10:

```
    #[test]
    fun test_continue_loop() {
        let mut x = 0;

        // This will loop until `x` is 10.
        loop {
            x = x + 1;

            // If `x` is odd, then skip the rest of the iteration.
            if (x % 2 == 1) {
                continue // Skip the rest of the iteration.
            };

            std::debug::print(&x);

            // If `x` is 10, then exit the loop.
            if (x == 10) {
                break // Exit the loop.
            }
        };

        assert!(x == 10, 0); // 10
    }
```

`break` and `continue` statements can be used in both `while` and `loop` loops.

Early Return

The `return` statement is used to exit a function early and return a value. It is often used in combination with a conditional statement to exit the function when a certain condition is met. The syntax for the `return` statement is:

```
return <expression>
```

Here is an example of a function that returns a value when a certain condition is met:

```
    /// This function returns `true` if `x` is greater than 0 and not 5,
    /// otherwise it returns `false`.
    fun is_positive(x: u8): bool {
        if (x == 5) {
            return false
        };

        if (x > 0) {
            return true
        };

        false
    }

    #[test]
    fun test_return() {
        assert!(is_positive(5) == false, 0);
        assert!(is_positive(0) == false, 0);
        assert!(is_positive(1) == true, 0);
    }
```

Unlike in other languages, the `return` statement is not required for the last expression in a function. The last expression in a function block is automatically returned. However, the `return` statement is useful when we want to exit a function early if a certain condition is met.

Constants

Constants are immutable values that are defined at the module level. They often serve as a way to give names to static values that are used throughout a module. For example, if there's a default price for a product, you might define a constant for it. Constants are stored in the module's bytecode, and each time they are used, the value is copied.

```
module book::shop_price {
    use sui::coin::Coin;
    use sui::sui::SUI;

    /// The price of an item in the shop.
    const ITEM_PRICE: u64 = 100;
    /// The owner of the shop, an address.
    const SHOP_OWNER: address = @0xa11ce;

    /// An item sold in the shop.
    public struct Item { /* ... */ }

    /// Purchase an item from the shop.
    public fun purchase(coin: Coin<SUI>): Item {
        assert!(coin.value() == ITEM_PRICE, 0);

        transfer::public_transfer(coin, SHOP_OWNER);

        Item { /* ... */ }
    }
}
```

Naming Convention

Constants must start with a capital letter - this is enforced at the compiler level. For constants used as a value, there's a convention to use uppercase letters and underscores to separate words. It's a way to make constants stand out from other identifiers in the code. One exception is made for error constants, which are written in ECamelCase.

```
/// Price of the item used at the shop.
const ITEM_PRICE: u64 = 100;

/// Error constant.
const EItemNotFound: u64 = 1;
```

Constants are Immutable

Constants can't be changed and assigned new values. They are part of the package bytecode, and inherently immutable.

```
module book::immutable_constants {
    const ITEM_PRICE: u64 = 100;

    // emits an error
    fun change_price() {
        ITEM_PRICE = 200;
    }
}
```

Using Config Pattern

A common use case for an application is to define a set of constants that are used throughout the codebase. But due to constants being private to the module, they can't be accessed from other modules. One way to solve this is to define a "config" module that exports the constants.

```
module book::config {
    const ITEM_PRICE: u64 = 100;
    const TAX_RATE: u64 = 10;
    const SHIPPING_COST: u64 = 5;

    /// Returns the price of an item.
    public fun item_price(): u64 { ITEM_PRICE }
    /// Returns the tax rate.
    public fun tax_rate(): u64 { TAX_RATE }
    /// Returns the shipping cost.
    public fun shipping_cost(): u64 { SHIPPING_COST }
}
```

This way other modules can import and read the constants, and the update process is simplified. If the constants need to be changed, only the config module needs to be updated during the package upgrade.

Links

- Constants in the Move Reference
- Coding conventions for constants

Aborting Execution

A transaction can either succeed or fail. Successful execution applies all the changes made to objects and on-chain data, and the transaction is committed to the blockchain. Alternatively, if a transaction aborts, the changes are not applied. The `abort` keyword is used to abort a transaction and revert the changes made so far.

> It is important to note that there is no catch mechanism in Move. If a transaction aborts, the changes made so far are reverted, and the transaction is considered failed.

Abort

The `abort` keyword is used to abort the execution of a transaction. It is used in combination with an abort code, which will be returned to the caller of the transaction. The abort code is an integer of type `u64`.

```
let user_has_access = true;

// abort with a predefined constant if `user_has_access` is false
if (!user_has_access) {
    abort 0
};

// there's an alternative syntax using parenthesis`
if (user_has_access) {
   abort(1)
};
```

The code above will, of course, abort with abort code `1`.

assert!

The `assert!` macro is a built-in macro that can be used to assert a condition. If the condition is false, the transaction will abort with the given abort code. The `assert!` macro is a convenient way to abort a transaction if a condition is not met. The macro shortens the code otherwise written with an `if` expression + `abort`. The `code` argument is required and has to be a `u64` value.

```
// aborts if `user_has_access` is `false` with abort code 0
assert!(user_has_access, 0);

// expands into:
if (!user_has_access) {
    abort 0
};
```

Error constants

To make error codes more descriptive, it is a good practice to define error constants. Error constants are defined as `const` declarations and are usually prefixed with `E` followed by a camel case name. Error constants are no different from other constants and don't have special handling, however, they are used to increase the readability of the code and make it easier to understand the abort scenarios.

```
/// Error code for when the user has no access.
const ENoAccess: u64 = 0;
/// Trying to access a field that does not exist.
const ENoField: u64 = 1;

/// Updates a record.
public fun update_record(/* ... , */ user_has_access: bool, field_exists: bool) {
    // asserts are way more readable now
    assert!(user_has_access, ENoAccess);
    assert!(field_exists, ENoField);

    /* ... */
}
```

Further reading

- Abort and Assert in the Move Reference.
- We suggest reading the Better Error Handling guide to learn about best practices for error handling in Move.

Function

Functions are the building blocks of Move programs. They are called from user transactions and from other functions and group executable code into reusable units. Functions can take arguments and return a value. They are declared with the `fun` keyword at the module level. Just like any other module member, by default they're private and can only be accessed from within the module.

```
module book::math {
    /// Function takes two arguments of type `u64` and returns their sum.
    /// The `public` visibility modifier makes the function accessible from
    /// outside the module.
    public fun add(a: u64, b: u64): u64 {
        a + b
    }

    #[test]
    fun test_add() {
        let sum = add(1, 2);
        assert!(sum == 3, 0);
    }
}
```

In this example, we define a function `add` that takes two arguments of type `u64` and returns their sum. The function is called from the `test_add` function, which is a test function located in the same module. In the test we compare the result of the `add` function with the expected value and abort the execution if the result is different.

Function declaration

> There's a convention to call functions in Move with the `snake_case` naming convention. This means that the function name should be all lowercase with words separated by underscores. For example, `do_something`, `add`, `get_balance`, `is_authorized`, and so on.

A function is declared with the `fun` keyword followed by the function name (a valid Move identifier), a list of arguments in parentheses, and a return type. The function body is a block of

code that contains a sequence of statements and expressions. The last expression in the function body is the return value of the function.

```
fun return_nothing() {
    // empty expression, function returns `()`
}
```

Accessing functions

Just like any other module member, functions can be imported and accessed via a path. The path consists of the module path and the function name separated by `::`. For example, if you have a function called `add` in the `math` module in the `book` package, the path to it will be `book::math::add`, or, if the module is imported, `math::add`.

```
module book::use_math {
    use book::math;

    fun call_add() {
        // function is called via the path
        let sum = math::add(1, 2);
    }
}
```

Multiple return values

Move functions can return multiple values, which is useful when you need to return more than one value from a function. The return type of the function is a tuple of types. The return value is a tuple of expressions.

```
fun get_name_and_age(): (vector<u8>, u8) {
    (b"John", 25)
}
```

Result of a function call with tuple return has to be unpacked into variables via `let (tuple)` syntax:

```
// Tuple must be destructured to access its elements.
// Name and age are declared as immutable variables.
let (name, age) = get_name_and_age();
assert!(name == b"John", 0);
assert!(age == 25, 0);
```

If any of the declared values need to be declared as mutable, the `mut` keyword is placed before the variable name:

```
// declare name as mutable, age as immutable
let (mut name, age) = get_name_and_age();
```

If some of the arguments are not used, they can be ignored with the `_` symbol:

```
// ignore the name, only use the age
let (_, age) = get_name_and_age();
```

Further reading

- Functions in the Move Reference.

Struct Methods

Move Compiler supports *receiver syntax*, which allows defining methods which can be called on instances of a struct. This is similar to the method syntax in other programming languages. It is a convenient way to define functions which operate on the fields of a struct.

Method syntax

If the first argument of a function is a struct internal to the module, then the function can be called using the `.` operator. If the function uses a struct from another module, then method won't be associated with the struct by default. In this case, the function can be called using the standard function call syntax.

When a module is imported, the methods are automatically associated with the struct.

```
module book::hero {
    /// A struct representing a hero.
    public struct Hero has drop {
        health: u8,
        mana: u8,
    }

    /// Create a new Hero.
    public fun new(): Hero { Hero { health: 100, mana: 100 } }

    /// A method which casts a spell, consuming mana.
    public fun heal_spell(hero: &mut Hero) {
        hero.health = hero.health + 10;
        hero.mana = hero.mana - 10;
    }

    /// A method which returns the health of the hero.
    public fun health(hero: &Hero): u8 { hero.health }

    /// A method which returns the mana of the hero.
    public fun mana(hero: &Hero): u8 { hero.mana }

    #[test]
    // Test the methods of the `Hero` struct.
    fun test_methods() {
        let mut hero = new();
        hero.heal_spell();

        assert!(hero.health() == 110, 1);
        assert!(hero.mana() == 90, 2);
    }
}
```

Method Aliases

For modules that define multiple structs and their methods, it is possible to define method aliases to avoid name conflicts, or to provide a better-named method for a struct.

The syntax for aliases is:

```
// for local method association
use fun function_path as Type.method_name;

// exported alias
public use fun function_path as Type.method_name;
```

> Public aliases are only allowed for structs defined in the same module. If a struct is defined in another module, an alias can still be created but cannot be made public.

In the example below, we changed the `hero` module and added another type - `Villain`. Both `Hero` and `Villain` have similar field names and methods. And to avoid name conflicts, we prefixed methods with `hero_` and `villain_` respectively. However, we can create aliases for these methods so that they can be called on the instances of the structs without the prefix.

```
module book::hero_and_villain {
    /// A struct representing a hero.
    public struct Hero has drop {
        health: u8,
    }

    /// A struct representing a villain.
    public struct Villain has drop {
        health: u8,
    }

    /// Create a new Hero.
    public fun new_hero(): Hero { Hero { health: 100 } }

    /// Create a new Villain.
    public fun new_villain(): Villain { Villain { health: 100 } }

    // Alias for the `hero_health` method. Will be imported automatically when
    // the module is imported.
    public use fun hero_health as Hero.health;

    public fun hero_health(hero: &Hero): u8 { hero.health }

    // Alias for the `villain_health` method. Will be imported automatically
    // when the module is imported.
    public use fun villain_health as Villain.health;

    public fun villain_health(villain: &Villain): u8 { villain.health }

    #[test]
    // Test the methods of the `Hero` and `Villain` structs.
    fun test_associated_methods() {
        let hero = new_hero();
        assert!(hero.health() == 100, 1);

        let villain = new_villain();
        assert!(villain.health() == 100, 3);
    }
}
```

As you can see, in the test function, we called the `health` method on the instances of `Hero` and `Villain` without the prefix. The compiler will automatically associate the methods with the structs.

Aliasing an external module's method

It is also possible to associate a function defined in another module with a struct from the current module. Following the same approach, we can create an alias for the method defined in another module. Let's use the `bcs::to_bytes` method from the Standard Library and associate it with the `Hero` struct. It will allow serializing the `Hero` struct to a vector of bytes.

```
// TODO: better example (external module...)
module book::hero_to_bytes {
    // Alias for the `bcs::to_bytes` method. Imported aliases should be defined
    // in the top of the module.
    // public use fun bcs::to_bytes as Hero.to_bytes;

    /// A struct representing a hero.
    public struct Hero has drop {
        health: u8,
        mana: u8,
    }

    /// Create a new Hero.
    public fun new(): Hero { Hero { health: 100, mana: 100 } }

    #[test]
    // Test the methods of the `Hero` struct.
    fun test_hero_serialize() {
        // let mut hero = new();
        // let serialized = hero.to_bytes();
        // assert!(serialized.length() == 3, 1);
    }
}
```

Further reading

- Method Syntax in the Move Reference.

Visibility Modifiers

Every module member has a visibility. By default, all module members are *private* - meaning they are only accessible within the module they are defined in. However, you can add a visibility modifier to make a module member *public* - visible outside the module, or *public(package)* - visible in the modules within the same package, or *entry* - can be called from a transaction but can't be called from other modules.

Internal Visibility

A function or a struct defined in a module which has no visibility modifier is *private* to the module. It can't be called from other modules.

```
module book::internal_visibility {
    // This function can be called from other functions in the same module
    fun internal() { /* ... */ }

    // Same module -> can call internal()
    fun call_internal() {
        internal();
    }
}
```

```
module book::try_calling_internal {
    use book::internal_visibility;

    // Different module -> can't call internal()
    fun try_calling_internal() {
        internal_visibility::internal();
    }
}
```

Public Visibility

A struct or a function can be made *public* by adding the `public` keyword before the `fun` or `struct` keyword.

```
module book::public_visibility {
    // This function can be called from other modules
    public fun public() { /* ... */ }
}
```

A public function can be imported and called from other modules. The following code will compile:

```
module book::try_calling_public {
    use book::public_visibility;

    // Different module -> can call public()
    fun try_calling_public() {
        public_visibility::public();
    }
}
```

Package Visibility

Move 2024 introduces the *package visibility* modifier. A function with *package visibility* can be called from any module within the same package. It can't be called from other packages.

```
module book::package_visibility {
    public(package) fun package_only() { /* ... */ }
}
```

A package function can be called from any module within the same package:

```
module book::try_calling_package {
    use book::package_visibility;

    // Same package `book` -> can call package_only()
    fun try_calling_package() {
        package_visibility::package_only();
    }
}
```

Ownership and Scope

Every variable in Move has a scope and an owner. The scope is the range of code where the variable is valid, and the owner is the scope that this variable belongs to. Once the owner scope ends, the variable is dropped. This is a fundamental concept in Move, and it is important to understand how it works.

Ownership

A variable defined in a function scope is owned by this scope. The runtime goes through the function scope and executes every expression and statement. Once the function scope end, the variables defined in it are dropped or deallocated.

```
module book::ownership {
    public fun owner() {
        let a = 1; // a is owned by the `owner` function
    } // a is dropped here

    public fun other() {
        let b = 2; // b is owned by the `other` function
    } // b is dropped here

    #[test]
    fun test_owner() {
        owner();
        other();
        // a & b is not valid here
    }
}
```

In the example above, the variable `a` is owned by the `owner` function, and the variable `b` is owned by the `other` function. When each of these functions are called, the variables are defined, and when the function ends, the variables are discarded.

Returning a Value

If we changed the `owner` function to return the variable `a`, then the ownership of `a` would be transferred to the caller of the function.

```
module book::ownership {
    public fun owner(): u8 {
        let a = 1; // a defined here
        a // scope ends, a is returned
    }

    #[test]
    fun test_owner() {
        let a = owner();
        // a is valid here
    } // a is dropped here
}
```

Passing by Value

Additionally, if we passed the variable `a` to another function, the ownership of `a` would be transferred to this function. When performing this operation, we *move* the value from one scope to another. This is also called *move semantics*.

```
module book::ownership {
    public fun owner(): u8 {
        let a = 10;
        a
    } // a is returned

    public fun take_ownership(v: u8) {
        // v is owned by `take_ownership`
    } // v is dropped here

    #[test]
    fun test_owner() {
        let a = owner();
        take_ownership(a);
        // a is not valid here
    }
}
```

Scopes with Blocks

Each function has a main scope, and it can also have sub-scopes via the use of blocks. A block is a sequence of statements and expressions, and it has its own scope. Variables defined in a block are owned by this block, and when the block ends, the variables are dropped.

```
module book::ownership {
    public fun owner() {
        let a = 1; // a is owned by the `owner` function's scope
        {
            let b = 2; // b is owned by the block
            {
                let c = 3; // c is owned by the block
            }; // c is dropped here
        }; // b is dropped here
        // a = b; // error: b is not valid here
        // a = c; // error: c is not valid here
    } // a is dropped here
}
```

However, shall we use the return value of a block, the ownership of the variable is transferred to the caller of the block.

```
module book::ownership {
    public fun owner(): u8 {
        let a = 1; // a is owned by the `owner` function's scope
        let b = {
            let c = 2; // c is owned by the block
            c // c is returned
        }; // c is dropped here
        a + b // both a and b are valid here
    }
}
```

Copyable Types

Some types in Move are *copyable*, which means that they can be copied without transferring the ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. Move compiler will automatically copy these types when they are passed to a function or returned from a function, or when they're *moved* to a scope and then accessed in their original scope.

Further reading

- Local Variables and Scopes in the Move Reference.

Abilities: Copy

In Move, the *copy* ability on a type indicates that the instance or the value of the type can be copied. While this behavior may feel very natural when working with numbers or other simple types, it is not the default for custom types in Move. This is because Move is designed to express digital assets and resources, and inability to copy is a key element of the resource model.

However, Move type system allows you to define custom types with the *copy* ability.

```
public struct Copyable has copy {}
```

In the example above, we define a custom type `Copyable` with the *copy* ability. This means that instances of `Copyable` can be copied, both implicitly and explicitly.

```
let a = Copyable {};
let b = a;   // `a` is copied to `b`
let c = *&b; // explicit copy via dereference operator

let Copyable {} = a; // doesn't have `drop`
let Copyable {} = b; // doesn't have `drop`
let Copyable {} = c; // doesn't have `drop`
```

In the example above, `a` is copied to `b` implicitly, and then explicitly copied to `c` using the dereference operator. If `Copyable` did not have the *copy* ability, the code would not compile, and the Move compiler would raise an error.

Copying and Drop

The `copy` ability is closely related to `drop` ability. If a type has the *copy* ability, very likely that it should have `drop` too. This is because the *drop* ability is required to clean up the resources when the instance is no longer needed. If a type has only *copy*, then managing its instances gets more complicated, as the values cannot be ignored.

```
public struct Value has copy, drop {}
```

All of the primitive types in Move behave as if they have the *copy* and *drop* abilities. This means that they can be copied and dropped, and the Move compiler will handle the memory management for them.

Types with the `copy` Ability

All native types in Move have the `copy` ability. This includes:

- bool
- unsigned integers
- vector
- address

All of the types defined in the standard library have the `copy` ability as well. This includes:

- Option
- String
- TypeName

Further reading

- Type Abilities in the Move Reference.

References

In the Ownership and Scope section, we explained that when a value is passed to a function, it is *moved* to the function's scope. This means that the function becomes the owner of the value, and the original scope (owner) can no longer use it. This is an important concept in Move, as it ensures that the value is not used in multiple places at the same time. However, there are use cases when we want to pass a value to a function but retain the ownership. This is where references come into play.

To illustrate this, let's consider a simple example - an application for a metro (subway) pass. We will look at 4 different scenarios:

1. Card can be purchased at the kiosk for a fixed price
2. Card can be shown to inspectors to prove that the passenger has a valid pass
3. Card can be used at the turnstile to enter the metro, and spend a ride
4. Card can be recycled once it's empty

Layout

The initial layout of the metro pass application is simple. We define the `Card` type and the `USES` constant that represents the number of rides for a single card. We also add an error constant for the case when the card is empty.

```
module book::metro_pass {
    /// Error code for when the card is empty.
    const ENoUses: u64 = 0;

    /// Number of uses for a metro pass card.
    const USES: u8 = 3;

    /// A metro pass card
    public struct Card { uses: u8 }

    /// Purchase a metro pass card.
    public fun purchase(/* pass a Coin */): Card {
        Card { uses: USES }
    }
}
```

Reference

References are a way to *show* a value to a function without giving up the ownership. In our case, when we show the Card to the inspector, we don't want to give up the ownership of it, and we don't allow them to spend the rides. We just want to allow *reading* the value of the Card and prove its ownership.

To do so, in the function signature, we use the `&` symbol to indicate that we are passing a reference to the value, not the value itself.

```
/// Show the metro pass card to the inspector.
public fun is_valid(card: &Card): bool {
    card.uses > 0
}
```

Now the function can't take the ownership of the card, and it can't spend the rides. But it can read its value. Worth noting, that a signature like this makes it impossible to call the function without a Card at all. This is an important property which allows the Capability Pattern which we will cover in the next chapters.

Mutable Reference

In some cases, we want to allow the function to change the value of the Card. For example, when we use the Card at the turnstile, we want to spend a ride. To implement it, we use the `&mut` keyword in the function signature.

```
/// Use the metro pass card at the turnstile to enter the metro.
public fun enter_metro(card: &mut Card) {
    assert!(card.uses > 0, ENoUses);
    card.uses = card.uses - 1;
}
```

As you can see in the function body, the `&mut` reference allows mutating the value, and the function can spend the rides.

Passing by Value

Lastly, let's give an illustration of what happens when we pass the value itself to the function. In this case, the function takes the ownership of the value, and the original scope can no longer use

it. The owner of the Card can recycle it, and, hence, lose the ownership.

```
    /// Recycle the metro pass card.
    public fun recycle(card: Card) {
        assert!(card.uses == 0, ENoUses);
        let Card { uses: _ } = card;
    }
```

In the `recycle` function, the Card is *taken by value* and can be unpacked and destroyed. The original scope can't use it anymore.

Full Example

To illustrate the full flow of the application, let's put all the pieces together in a test.

```
    #[test]
    fun test_card_2024() {
        // declaring variable as mutable because we modify it
        let mut card = purchase();

        card.enter_metro(); // modify the card but don't move it
        assert!(card.is_valid(), 0); // read the card!

        card.enter_metro(); // modify the card but don't move it
        card.enter_metro(); // modify the card but don't move it

        card.recycle(); // move the card out of the scope
    }
```

Generics

Generics are a way to define a type or function that can work with any type. This is useful when you want to write a function which can be used with different types, or when you want to define a type that can hold any other type. Generics are the foundation of many advanced features in Move, such as collections, abstract implementations, and more.

In the Standard Library

In this chapter we already mentioned the vector type, which is a generic type that can hold any other type. Another example of a generic type in the standard library is the Option type, which is used to represent a value that may or may not be present.

Generic Syntax

To define a generic type or function, a type signature needs to have a list of generic parameters enclosed in angle brackets (`<` and `>`). The generic parameters are separated by commas.

```
/// Container for any type `T`.
public struct Container<T> has drop {
    value: T,
}

/// Function that creates a new `Container` with a generic value `T`.
public fun new<T>(value: T): Container<T> {
    Container { value }
}
```

In the example above, `Container` is a generic type with a single type parameter `T`, the `value` field of the container stores the `T`. The `new` function is a generic function with a single type parameter `T`, and it returns a `Container` with the given value. Generic types must be initialed with a concrete type, and generic functions must be called with a concrete type.

```
#[test]
fun test_container() {
    // these three lines are equivalent
    let container: Container<u8> = new(10); // type inference
    let container = new<u8>(10); // create a new `Container` with a `u8` value
    let container = new(10u8);

    assert!(container.value == 10, 0x0);

    // Value can be ignored only if it has the `drop` ability.
    let Container { value: _ } = container;
}
```

In the test function `test_generic` we demonstrate three equivalent ways to create a new `Container` with a `u8` value. Because numeric types need to be inferred, we specify the type of the number literal.

Multiple Type Parameters

You can define a type or function with multiple type parameters. The type parameters are then separated by commas.

```
/// A pair of values of any type `T` and `U`.
public struct Pair<T, U> {
    first: T,
    second: U,
}

/// Function that creates a new `Pair` with two generic values `T` and `U`.
public fun new_pair<T, U>(first: T, second: U): Pair<T, U> {
    Pair { first, second }
}
```

In the example above, `Pair` is a generic type with two type parameters `T` and `U`, and the `new_pair` function is a generic function with two type parameters `T` and `U`. The function returns a `Pair` with the given values. The order of the type parameters is important, and it should match the order of the type parameters in the type signature.

```
#[test]
fun test_generic() {
    // these three lines are equivalent
    let pair_1: Pair<u8, bool> = new_pair(10, true); // type inference
    let pair_2 = new_pair<u8, bool>(10, true); // create a new `Pair` with a `u8`
and `bool` values
    let pair_3 = new_pair(10u8, true);

    assert!(pair_1.first == 10, 0x0);
    assert!(pair_1.second, 0x0);

    // Unpacking is identical.
    let Pair { first: _, second: _ } = pair_1;
    let Pair { first: _, second: _ } = pair_2;
    let Pair { first: _, second: _ } = pair_3;

}
```

If we added another instance where we swapped type parameters in the `new_pair` function, and tried to compare two types, we'd see that the type signatures are different, and cannot be compared.

```
#[test]
fun test_swap_type_params() {
    let pair1: Pair<u8, bool> = new_pair(10u8, true);
    let pair2: Pair<bool, u8> = new_pair(true, 10u8);

    // this line will not compile
    // assert!(pair1 == pair2, 0x0);

    let Pair { first: pf1, second: ps1 } = pair1; // first1: u8, second1: bool
    let Pair { first: pf2, second: ps2 } = pair2; // first2: bool, second2: u8

    assert!(pf1 == ps2, 0x0); // 10 == 10
    assert!(ps1 == pf2, 0x0); // true == true
}
```

Types for variables `pair1` and `pair2` are different, and the comparison will not compile.

Why Generics?

In the examples above we focused on instantiating generic types and calling generic functions to create instances of these types. However, the real power of generics is the ability to define shared behavior for the base, generic type, and then use it independently of the concrete types. This is especially useful when working with collections, abstract implementations, and other advanced features in Move.

```
/// A user record with name, age, and some generic metadata
public struct User<T> {
    name: String,
    age: u8,
    /// Varies depending on application.
    metadata: T,
}
```

In the example above, `User` is a generic type with a single type parameter `T`, with shared fields `name` and `age`, and the generic `metadata` field which can store any type. No matter what the `metadata` is, all of the instances of `User` will have the same fields and methods.

```
/// Updates the name of the user.
public fun update_name<T>(user: &mut User<T>, name: String) {
    user.name = name;
}

/// Updates the age of the user.
public fun update_age<T>(user: &mut User<T>, age: u8) {
    user.age = age;
}
```

Phantom Type Parameters

In some cases, you may want to define a generic type with a type parameter that is not used in the fields or methods of the type. This is called a *phantom type parameter*. Phantom type parameters are useful when you want to define a type that can hold any other type, but you want to enforce some constraints on the type parameter.

```
/// A generic type with a phantom type parameter.
public struct Coin<phantom T> {
    value: u64
}
```

The `Coin` type here does not contain any fields or methods that use the type parameter `T`. It is used to differentiate between different types of coins, and to enforce some constraints on the type parameter `T`.

```
public struct USD {}
public struct EUR {}

#[test]
fun test_phantom_type() {
    let coin1: Coin<USD> = Coin { value: 10 };
    let coin2: Coin<EUR> = Coin { value: 20 };

    // Unpacking is identical because the phantom type parameter is not used.
    let Coin { value: _ } = coin1;
    let Coin { value: _ } = coin2;
}
```

In the example above, we demonstrate how to create two different instances of `Coin` with different phantom type parameters `USD` and `EUR`. The type parameter `T` is not used in the fields or methods of the `Coin` type, but it is used to differentiate between different types of coins. It will make sure that the `USD` and `EUR` coins are not mixed up.

Constraints on Type Parameters

Type parameters can be constrained to have certain abilities. This is useful when you need the inner type to allow certain behavior, such as *copy* or *drop*. The syntax for constraining a type parameter is `T: <ability> + <ability>`.

```
/// A generic type with a type parameter that has the `drop` ability.
public struct Droppable<T: drop> {
    value: T,
}

/// A generic struct with a type parameter that has the `copy` and `drop`
abilities.
public struct CopyableDroppable<T: copy + drop> {
    value: T, // T must have the `copy` and `drop` abilities
}
```

Move Compiler will enforce that the type parameter `T` has the specified abilities. If the type parameter does not have the specified abilities, the code will not compile.

```
/// Type without any abilities.
public struct NoAbilities {}

#[test]
fun test_constraints() {
    // Fails - `NoAbilities` does not have the `drop` ability
    // let droppable = Droppable<NoAbilities> { value: 10 };

    // Fails - `NoAbilities` does not have the `copy` and `drop` abilities
    // let copyable_droppable = CopyableDroppable<NoAbilities> { value: 10 };
}
```

Further Reading

- Generics in the Move Reference.

Type Reflection

In programming languages *reflection* is the ability of a program to examine and modify its own structure and behavior. In Move, there's a limited form of reflection that allows you to inspect the type of a value at runtime. This is useful when you need to store type information in a homogeneous collection, or when you need to check if a type belongs to a package.

Type reflection is implemented in the Standard Library module `std::type_name`. Expressed very roughly, it gives a single function `get<T>()` which returns the name of the type `T`.

In practice

The module is pretty straightforward, and operations allowed on the result are limited to getting a string representation and extracting the module and address of the type.

```
module book::type_reflection {
    use std::ascii::String;
    use std::type_name::{Self, TypeName};

    /// A function that returns the name of the type `T` and its module and
address.
    public fun do_i_know_you<T>(): (String, String, String) {
        let type_name: TypeName = type_name::get<T>();

        // there's a way to borrow
        let str: &String = type_name.borrow_string();

        let module_name: String = type_name.get_module();
        let address_str: String = type_name.get_address();

        // and a way to consume the value
        let str = type_name.into_string();

        (str, module_name, address_str)
    }

    #[test_only]
    public struct MyType {}

    #[test]
    fun test_type_reflection() {
        let (type_name, module_name, _address_str) = do_i_know_you<MyType>();

        //
        assert!(module_name == b"type_reflection".to_ascii_string(), 1);
    }
}
```

Further reading

Type reflection is an important part of the language, and it is a crucial part of some of the more advanced patterns.

Testing

A crucial part of any software development, and even more - blockchain development, is testing. Here, we will cover the basics of testing in Move and how to write and organize tests for your Move code.

The `#[test]` attribute

Tests in Move are functions marked with the `#[test]` attribute. This attribute tells the compiler that the function is a test function, and it should be run when the tests are executed. Test functions are regular functions, but they must take no arguments and have no return value. They are excluded from the bytecode, and are never published.

```
module book::testing {
    // Test attribute is placed before the `fun` keyword. Can be both above or
    // right before the `fun` keyword: `#[test] fun my_test() { ... }`
    // The name of the test would be `book::testing::simple_test`.
    #[test]
    fun simple_test() {
        let sum = 2 + 2;
        assert!(sum == 4, 1);
    }

    // The name of the test would be `book::testing::more_advanced_test`.
    #[test] fun more_advanced_test() {
        let sum = 2 + 2 + 2;
        assert!(sum == 4, 1);
    }
}
```

Running Tests

To run tests, you can use the `sui move test` command. This command will first build the package in the *test mode* and then run all the tests found in the package. During test mode, modules from both `sources/` and `tests/` directories are processed, and the tests are executed.

```
$ sui move test
> INCLUDING DEPENDENCY Sui
> INCLUDING DEPENDENCY MoveStdlib
> BUILDING book
> Running Move unit tests
> ...
```

Test Fail Cases with `#[expected_failure]`

Tests for fail cases can be marked with `#[expected_failure]`. This attribute placed on a `#[test]` function tells the compiler that the test is expected to fail. This is useful when you want to test that a function fails when a certain condition is met.

> This attribute can only be placed on a `#[test]` function.

The attribute can take an argument for abort code, which is the expected abort code when the test fails. If the test fails with a different abort code, the test will fail. If the execution did not abort, the test will also fail.

```
module book::testing_failure {

    const EInvalidArgument: u64 = 1;

    #[test]
    #[expected_failure(abort_code = 0)]
    fun test_fail() {
        abort 0 // aborts with 0
    }

    // attributes can be grouped together
    #[test, expected_failure(abort_code = EInvalidArgument)]
    fun test_fail_1() {
        abort 1 // aborts with 1
    }
}
```

The `abort_code` argument can use constants defined in the tests module as well as imported from other modules. This is the only case where constants can be used and "accessed" in other modules.

Utilities with `#[test_only]`

In some cases, it is helpful to give the test environment access to some of the internal functions or features. It simplifies the testing process and allows for more thorough testing. However, it is important to remember that these functions should not be included in the final package. This is where the `#[test_only]` attribute comes in handy.

```
module book::testing {
    // Public function which uses the `secret` function.
    public fun multiply_by_secret(x: u64): u64 {
        x * secret()
    }

    /// Private function which is not available to the public.
    fun secret(): u64 { 100 }

    #[test_only]
    /// This function is only available for testing purposes in tests and other
    /// test-only functions. Mind the visibility - for `#[test_only]` it is
    /// common to use `public` visibility.
    public fun secret_for_testing(): u64 {
        secret()
    }

    #[test]
    // In the test environment we have access to the `secret_for_testing` function.
    fun test_multiply_by_secret() {
        let expected = secret_for_testing() * 2;
        assert!(multiply_by_secret(2) == expected, 1);
    }
}
```

Functions marked with the `#[test_only]` will be available to the test environment, and to the other modules if their visibility is set so.

Further Reading

- Unit Testing in the Move Reference.

Object Model

This chapter describes the Object Model of Sui. It focuses on the theory and concepts behind the Object Model, preparing you for a practical dive into Sui Storage operations and resource ownership. For convenience and easier lookup, we split the chapter into several sections, each covering a specific aspect of the Object Model.

> ⓘ In no way should this chapter be considered a comprehensive guide to the Object Model. It is only a high-level overview of the concepts and principles behind the Object Model.
>
> For a more detailed description, refer to the Sui Documentation.

Move - Language for Digital Assets

Smart-contract programming languages have historically focused on defining and managing digital assets. For example, the ERC-20 standard in Ethereum pioneered a set of standards to interact with digital currency tokens, establishing a blueprint for creating and managing digital currencies on the blockchain. Subsequently, the introduction of the ERC-721 standard marked a significant evolution, popularising the concept of non-fungible tokens (NFTs), which represent unique, indivisible assets. These standards laid the groundwork for the complex digital assets we see today.

However, Ethereum's programming model lacked a native representation of assets. In other words, externally, a Smart Contract behaved like an asset, but the language itself did not have a way to inherently represent assets. From the start, Move aimed to provide a first-class abstraction for assets, opening up new avenues for thinking about and programming assets.

It is important to highlight which properties are essential for an asset:

- Ownership: Every asset is associated with an owner(s), mirroring the straightforward concept of ownership in the physical world—just as you own a car, you can own a digital asset. Move enforces ownership in such a way that once an asset is *moved*, the previous owner completely loses any control over it. This mechanism ensures a clear and secure change of ownership.

- Non-copyable: In the real world, unique items cannot be duplicated effortlessly. Move applies this principle to digital assets, ensuring they cannot be arbitrarily copied within the program. This property is crucial for maintaining the scarcity and uniqueness of digital assets, mirroring the intrinsic value of physical assets.

- Non-discardable: Just as you cannot accidentally lose a house or a car without a trace, Move ensures that no asset can be discarded or lost in a program. Instead, assets must be explicitly transferred or destroyed. This property guarantees the deliberate handling of digital assets, preventing accidental loss and ensuring accountability in asset management.

Move managed to encapsulate these properties in its design, becoming an ideal language for digital assets.

Summary

- Move was designed to provide a first-class abstraction for digital assets, enabling developers to create and manage assets natively.

- Essential properties of digital assets include ownership, non-copyability, and non-discardability, which Move enforces in its design.
- Move's asset model mirrors real-world asset management, ensuring secure and accountable asset ownership and transfer.

Further reading

- Move: A Language With Programmable Resources (pdf) by Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, Runtian Zhou*

Evolution of Move

While Move was created to manage digital assets, its initial storage model was bulky and not well-suited for many use cases. For instance, if Alice wanted to transfer an asset X to Bob, Bob had to create a new "empty" resource, and then Alice could transfer asset X to Bob. This process was not intuitive and presented implementation challenges, partly due to the restrictive design of Diem. Another drawback of the original design was the lack of built-in support for a "transfer" operation, requiring every module to implement its own storage transfer logic. Additionally, managing heterogeneous collections of assets in a single account was particularly challenging.

Sui addressed these challenges by redesigning the storage and ownership model of objects to more closely resemble real-world object interactions. With a native concept of ownership and *transfer*, Alice can directly transfer asset X to Bob. Furthermore, Bob can maintain a collection of different assets without any preparatory steps. These improvements laid the foundation for the Object Model in Sui.

Summary

- Move's initial storage model was not well-suited for managing digital assets, requiring complex and restrictive transfer operations.
- Sui introduced the Object Model, which provides a native concept of ownership, simplifying asset management and enabling heterogeneous collections.

Further reading

- Why We Created Sui Move by Sam Blackshear

What is an Object?

The Object Model in Sui can be viewed as a high-level abstraction representing digital assets as *objects*. These objects have their own type and associated behaviors, a unique identifier, and support native storage operations like *transfer* and *share*. Designed to be intuitive and easy to use, the Object Model enables a wide range of use cases to be implemented with ease.

Objects in Sui have the following properties:

- Type: Every object has a type, defining the structure and behavior of the object. Objects of different types cannot be mixed or used interchangeably, ensuring objects are used correctly according to their type system.
- Unique ID: Each object has a unique identifier, distinguishing it from other objects. This ID is generated upon the object's creation and is immutable. It's used to track and identify objects within the system.
- Owner: Every object is associated with an owner, who has control over the object. Ownership on Sui can be exclusive to an account, shared across the network, or frozen, allowing read-only access without modification or transfer capabilities. We will discuss ownership in more detail in the following sections.
- Data: Objects encapsulate their data, simplifying management and manipulation. The data structure and operations are defined by the object's type.
- Version: The transition from accounts to objects is facilitated by object versioning. Traditionally, blockchains use a *nonce* to prevent replay attacks. In Sui, the object's version acts as a nonce, preventing replay attacks for each object.
- Digest: Every object has a digest, which is a hash of the object's data. The digest is used to cryptographically verify the integrity of the object's data and ensures that it has not been tampered with. The digest is calculated when the object is created and is updated whenever the object's data changes.

Summary

- Objects in Sui are high-level abstractions representing digital assets.
- Objects have a type, unique ID, owner, data, version, and digest.
- The Object Model simplifies asset management and enables a wide range of use cases.

Further reading

- Object Model in Sui Documentation.

Ownership

Sui introduces four distinct ownership types for objects: single owner, shared state, immutable shared state, and object-owner. Each model offers unique characteristics and suits different use cases, enhancing flexibility and control in object management.

Account Owner (or Single Owner)

The account owner, also known as the *single owner* model, is the foundational ownership type in Sui. Here, an object is owned by a single account, granting that account exclusive control over the object within the behaviors associated with its type. This model embodies the concept of *true ownership*, where the account possesses complete authority over the object, making it inaccessible to others for modification or transfer. This level of ownership clarity is a significant advantage over other blockchain systems, where ownership definitions can be more ambiguous, and smart contracts may have the ability to alter or transfer assets without the owner's consent.

Shared State

Single owner model has its limitations: for example, it is very tricky to implement a marketplace for digital assets without a shared state. For a generic marketplace scenario, imagine that Alice owns an asset X, and she wants to sell it by putting it into a shared marketplace. Then Bob can come and buy the asset directly from the marketplace. The reason why this is tricky is that it is impossible to write a smart contract that would "lock" the asset in Alice's account and take it out when Bob buys it. First, it will be a violation of the single owner model, and second, it requires a shared access to the asset.

To facilitate a problem of shared data access, Sui has introduced a shared ownership model. In this model, an object can be shared with the network. Shared objects can be read and modified by any account on the network, and the rules of interaction are defined by the implementation of the object. Typical uses for shared objects are: marketplaces, shared resources, escrows, and other scenarios where multiple accounts need access to the same state.

Immutable (Frozen) State

Sui also offers the *frozen object* model, where an object becomes permanently read-only. These immutable objects, while readable, cannot be modified or moved, providing a stable and constant

state accessible to all network participants. Frozen objects are ideal for public data, reference materials, and other use cases where the state permanence is desirable.

Object Owner

The last ownership model in Sui is the *object owner*. In this model, an object is owned by another object. This feature allows creating complex relationships between objects, store large heterogenious collections, and implementing extensible and modular systems. Practically speaking, since the transactions are initiated by accounts, the transaction still accesses the parent object, but it can then access the child objects through the parent object.

A use case we love to mention is a game character. Alice can own the Hero object from a game, and the Hero can own items: also represented as objects, like a "Map", or a "Compass". Alice may take the "Map" from the "Hero" object, and then send it to Bob, or sell it on a marketplace. With object owner, it becomes very natural to imagine how the assets can be structured and managed in relation to each other.

Summary

- Single Owner: Objects are owned by a single account, granting exclusive control over the object.
- Shared State: Objects can be shared with the network, allowing multiple accounts to read and modify the object.
- Immutable State: Objects become permanently read-only, providing a stable and constant state.
- Object Owner: Objects can own other objects, enabling complex relationships and modular systems.

Next Steps

In the next section we will talk about transaction execution paths in Sui, and how the ownership models affect the transaction execution.

Fast Path & Consensus

The Object Model allows for variable transaction execution paths, depending on the object's ownership type. The transaction execution path determines how the transaction is processed and validated by the network. In this section, we'll explore the different transaction execution paths in Sui and how they interact with the consensus mechanism.

Concurrency Challenge

At its core, blockchain technology faces a fundamental concurrency challenge: multiple parties may try to modify or access the same data simultaneously in a decentralized environment. This requires a system for sequencing and validating transactions to support the network's consistency. Sui addresses this challenge through a consensus mechanism, ensuring all nodes agree on the transactions' sequence and state.

Consider a marketplace scenario where Alice and Bob simultaneously attempt to purchase the same asset. The network must resolve this conflict to prevent double-spending, ensuring that at most one transaction succeeds while the other is rightfully rejected.

Fast Path

However, not all transactions require the same level of validation and consensus. For example, if Alice wants to transfer an object that she owns to Bob, the network can process this transaction without sequencing it with respect to all other transactions in the network, as only Alice has the authority to access the object. This is known as the *fast path* execution, where transactions accessing account-owned objects are processed quickly without the need for extensive consensus. No concurrent data access -> simpler challenge -> fast path.

Another ownership model that allows for fast path execution is the *immutable state*. Since immutable objects cannot change, transactions involving them can be processed quickly without the need to sequence them.

Consensus Path

Transactions that do access shared state - on Sui it is represented with shared objects - require sequencing to ensure that the state is updated and consistented across all nodes. This is known

as the execution through *consensus*, where transactions accessing shared objects are subject to the agreement process to maintain network consistency.

Objects owned by Objects

Lastly, it is important to mention that objects owned by other objects are subject to the same rules as the parent object. If the parent object is *shared*, the child object is also transitively shared. If the parent object is immutable, the child object is also immutable.

Summary

- Fast Path: Transactions involving account-owned objects or immutable shared state are processed quickly without the need for extensive consensus.
- Consensus Path: Transactions involving shared objects require sequencing and consensus to ensure network integrity.
- Objects owned by Objects: Child objects inherit the ownership model of the parent object.

Using Objects

In the Object Model chapter we briefly explained the evolution of the Move language from an account-based model to an object-based model. In this chapter, we will dive deeper into the object model and explore how to use objects in your Sui applications. If you haven't read the Object Model chapter yet, we recommend you do so before continuing with this chapter.

The Key Ability

In the Basic Syntax chapter we already covered two out of four abilities - Drop and Copy. They affect the behaviour of the value in a scope and are not directly related to storage. It is time to cover the `key` ability, which allows the struct to be stored.

Historically, the `key` ability was created to mark the type as a *key in the storage*. A type with the `key` ability could be stored at top-level in the storage, and could be *directly owned* by an account or address. With the introduction of the Object Model, the `key` ability naturally became the defining ability for the object.

Object Definition

A struct with the `key` ability is considered an object and can be used in the storage functions. The Sui Verifier will require the first field of the struct to be named `id` and have the type `UID`.

```
public struct Object has key {
    id: UID, // required
    name: String,
}

/// Creates a new Object with a Unique ID
public fun new(name: String, ctx: &mut TxContext): Object {
    Object {
        id: object::new(ctx), // creates a new UID
        name,
    }
}
```

A struct with the `key` ability is still a struct, and can have any number of fields and associated functions. There is no special handling or syntax for packing, accessing or unpacking the struct.

However, because the first field of an object struct must be of type `UID` - a non-copyable and non-droppable type (we will get to it very soon!), the struct transitively cannot have `drop` and `copy` abilities. Thus, the object is non-discardable by design.

Types with the `key` Ability

Due to the `UID` requirement for types with `key`, none of the native types in Move can have the `key` ability, nor can any of the Standard Library types. The `key` ability is only present in the Sui Framework and custom types.

Next Steps

Key ability defines the object in Move, and objects are intended to be *stored*. In the next section we present the `sui::transfer` module, which provides native storage functions for objects.

Further reading

- Type Abilities in the Move Reference.

Storage Functions

The module that defines main storage operations is `sui::transfer`. It is implicitly imported in all packages that depend on the Sui Framework, so, like other implicitly imported modules (e.g. `std::option` or `std::vector`), it does not require adding a use statement.

Overview

The `transfer` module provides functions to perform all three storage operations matching ownership types which we explained before:

> On this page we will only talk about so-called *restricted* storage operations, later we will cover *public* ones, after the `store` ability is introduced.

1. *Transfer* - send an object to an address, put it into *account owned* state;
2. *Share* - put an object into a *shared* state, so it is available to everyone;
3. *Freeze* - put an object into *immutable* state, so it becomes a public constant and can never change.

The `transfer` module is a go-to for most of the storage operations, except a special case with Dynamic Fields awaits us in the next chapter.

Ownership and References: A Quick Recap

In the Ownership and Scope and References chapters, we covered the basics of ownership and references in Move. It is important that you understand these concepts when using storage functions. Here is a quick recap of the most important points:

- The *move* semantics in Move means that the value is *moved* from one scope to another. In other words, if an instance of a type is passed to a function *by value*, it is *moved* to the function scope and can't be accessed in the caller scope anymore.
- To maintain the ownership of the value, you can pass it *by reference*. Either by *immutable reference* `&T` or *mutable reference* `&mut T`. Then the value is *borrowed* and can be accessed in the caller scope, however the owner stays the same.

```
/// Moved by value
public fun take<T>(value: T) { /* value is moved here! */ abort 0 }

/// For immutable reference
public fun borrow<T>(value: &T) { /* value is borrowed here! can be read */ abort 0
}

/// For mutable reference
public fun borrow_mut<T>(value: &mut T) { /* value is mutably borrowed here! */
abort 0 }
```

Transfer

The `transfer::transfer` function is a public function used to transfer an object to another address. Its signature is as follows, only accepts a type with the `key` ability and an address of the recipient. Please, note that the object is passed into the function *by value*, therefore it is *moved* to the function scope and then moved to the recipient address:

```
// File: sui-framework/sources/transfer.move
public fun transfer<T: key>(obj: T, recipient: address);
```

In the next example, you can see how it can be used in a module that defines and sends an object to the transaction sender.

```
module book::transfer_to_sender {

    /// A struct with `key` is an object. The first field is `id: UID`!
    public struct AdminCap has key { id: UID }

    /// `init` function is a special function that is called when the module
    /// is published. It is a good place to create application objects.
    fun init(ctx: &mut TxContext) {
        // Create a new `AdminCap` object, in this scope.
        let admin_cap = AdminCap { id: object::new(ctx) };

        // Transfer the object to the transaction sender.
        transfer::transfer(admin_cap, ctx.sender());

        // admin_cap is gone! Can't be accessed anymore.
    }

    /// Transfers the `AdminCap` object to the `recipient`. Thus, the recipient
    /// becomes the owner of the object, and only they can access it.
    public fun transfer_admin_cap(cap: AdminCap, recipient: address) {
        transfer::transfer(cap, recipient);
    }
}
```

When the module is published, the `init` function will get called, and the `AdminCap` object which we created there will be *transferred* to the transaction sender. The `ctx.sender()` function returns the sender address for the current transaction.

Once the `AdminCap` has been transferred to the sender, for example, to `0xa11ce`, the sender, and only the sender, will be able to access the object. The object is now *account owned*.

> Account owned objects are a subject to *true ownership* - only the account owner can access them. This is a fundamental concept in the Sui storage model.

Let's extend the example with a function that uses `AdminCap` to authorize a mint of a new object and its transfer to another address:

```
/// Some `Gift` object that the admin can `mint_and_transfer`.
public struct Gift has key { id: UID }

/// Creates a new `Gift` object and transfers it to the `recipient`.
public fun mint_and_transfer(
    _: &AdminCap, recipient: address, ctx: &mut TxContext
) {
    let gift = Gift { id: object::new(ctx) };
    transfer::transfer(gift, recipient);
}
```

The `mint_and_transfer` function is a public function that "could" be called by anyone, but it requires an `AdminCap` object to be passed as the first argument by reference. Without it, the function will not be callable. This is a simple way to restrict access to privileged functions called *Capability*. Because the `AdminCap` object is *account owned*, only `0xa11ce` will be able to call the `mint_and_transfer` function.

The `Gift`s sent to recipients will also be *account owned*, each gift being unique and owned exclusively by the recipient.

A quick recap:

- `transfer` function is used to send an object to an address;
- The object becomes *account owned* and can only be accessed by the recipient;
- Functions can be gated by requiring an object to be passed as an argument, creating a *capability*.

Freeze

The `transfer::freeze_object` function is public function used to put an object into an *immutable* state. Once an object is *frozen*, it can never be changed, and it can be accessed by anyone by immutable reference.

The function signature is as follows, only accepts a type with the `key` ability. Just like all other storage functions, it takes the object *by value*:

```
// File: sui-framework/sources/transfer.move
public fun freeze_object<T: key>(obj: T);
```

Let's expand on the previous example and add a function that allows the admin to create a `Config` object and freeze it:

```
/// Some `Config` object that the admin can `create_and_freeze`.
public struct Config has key {
    id: UID,
    message: String
}

/// Creates a new `Config` object and freezes it.
public fun create_and_freeze(
    _: &AdminCap,
    message: String,
    ctx: &mut TxContext
) {
    let config = Config {
        id: object::new(ctx),
        message
    };

    // Freeze the object so it becomes immutable.
    transfer::freeze_object(config);
}

/// Returns the message from the `Config` object.
/// Can access the object by immutable reference!
public fun message(c: &Config): String { c.message }
```

Config is an object that has a `message` field, and the `create_and_freeze` function creates a new `Config` and freezes it. Once the object is frozen, it can be accessed by anyone by immutable reference. The `message` function is a public function that returns the message from the `Config` object. Config is now publicly available by its ID, and the message can be read by anyone.

> Function definitions are not connected to the object's state. It is possible to define a function that takes a mutable reference to an object that is used as frozen. However, it won't be callable on a frozen object.

The `message` function can be called on an immutable `Config` object, however, two functions below are not callable on a frozen object:

```
// === Functions below can't be called on a frozen object! ===

/// The function can be defined, but it won't be callable on a frozen object.
/// Only immutable references are allowed.
public fun message_mut(c: &mut Config): &mut String { &mut c.message }

/// Deletes the `Config` object, takes it by value.
/// Can't be called on a frozen object!
public fun delete_config(c: Config) {
    let Config { id, message: _ } = c;
    id.delete()
}
```

To summarize:

- `transfer::freeze_object` function is used to put an object into an *immutable* state;
- Once an object is *frozen*, it can never be changed, deleted or transferred, and it can be accessed by anyone by immutable reference;

Owned -> Frozen

Since the `transfer::freeze_object` signature accepts any type with the `key` ability, it can take an object that was created in the same scope, but it can also take an object that was owned by an account. This means that the `freeze_object` function can be used to *freeze* an object that was *transferred* to the sender. For security concerns, we would not want to freeze the `AdminCap` object - it would be a security risk to allow access to it to anyone. However, we can freeze the `Gift` object that was minted and transferred to the recipient:

> Single Owner -> Immutable conversion is possible!

```
/// Freezes the `Gift` object so it becomes immutable.
public fun freeze_gift(gift: Gift) {
    transfer::freeze_object(gift);
}
```

Share

The `transfer::share_object` function is a public function used to put an object into a *shared* state. Once an object is *shared*, it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the `key` ability:

```
// File: sui-framework/sources/transfer.move
public fun share_object<T: key>(obj: T);
```

Once an object is *shared*, it is publicly available as a mutable reference.

Special Case: Shared Object Deletion

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

```
/// Creates a new `Config` object and shares it.
public fun create_and_share(message: String, ctx: &mut TxContext) {
    let config = Config {
        id: object::new(ctx),
        message
    };

    // Share the object so it becomes shared.
    transfer::share_object(config);
}
```

The `create_and_share` function creates a new `Config` object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
/// Deletes the `Config` object, takes it by value.
/// Can be called on a shared object!
public fun delete_config(c: Config) {
    let Config { id, message: _ } = c;
    id.delete()
}
```

The `delete_config` function takes the `Config` object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the `Config` object back or attempted to

`freeze` or `transfer` it, the Sui Verifier would reject the transaction.

```
// Won't work!
public fun transfer_shared(c: Config, to: address) {
    transfer::transfer(c, to);
}
```

To summarize:

- `share_object` function is used to put an object into a *shared* state;
- Once an object is *shared*, it can be accessed by anyone by a mutable reference;
- Shared objects can be deleted, but they can't be transferred or frozen.

Next Steps

Now that you know main features of the `transfer` module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Ability: Store

Now that you have an understanding of top-level storage functions which are enabled by the `key` ability, we can talk about the last ability in the list - `store`.

Definition

The `store` is a special ability that allows a type to be *stored* in objects. This ability is required for the type to be used as a field in a struct that has the `key` ability. Another way to put it is that the `store` ability allows the value to be *wrapped* in an object.

> The `store` ability also relaxes restrictions on transfer operations. We talk about it more in the Restricted and Public Transfer section.

Example

In previous sections we already used types with the `key` ability: all objects must have a `UID` field, which we used in examples; we also used the `Storable` type as a part of the `Config` struct. The `Config` type also has the `store` ability.

```
/// This type has the `store` ability.
public struct Storable has store {}

/// Config contains a `Storable` field which must have the `store` ability.
public struct Config has key, store {
    id: UID,
    stores: Storable,
}

/// MegaConfig contains a `Config` field which has the `store` ability.
public struct MegaConfig has key {
    id: UID,
    config: Config, // there it is!
}
```

Types with the `store` Ability

All native types (except for references) in Move have the `store` ability. This includes:

- bool
- unsigned integers
- vector
- address

All of the types defined in the standard library have the `store` ability as well. This includes:

- Option
- String
- TypeName

Further reading

- Type Abilities in the Move Reference.

UID and ID

The `UID` type is defined in the `sui::object` module and is a wrapper around an `ID` which, in turn, wraps the `address` type. The UIDs on Sui are guaranteed to be unique, and can't be reused after the object was deleted.

```
// File: sui-framework/sources/object.move
/// UID is a unique identifier of an object
public struct UID has store {
    id: ID
}

/// ID is a wrapper around an address
public struct ID has store, drop {
    bytes: address
}
```

Fresh UID generation:

- UID is derived from the `tx_hash` and an `index` which is incremented for each new UID.
- The `derive_id` function is implemented in the `sui::tx_context` module, and that is why TxContext is required for UID generation.
- Sui Verifier will not allow using a UID that wasn't created in the same function. That prevents UIDs from being pre-generated and reused after the object was unpacked.

New UID is created with the `object::new(ctx)` function. It takes a mutable reference to TxContext, and returns a new UID.

```
let ctx = &mut tx_context::dummy();
let uid = object::new(ctx);
```

On Sui, `UID` acts as a representation of an object, and allows defining behaviors and features of an object. One of the key-features - Dynamic Fields - is possible because of the `UID` type being explicit. Additionally, it allows the Transfer To Object (TTO) which we will explain later in this chapter.

UID lifecycle

The `UID` type is created with the `object::new(ctx)` function, and it is destroyed with the `object::delete(uid)` function. The `object::delete` consumes the UID *by value*, and it is impossible to delete it unless the value was unpacked from an Object.

```
let ctx = &mut tx_context::dummy();

let char = Character {
    id: object::new(ctx)
};

let Character { id } = char;
id.delete();
```

Keeping the UID

The `UID` does not need to be deleted immediately after the object struct is unpacked. Sometimes it may carry Dynamic Fields or objects transferred to it via Transfer To Object. In such cases, the UID may be kept and stored in a separate object.

Proof of Deletion

The ability to return the UID of an object may be utilized in pattern called *proof of deletion*. It is a rarely used technique, but it may be useful in some cases, for example, the creator or an application may incentivize the deletion of an object by exchanging the deleted IDs for some reward.

In framework development this method could be used to ignore / bypass certain restrictions on "taking" the object. If there's a container that enforces certain logic on transfers, like Kiosk does, there could be a special scenario of skipping the checks by providing a proof of deletion.

This is one of the open topics for exploration and research, and it may be used in various ways.

ID

When talking about `UID` we should also mention the `ID` type. It is a wrapper around the `address` type, and is used to represent an address-pointer. Usually, `ID` is used to point at an object, however, there's no restriction, and no guarantee that the `ID` points to an existing object.

> ID can be received as a transaction argument in a Transaction Block. Alternatively, ID can be created from an `address` value using `to_id()` function.

fresh_object_address

TxContext provides the `fresh_object_address` function which can be utilized to create unique addresses and `ID` - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

Restricted and Public Transfer

Storage Operations that we described in the previous sections are restricted by default - they can only be called in the module defining the object. In other terms, the type must be *internal* to the module to be used in storage operations. This restriction is implemented in the Sui Verifier and is enforced at the bytecode level.

However, to allow objects to be transferred and stored in other modules, these restrictions can be relaxed. The `sui::transfer` module offers a set of *public_** functions that allow calling storage operations in other modules. The functions are prefixed with `public_` and are available to all modules and transactions.

Public Storage Operations

The `sui::transfer` module provides the following public functions. They are almost identical to the ones we already covered, but can be called from any module.

```
// File: sui-framework/sources/transfer.move
/// Public version of the `transfer` function.
public fun public_transfer<T: key + store>(object: T, to: address) {}

/// Public version of the `share_object` function.
public fun public_share_object<T: key + store>(object: T) {}

/// Public version of the `freeze_object` function.
public fun public_freeze_object<T: key + store>(object: T) {}
```

To illustrate the usage of these functions, consider the following example: module A defines an ObjectK with `key` and ObjectKS with `key + store` abilities, and module B tries to implement a `transfer` function for these objects.

> In this example we use `transfer::transfer`, but the behaviour is identical for `share_object` and `freeze_object` functions.

```
/// Defines `ObjectK` and `ObjectKS` with `key` and `key + store`
/// abilities respectively
module book::transfer_a {
    public struct ObjectK has key { id: UID }
    public struct ObjectKS has key, store { id: UID }
}

/// Imports the `ObjectK` and `ObjectKS` types from `transfer_a` and attempts
/// to implement different `transfer` functions for them
module book::transfer_b {
    // types are not internal to this module
    use book::transfer_a::{ObjectK, ObjectKS};

    // Fails! ObjectK is not `store`, and ObjectK is not internal to this module
    public fun transfer_k(k: ObjectK, to: address) {
        sui::transfer::transfer(k, to);
    }

    // Fails! ObjectKS has `store` but the function is not public
    public fun transfer_ks(ks: ObjectKS, to: address) {
        sui::transfer::transfer(ks, to);
    }

    // Fails! ObjectK is not `store`, `public_transfer` requires `store`
    public fun public_transfer_k(k: ObjectK) {
        sui::transfer::public_transfer(k);
    }

    // Works! ObjectKS has `store` and the function is public
    public fun public_transfer_ks(y: ObjectKS, to: address) {
        sui::transfer::public_transfer(y, to);
    }
}
```

To expand on the example above:

- ☐ `transfer_k` fails because ObjectK is not internal to module `transfer_b`
- ☐ `transfer_ks` fails because ObjectKS is not internal to module `transfer_b`
- ☐ `public_transfer_k` fails because ObjectK does not have the `store` ability
- ☑ `public_transfer_ks` works because ObjectKS has the `store` ability and the transfer is public

Implications of `store`

The decision on whether to add the `store` ability to a type should be made carefully. On one hand, it is de-facto a requirement for the type to be *usable* by other applications. On the other hand, it allows *wrapping* and changing the intended storage model. For example, a character may be intended to be owned by accounts, but with the `store` ability it can be frozen (cannot be shared - this transition is restricted).

Advanced Programmability

In previous chapters we've covered the basics of Move and Sui Storage Model. Now it's time to dive deeper into the advanced topics of Sui programmability.

This chapter introduces more complex concepts, practices and features of Move and Sui that are essential for building more sophisticated applications. It is intended for developers who are already familiar with the basics of Move and Sui, and are looking to expand their knowledge and skills.

Transaction Context

Every transaction has the execution context. The context is a set of predefined variables that are available to the program during execution. For example, every transaction has a sender address, and the transaction context contains a variable that holds the sender address.

The transaction context is available to the program through the `TxContext` struct. The struct is defined in the `sui::tx_context` module and contains the following fields:

```
// File: sui-framework/sources/tx_context.move
/// Information about the transaction currently being executed.
/// This cannot be constructed by a transaction--it is a privileged object created
by
/// the VM and passed in to the entrypoint of the transaction as `&mut TxContext`.
struct TxContext has drop {
    /// The address of the user that signed the current transaction
    sender: address,
    /// Hash of the current transaction
    tx_hash: vector<u8>,
    /// The current epoch number
    epoch: u64,
    /// Timestamp that the epoch started at
    epoch_timestamp_ms: u64,
    /// Counter recording the number of fresh id's created while executing
    /// this transaction. Always 0 at the start of a transaction
    ids_created: u64
}
```

Transaction context cannot be constructed manually or directly modified. It is created by the system and passed to the function as a reference in a transaction. Any function called in a Transaction has access to the context and can pass it into the nested calls.

> `TxContext` has to be the last argument in the function signature.

Reading the Transaction Context

With only exception of the `ids_created`, all of the fields in the `TxContext` have getters. The getters are defined in the `sui::tx_context` module and are available to the program. The getters

don't require `&mut` because they don't modify the context.

```
public fun some_action(ctx: &TxContext) {
    let me = ctx.sender();
    let epoch = ctx.epoch();
    let digest = ctx.digest();
    // ...
}
```

Mutability

The `TxContext` is required to create new objects (or just `UID`s) in the system. New UIDs are derived from the transaction digest, and for the digest to be unique, there needs to be a changing parameter. Sui uses the `ids_created` field for that. Every time a new UID is created, the `ids_created` field is incremented by one. This way, the digest is always unique.

Internally, it is represented as the `derive_id` function:

```
// File: sui-framework/sources/tx_context.move
native fun derive_id(tx_hash: vector<u8>, ids_created: u64): address;
```

Generating unique addresses

The underlying `derive_id` function can also be utilized in your program to generate unique addresses. The function itself is not exposed, but a wrapper function `fresh_object_address` is available in the `sui::tx_context` module. It may be useful if you need to generate a unique identifier in your program.

```
// File: sui-framework/sources/tx_context.move
/// Create an `address` that has not been used. As it is an object address, it will
never
/// occur as the address for a user.
/// In other words, the generated address is a globally unique object ID.
public fun fresh_object_address(ctx: &mut TxContext): address {
    let ids_created = ctx.ids_created;
    let id = derive_id(*&ctx.tx_hash, ids_created);
    ctx.ids_created = ids_created + 1;
    id
}
```

Module Initializer

A common use case in many applications is to run certain code just once when the package is published. Imagine a simple store module that needs to create the main Store object upon its publication. In Sui, this is achieved by defining an `init` function within the module. This function will automatically be called when the module is published.

> All of the modules' `init` functions are called during the publishing process. Currently, this behavior is limited to the publish command and does not extend to package upgrades.

```
module book::shop {
    /// The Capability which grants the Shop owner the right to manage
    /// the shop.
    public struct ShopOwnerCap has key, store { id: UID }

    /// The singular Shop itself, created in the `init` function.
    public struct Shop has key {
        id: UID,
        /* ... */
    }

    // Called only once, upon module publication. It must be
    // private to prevent external invocation.
    fun init(ctx: &mut TxContext) {
        // Transfers the ShopOwnerCap to the sender (publisher).
        transfer::transfer(ShopOwnerCap {
            id: object::new(ctx)
        }, ctx.sender());

        // Shares the Shop object.
        transfer::share_object(Shop {
            id: object::new(ctx)
        });
    }
}
```

In the same package, another module can have its own `init` function, encapsulating distinct logic.

```
// In the same package as the `shop` module
module book::bank {

    public struct Bank has key {
        id: UID,
        /* ... */
    }

    fun init(ctx: &mut TxContext) {
        transfer::share_object(Bank {
            id: object::new(ctx)
        });
    }
}
```

`init` features

The function is called on publish, if it is present in the module and follows the rules:

- The function has to be named `init`, be private and have no return values.
- Takes one or two arguments: One Time Witness (optional) and TxContext. With `TxContext` always being the last argument.

```
fun init(ctx: &mut TxContext) { /* ... */}
fun init(otw: OTW, ctx: &mut TxContext) { /* ... */ }
```

TxContext can also be passed as immutable reference: `&TxContext`. However, practically speaking, it should always be `&mut TxContext` since the `init` function can't access the onchain state and to create new objects it requires the mutable reference to the context.

```
fun init(ctx: &TxContext) { /* ... */}
fun init(otw: OTW, ctx: &TxContext) { /* ... */ }
```

Trust and security

While `init` function can be used to create sensitive objects once, it is important to know that the same object (eg. `StoreOwnerCap` from the first example) can still be created in another function. Especially given that new functions can be added to the module during an upgrade. So the `init`

function is a good place to set up the initial state of the module, but it is not a security measure on its own.

There are ways to guarantee that the object was created only once, such as the One Time Witness. And there are ways to limit or disable the upgrade of the module, which we will cover in the Package Upgrades chapter.

Next steps

As follows from the definition, the `init` function is guaranteed to be called only once when the module is published. So it is a good place to put the code that initializes module's objects and sets up the environment and configuration.

For example, if there's a Capability which is required for certain actions, it should be created in the `init` function. In the next chapter we will talk about the `Capability` pattern in more detail.

Pattern: Capability

In programming, a *capability* is a token that gives the owner the right to perform a specific action. It is a pattern that is used to control access to resources and operations. A simple example of a capability is a key to a door. If you have the key, you can open the door. If you don't have the key, you can't open the door. A more practical example is an Admin Capability which allows the owner to perform administrative operations, which regular users cannot.

Capability is an Object

In the Sui Object Model, capabilities are represented as objects. An owner of an object can pass this object to a function to prove that they have the right to perform a specific action. Due to strict typing, the function taking a capability as an argument can only be called with the correct capability.

> There's a convention to name capabilities with the `Cap` suffix, for example, `AdminCap` or `KioskOwnerCap`.

```
module book::capability {
    use std::string::String;
    use sui::event;

    /// The capability granting the application admin the right to create new
    /// accounts in the system.
    public struct AdminCap has key, store { id: UID }

    /// The user account in the system.
    public struct Account has key, store {
        id: UID,
        name: String
    }

    /// A simple `Ping` event with no data.
    public struct Ping has copy, drop { by: ID }

    /// Creates a new account in the system. Requires the `AdminCap` capability
    /// to be passed as the first argument.
    public fun new(_: &AdminCap, name: String, ctx: &mut TxContext): Account {
        Account {
            id: object::new(ctx),
            name,
        }
    }

    /// Account, and any other objects, can also be used as a Capability in the
    /// application. For example, to emit an event.
    public fun send_ping(acc: &Account) {
        event::emit(Ping {
            by: acc.id.to_inner()
        })
    }

    /// Updates the account name. Can only be called by the `Account` owner.
    public fun update(account: &mut Account, name: String) {
        account.name = name;
    }
}
```

Using `init` for Admin Capability

A very common practice is to create a single `AdminCap` object on package publish. This way, the application can have a setup phase where the admin account prepares the state of the application.

```
module book::admin_cap {
    /// The capability granting the admin privileges in the system.
    /// Created only once in the `init` function.
    public struct AdminCap has key { id: UID }

    /// Create the AdminCap object on package publish and transfer it to the
    /// package owner.
    fun init(ctx: &mut TxContext) {
        transfer::transfer(
            AdminCap { id: object::new(ctx) },
            ctx.sender()
        )
    }
}
```

Address check vs Capability

Utilizing objects as capabilities is a relatively new concept in blockchain programming. And in other smart-contract languages, authorization is often performed by checking the address of the sender. This pattern is still viable on Sui, however, overall recommendation is to use capabilities for better security, discoverability, and code organization.

Let's look at how the `new` function that creates a user would look like if it was using the address check:

```
/// Error code for unauthorized access.
const ENotAuthorized: u64 = 0;

/// The application admin address.
const APPLICATION_ADMIN: address = @0xa11ce;

/// Creates a new user in the system. Requires the sender to be the application
/// admin.
public fun new(ctx: &mut TxContext): User {
    assert!(ctx.sender() == APPLICATION_ADMIN, ENotAuthorized);
    User { id: object::new(ctx) }
}
```

And now, let's see how the same function would look like with the capability:

```
/// Grants the owner the right to create new users in the system.
public struct AdminCap {}

/// Creates a new user in the system. Requires the `AdminCap` capability to be
/// passed as the first argument.
public fun new(_: &AdminCap, ctx: &mut TxContext): User {
    User { id: object::new(ctx) }
}
```

Using capabilities has several advantages over the address check:

- Migration of admin rights is easier with capabilities due to them being objects. In case of address, if the admin address changes, all the functions that check the address need to be updated - hence, require a package upgrade.
- Function signatures are more descriptive with capabilities. It is clear that the `new` function requires the `AdminCap` to be passed as an argument. And this function can't be called without it.
- Object Capabilities don't require extra checks in the function body, and hence, decrease the chance of a developer mistake.
- An owned Capability also serves in discovery. The owner of the AdminCap can see the object in their account (via a Wallet or Explorer), and know that they have the admin rights. This is less transparent with the address check.

However, the address approach has its own advantages. For example, if an address is multisig, and transaction building gets more complex, it might be easier to check the address. Also, if there's a central object of the application that is used in every function, it can store the admin address, and this would simplify migration. The central object approach is also valuable for revokable capabilities, where the admin can revoke the capability from the user.

Epoch and Time

Sui has two ways of accessing the current time: `Epoch` and `Time`. The former represents operational periods in the system and changed roughly every 24 hours. The latter represents the current time in milliseconds since the Unix Epoch. Both can be accessed freely in the program.

Epoch

Epochs are used to separate the system into operational periods. During an epoch the validator set is fixed, however, at the epoch boundary, the validator set can be changed. Epochs play a crucial role in the consensus algorithm and are used to determine the current validator set. They are also used as measurement in the staking mechanism.

Epoch can be read from the transaction context:

```
public fun current_epoch(ctx: &TxContext) {
    let epoch = ctx.epoch();
    // ...
}
```

It is also possible to get the unix timestamp of the epoch start:

```
public fun current_epoch_start(ctx: &TxContext) {
    let epoch_start = ctx.epoch_timestamp_ms();
    // ...
}
```

Normally, epochs are used in staking and system operations, however, in custom scenarios they can be used to emulate 24h periods. They are critical if an application relies on the staking logic or needs to know the current validator set.

Time

For a more precise time measurement, Sui provides the `Clock` object. It is a system object that is updated during checkpoints by the system, which stores the current time in milliseconds since the Unix Epoch. The `Clock` object is defined in the `sui::clock` module and has a reserved address `0x6`.

Clock is a shared object, but a transaction attempting to access it mutably will fail. This limitation allows parallel access to the `Clock` object, which is important for maintaining performance.

```
// File: sui-framework/clock.move
/// Singleton shared object that exposes time to Move calls.  This
/// object is found at address 0x6, and can only be read (accessed
/// via an immutable reference) by entry functions.
///
/// Entry Functions that attempt to accept `Clock` by mutable
/// reference or value will fail to verify, and honest validators
/// will not sign or execute transactions that use `Clock` as an
/// input parameter, unless it is passed by immutable reference.
struct Clock has key {
    id: UID,
    /// The clock's timestamp, which is set automatically by a
    /// system transaction every time consensus commits a
    /// schedule, or by `sui::clock::increment_for_testing` during
    /// testing.
    timestamp_ms: u64,
}
```

There is only one public function available in the `Clock` module - `timestamp_ms`. It returns the current time in milliseconds since the Unix Epoch.

```
use sui::clock::Clock;

/// Clock needs to be passed as an immutable reference.
public fun current_time(clock: &Clock) {
    let time = clock.timestamp_ms();
    // ...
}
```

Collections

Collection types are a fundamental part of any programming language. They are used to store a collection of data, such as a list of items. The `vector` type has already been covered in the vector section, and in this chapter we will cover the vector-based collection types offered by the Sui Framework.

Vector

While we have previously covered the `vector` type in the vector section, it is worth going over it again in a new context. This time we will cover the usage of the `vector` type in objects and how it can be used in an application.

```
module book::collections_vector {
    use std::string::String;

    /// The Book that can be sold by a `BookStore`
    public struct Book has key, store {
        id: UID,
        name: String
    }

    /// The BookStore that sells `Book`s
    public struct BookStore has key, store {
        id: UID,
        books: vector<Book>
    }
}
```

VecSet

`VecSet` is a collection type that stores a set of unique items. It is similar to a `vector`, but it does not allow duplicate items. This makes it useful for storing a collection of unique items, such as a list of unique IDs or addresses.

```
module book::collections_vec_set {
    use sui::vec_set::{Self, VecSet};

    public struct App has drop {
        /// `VecSet` used in the struct definition
        subscribers: VecSet<address>
    }

    #[test]
    fun vec_set_playground() {
        let set = vec_set::empty<u8>(); // create an empty set
        let mut set = vec_set::singleton(1); // create a set with a single item

        set.insert(2); // add an item to the set
        set.insert(3);

        assert!(set.contains(&1), 0); // check if an item is in the set
        assert!(set.size() == 3, 1); // get the number of items in the set
        assert!(!set.is_empty(), 2); // check if the set is empty

        set.remove(&2); // remove an item from the set
    }
}
```

VecSet will fail on attempt to insert an item that already exists in the set.

VecMap

`VecMap` is a collection type that stores a map of key-value pairs. It is similar to a `VecSet`, but it allows you to associate a value with each item in the set. This makes it useful for storing a collection of key-value pairs, such as a list of addresses and their balances, or a list of user IDs and their associated data.

Keys in a `VecMap` are unique, and each key can only be associated with a single value. If you try to insert a key-value pair with a key that already exists in the map, the old value will be replaced with the new value.

```
module book::collections {
    use std::string::String;
    use sui::vec_map::{Self, VecMap};

    public struct Metadata has drop {
        name: String,
        /// `VecMap` used in the struct definition
        attributes: VecMap<String, String>
    }

    #[test]
    fun vec_map_playground() {
        let mut map = vec_map::empty(); // create an empty map

        map.insert(2, b"two".to_string()); // add a key-value pair to the map
        map.insert(3, b"three".to_string());

        assert!(map.contains(&2), 0); // check if a key is in the map

        map.remove(&2); // remove a key-value pair from the map
    }
}
```

Limitations

Standard collection types are a great way to store typed data with guaranteed safety and consistency. However, they are limited by the type of data they can store - the type system won't allow you to store a wrong type in a collection; and they're limited in size - by the object size limit. They will work for relatively small-sized sets and lists, but for larger collections you may need to use a different approach.

Another limitations on collection types is inability to compare them. Because the order of insertion is not guaranteed, an attempt to compare a `VecSet` to another `VecSet` may not yield the expected results.

> This behavior is caught by the linter and will emit a warning: *Comparing collections of type 'sui::vec_set::VecSet' may yield unexpected result*

```
let mut set1 = vec_set::empty();
set1.insert(1);
set1.insert(2);

let mut set2 = vec_set::empty();
set2.insert(2);
set2.insert(1);

assert!(set1 == set2, 0);
```

In the example above, the comparison will fail because the order of insertion is not guaranteed, and the two `VecSet` instances may have different orders of elements. And the comparison will fail even if the two `VecSet` instances contain the same elements.

Summary

- Vector is a native type that allows storing a list of items.
- VecSet is built on top of vector and allows storing sets of unique items.
- VecMap is used to store key-value pairs in a map-like structure.
- Vector-based collections are strictly typed and limited by the object size limit and are best suited for small-sized sets and lists.

Next Steps

In the next section we will cover Dynamic Fields - an important primitive that allows for Dynamic Collections - a way to store large collections of data in a more flexible, yet more expensive way.

Dynamic Fields

Sui Object model allows objects to be attached to other objects as *dynamic fields*. The behavior is similar to how a `Map` works in other programming languages. However, unlike a `Map` which in Move would be strictly typed (we have covered it in the Collections section), dynamic fields allow attaching objects of any type. A similar approach from the world of frontend development would be a JavaScript Object type which allows storing any type of data dynamically.

> There's no limit to the number of dynamic fields that can be attached to an object. Thus, dynamic fields can be used to store large amounts of data that don't fit into the object limit size.

Dynamic Fields allow for a wide range of applications, from splitting data into smaller parts to avoid object size limit to attaching objects as a part of application logic.

Definition

Dynamic Fields are defined in the `sui::dynamic_field` module of the Sui Framework. They are attached to object's `UID` via a *name*, and can be accessed using that name. There can be only one field with a given name attached to an object.

File: sui-framework/sources/dynamic_field.move

```
/// Internal object used for storing the field and value
public struct Field<Name: copy + drop + store, Value: store> has key {
    /// Determined by the hash of the object ID, the field name
    /// value and it's type, i.e. hash(parent.id || name || Name)
    id: UID,
    /// The value for the name of this field
    name: Name,
    /// The value bound to this field
    value: Value,
}
```

As the definition shows, dynamic fields are stored in an internal `Field` object, which has the `UID` generated in a deterministic way based on the object ID, the field name, and the field type. The `Field` object contains the field name and the value bound to it. The constraints on the `Name` and `Value` type parameters define the abilities that the key and value must have.

Usage

The methods available for dynamic fields are straightforward: a field can be added with `add`, removed with `remove`, and read with `borrow` and `borrow_mut`. Additionally, the `exists_` method can be used to check if a field exists (for stricter checks with type, there is an `exists_with_type` method).

```
module book::dynamic_collection {
    // a very common alias for `dynamic_field` is `df` since the
    // module name is quite long
    use sui::dynamic_field as df;
    use std::string::String;

    /// The object that we will attach dynamic fields to.
    public struct Character has key {
        id: UID
    }

    // List of different accessories that can be attached to a character.
    // They must have the `store` ability.
    public struct Hat has key, store { id: UID, color: u32 }
    public struct Mustache has key, store { id: UID }

    #[test]
    fun test_character_and_accessories() {
        let ctx = &mut tx_context::dummy();
        let mut character = Character { id: object::new(ctx) };

        // Attach a hat to the character's UID
        df::add(
            &mut character.id,
            b"hat_key",
            Hat { id: object::new(ctx), color: 0xFF0000 }
        );

        // Similarly, attach a mustache to the character's UID
        df::add(
            &mut character.id,
            b"mustache_key",
            Mustache { id: object::new(ctx) }
        );

        // Check that the hat and mustache are attached to the character
        //
        assert!(df::exists_(&character.id, b"hat_key"), 0);
        assert!(df::exists_(&character.id, b"mustache_key"), 1);

        // Modify the color of the hat
        let hat: &mut Hat = df::borrow_mut(&mut character.id, b"hat_key");
        hat.color = 0x00FF00;

        // Remove the hat and mustache from the character
```

```
        let hat: Hat = df::remove(&mut character.id, b"hat_key");
        let mustache: Mustache = df::remove(&mut character.id, b"mustache_key");

        // Check that the hat and mustache are no longer attached to the character
        assert!(!df::exists_(&character.id, b"hat_key"), 0);
        assert!(!df::exists_(&character.id, b"mustache_key"), 1);

        sui::test_utils::destroy(character);
        sui::test_utils::destroy(mustache);
        sui::test_utils::destroy(hat);
    }
}
```

In the example above, we define a `Character` object and two different types of accessories that could never be put together in a vector. However, dynamic fields allow us to store them together in a single object. Both objects are attached to the `Character` via a `vector<u8>` (bytestring literal), and can be accessed using their respective keys.

As you can see, when we attached the accessories to the Character, we passed them *by value*. In other words, both values were moved to a new scope, and their ownership was transferred to the `Character` object. If we changed the ownership of `Character` object, the accessories would have been moved with it.

And the last important property of dynamic fields we should highlight is that they are *accessed through their parent*. This means that the `Hat` and `Mustache` objects are not directly accessible and follow the same rules as the parent object.

Foreign Types as Dynamic Fields

Dynamic fields allow objects to carry data of any type, including those defined in other modules. This is possible due to their generic nature and relatively weak constraints on the type parameters. Let's illustrate this by attaching a few different values to a `Character` object.

```
let mut character = Character { id: object::new(ctx) };

// Attach a `String` via a `vector<u8>` name
df::add(&mut character.id, b"string_key", b"Hello, World!".to_string());

// Attach a `u64` via a `u32` name
df::add(&mut character.id, 1000u32, 1_000_000_000u64);

// Attach a `bool` via a `bool` name
df::add(&mut character.id, true, false);
```

In this example we showed how different types can be used for both *name* and the *value* of a dynamic field. The `String` is attached via a `vector<u8>` name, the `u64` is attached via a `u32` name, and the `bool` is attached via a `bool` name. Anything is possible with dynamic fields!

Orphaned Dynamic Fields

> To prevent orphaned dynamic fields, please, use Dynamic Collection Types such as `Bag` as they track the dynamic fields and won't allow unpacking if there are attached fields.

The `object::delete()` function, which is used to delete a UID, does not track the dynamic fields, and cannot prevent dynamic fields from becoming orphaned. Once the parent UID is deleted, the dynamic fields are not automatically deleted, and they become orphaned. This means that the dynamic fields are still stored in the blockchain, but they will never become accessible again.

```
let hat = Hat { id: object::new(ctx), color: 0xFF0000 };
let mut character = Character { id: object::new(ctx) };

// Attach a `Hat` via a `vector<u8>` name
df::add(&mut character.id, b"hat_key", hat);

// ! DO NOT do this in your code
// ! Danger - deleting the parent object
let Character { id } = character;
id.delete();

// ...`Hat` is now stuck in a limbo, it will never be accessible again
```

Orphaned objects are not a subject to storage rebate, and the storage fees will remain unclaimed. One way to avoid orphaned dynamic fields during unpacking of an object is to return the `UID` and store it somewhere temporarily until the dynamic fields are removed and handled properly.

Custom Type as a Field Name

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

```
/// A custom type with fields in it.
public struct AccessoryKey has copy, drop, store { name: String }

/// An empty key, can be attached only once.
public struct MetadataKey has copy, drop, store {}
```

Two field names that we defined above are `AccessoryKey` and `MetadataKey`. The `AccessoryKey` has a `String` field in it, hence it can be used multiple times with different `name` values. The `MetadataKey` is an empty key, and can be attached only once.

```
let mut character = Character { id: object::new(ctx) };

// Attaching via an `AccessoryKey { name: b"hat" }`
df::add(
    &mut character.id,
    AccessoryKey { name: b"hat".to_string() },
    Hat { id: object::new(ctx), color: 0xFF0000 }
);
// Attaching via an `AccessoryKey { name: b"mustache" }`
df::add(
    &mut character.id,
    AccessoryKey { name: b"mustache".to_string() },
    Mustache { id: object::new(ctx) }
);

// Attaching via a `MetadataKey`
df::add(&mut character.id, MetadataKey {}, 42);
```

As you can see, custom types do work as field names but as long as they can be *constructed* by the module, in other words - if they are *internal* to the module and defined in it. This limitation on

struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Exposing UID

> Mutable access to `UID` is a security risk. Exposing `UID` of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields. Make sure to understand the implications before exposing the `UID` as a mutable reference.

Because dynamic fields are attached to `UID`s, their usage in other modules depends on whether the `UID` can be accessed. By default struct visibility protects the `id` field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to `UID`, dynamic fields can be read in other modules.

```
/// Exposes the UID of the character, so that other modules can read
/// dynamic fields.
public fun uid(c: &Character): &UID {
    &c.id
}
```

In the example above, we show how to expose the `UID` of a `Character` object. This solution may work for some applications, however, it is important to remember that exposed `UID` allows reading *any* dynamic field attached to the object.

If you need to expose the `UID` only within the package, use a restrictive visibility, like `public(package)`, or even better - use more specific accessor methods that would allow only reading specific fields.

```
/// Only allow modules in the same package to access the UID.
public(package) fun uid_package(c: &Character): &UID {
    &c.id
}

/// Allow borrowing dynamic fields from the character.
public fun borrow<Name: copy + store + drop, Value: store>(
    c: &Character,
    n: Name
): &Value {
    df::borrow(&c.id, n)
}
```

Dynamic Fields vs Fields

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Limits

Dynamic Fields are not subject to the object size limit, and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit, which is set to 1000 fields per transaction.

Applications

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them *later* and change the type of the field.

Next Steps

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

Dynamic Object Fields

> This section expands on the Dynamic Fields. Please, read it first to understand the basics of dynamic fields.

Another variation of dynamic fields is *dynamic object fields*, which have certain differences from regular dynamic fields. In this section, we will cover the specifics of dynamic object fields and explain how they differ from regular dynamic fields.

> General recommendation is to avoid using dynamic object fields in favor of (just) dynamic fields, especially if there's no need for direct discovery through the ID. The extra costs of dynamic object fields may not be justified by the benefits they provide.

Definition

Dynamic Object Fields are defined in the `sui::dynamic_object_fields` module in the Sui Framework. They are similar to dynamic fields in many ways, but unlike them, dynamic object fields have an extra constraint on the `Value` type. The `Value` must have a combination of `key` and `store`, not just `store` as in the case of dynamic fields.

They're less explicit in their framework definition, as the concept itself is more abstract:

File: sui-framework/sources/dynamic_object_fields.move

```
/// Internal object used for storing the field and the name associated with the
/// value. The separate type is necessary to prevent key collision with direct
/// usage of dynamic_field
public struct Wrapper<Name> has copy, drop, store {
    name: Name,
}
```

Unlike `Field` type in the Dynamic Fields section, the `Wrapper` type only stores the name of the field. The value is the object itself, and is *not wrapped*.

The constraints on the `Value` type become visible in the methods available for dynamic object fields. Here's the signature for the `add` function:

```
/// Adds a dynamic object field to the object `object: &mut UID` at field
/// specified by `name: Name`. Aborts with `EFieldAlreadyExists` if the object
/// already has that field with that name.
public fun add<Name: copy + drop + store, Value: key + store>(
    // we use &mut UID in several spots for access control
    object: &mut UID,
    name: Name,
    value: Value,
) { /* implementation omitted */ }
```

The rest of the methods which are identical to the ones in the Dynamic Fields section have the same constraints on the `Value` type. Let's list them for reference:

- `add` - adds a dynamic object field to the object
- `remove` - removes a dynamic object field from the object
- `borrow` - borrows a dynamic object field from the object
- `borrow_mut` - borrows a mutable reference to a dynamic object field from the object
- `exists_` - checks if a dynamic object field exists
- `exists_with_type` - checks if a dynamic object field exists with a specific type

Additionally, there is an `id` method which returns the `ID` of the `Value` object without specifying its type.

Usage & Differences with Dynamic Fields

The main difference between dynamic fields and dynamic object fields is that the latter allows storing *only objects* as values. This means that you can't store primitive types like `u64` or `bool`. It may be considered a limitation, if not for the fact that dynamic object fields are *not wrapped* into a separate object.

> The relaxed requirement for wrapping keeps the object available for off-chain discovery via its ID. However, this property may not be outstanding if wrapped object indexing is implemented, making the dynamic object fields a redundant feature.

```
module book::dynamic_object_field {
    use std::string::String;

    // there are two common aliases for the long module name: `dof` and
    // `ofield`. Both are commonly used and met in different projects.
    use sui::dynamic_object_field as dof;
    use sui::dynamic_field as df;

    /// The `Character` that we will use for the example
    public struct Character has key { id: UID }

    /// Metadata that doesn't have the `key` ability
    public struct Metadata has store, drop { name: String }

    /// Accessory that has the `key` and `store` abilities.
    public struct Accessory has key, store { id: UID }

    #[test]
    fun equip_accessory() {
        let ctx = &mut tx_context::dummy();
        let mut character = Character { id: object::new(ctx) };

        // Create an accessory and attach it to the character
        let hat = Accessory { id: object::new(ctx) };

        // Add the hat to the character. Just like with `dynamic_fields`
        dof::add(&mut character.id, b"hat_key", hat);

        // However for non-key structs we can only use `dynamic_field`
        df::add(&mut character.id, b"metadata_key", Metadata {
            name: b"John".to_string()
        });

        // Borrow the hat from the character
        let hat_id = dof::id(&character.id, b"hat_key").extract(); // Option<ID>
        let hat_ref: &Accessory = dof::borrow(&character.id, b"hat_key");
        let hat_mut: &mut Accessory = dof::borrow_mut(&mut character.id,
b"hat_key");
        let hat: Accessory = dof::remove(&mut character.id, b"hat_key");

        // Clean up, Metadata is an orphan now.
        sui::test_utils::destroy(hat);
        sui::test_utils::destroy(character);
```

```
    }
}
```

Pricing Differences

Dynamic Object Fields come a little more expensive than dynamic fields. Because of their internal structure, they require 2 objects: the Wrapper for Name and the Value. Because of this, the cost of adding and accessing object fields (loading 2 objects compared to 1 for dynamic fields) is higher.

Next Steps

Both dynamic field and dynamic object fields are powerful features which allow for innovative solutions in applications. However, they are relatively low-level and require careful handling to avoid orphaned fields. In the next section, we will introduce a higher-level abstraction - Dynamic Collections - which can help with managing dynamic fields and objects more effectively.

Dynamic Collections

Sui Framework offers a variety of collection types that build on the dynamic fields and dynamic object fields concepts. These collections are designed to be a safer and more understandable way to store and manage dynamic fields and objects.

For each collection type we will specify the primitive they use, and the specific features they offer.

> Unlike dynamic (object) fields which operate on UID, collection types have their own type and allow calling associated functions.

Common Concepts

All of the collection types share the same set of methods, which are:

- `add` - adds a field to the collection
- `remove` - removes a field from the collection
- `borrow` - borrows a field from the collection
- `borrow_mut` - borrows a mutable reference to a field from the collection
- `contains` - checks if a field exists in the collection
- `length` - returns the number of fields in the collection
- `is_empty` - checks if the `length` is 0

All collection types support index syntax for `borrow` and `borrow_mut` methods. If you see square brackets in the examples, they are translated into `borrow` and `borrow_mut` calls.

```
let hat: &Hat = &bag[b"key"];
let hat_mut: &mut Hat = &mut bag[b"key"];

// is equivalent to
let hat: &Hat = bag.borrow(b"key");
let hat_mut: &mut Hat = bag.borrow_mut(b"key");
```

In the examples we won't focus on these functions, but rather on the differences between the collection types.

Bag

Bag, as the name suggests, acts as a "bag" of heterogeneous values. It is a simple, non-generic type that can store any data. Bag will never allow orphaned fields, as it tracks the number of fields and can't be destroyed if it's not empty.

```
// File: sui-framework/sources/bag.move
public struct Bag has key, store {
    /// the ID of this bag
    id: UID,
    /// the number of key-value pairs in the bag
    size: u64,
}
```

Due to Bag storing any types, the extra methods it offers is:

- `contains_with_type` - checks if a field exists with a specific type

Used as a struct field:

```
/// Imported from the `sui::bag` module.
use sui::bag::{Self, Bag};

/// An example of a `Bag` as a struct field.
public struct Carrier has key {
    id: UID,
    bag: Bag
}
```

Using the Bag:

```
let mut bag = bag::new(ctx);

// bag has the `length` function to get the number of elements
assert!(bag.length() == 0, 0);

bag.add(b"my_key", b"my_value".to_string());

// length has changed to 1
assert!(bag.length() == 1, 1);

// in order: `borrow`, `borrow_mut` and `remove`
// the value type must be specified
let field_ref: &String = &bag[b"my_key"];
let field_mut: &mut String = &mut bag[b"my_key"];
let field: String = bag.remove(b"my_key");

// length is back to 0 - we can unpack
bag.destroy_empty();
```

ObjectBag

Defined in the `sui::object_bag` module. Identical to Bag, but uses dynamic object fields internally. Can only store objects as values.

Table

Table is a typed dynamic collection that has a fixed type for keys and values. It is defined in the `sui::table` module.

```
// File: sui-framework/sources/table.move
public struct Table<phantom K: copy + drop + store, phantom V: store> has key,
store {
    /// the ID of this table
    id: UID,
    /// the number of key-value pairs in the table
    size: u64,
}
```

Used as a struct field:

```
/// Imported from the `sui::table` module.
use sui::table::{Self, Table};

/// Some record type with `store`
public struct Record has store { /* ... */ }

/// An example of a `Table` as a struct field.
public struct UserRegistry has key {
    id: UID,
    table: Table<address, Record>
}
```

Using the Table:

```
#[test] fun test_table() {
let ctx = &mut tx_context::dummy();

// Table requires explicit type parameters for the key and value
// ...but does it only once in initialization.
let mut table = table::new<address, String>(ctx);

// table has the `length` function to get the number of elements
assert!(table.length() == 0, 0);

table.add(@0xa11ce, b"my_value".to_string());
table.add(@0xb0b, b"another_value".to_string());

// length has changed to 2
assert!(table.length() == 2, 2);

// in order: `borrow`, `borrow_mut` and `remove`
let addr_ref = &table[@0xa11ce];
let addr_mut = &mut table[@0xa11ce];

// removing both values
let _addr = table.remove(@0xa11ce);
let _addr = table.remove(@0xb0b);

// length is back to 0 - we can unpack
table.destroy_empty();
```

ObjectTable

Defined in the `sui::object_table` module. Identical to Table, but uses dynamic object fields internally. Can only store objects as values.

Summary

- Bag - a simple collection that can store any type of data
- ObjectBag - a collection that can store only objects
- Table - a typed dynamic collection that has a fixed type for keys and values
- ObjectTable - same as Table, but can only store objects

LinkedTable

This section is coming soon!

Pattern: Witness

Witness is a pattern of proving an existence by constructing a proof. In the context of programming, witness is a way to prove a certain property of a system by providing a value that can only be constructed if the property holds.

Witness in Move

In the Struct section we have shown that a struct can only be created - or *packed* - by the module defining it. Hence, in Move, a module proves ownership of the type by constructing it. This is one of the most important patterns in Move, and it is widely used for generic type instantiation and authorization.

Practically speaking, for the witness to be used, there has to be a function that expects a witness as an argument. In the example below it is the `new` function that expects a witness of the `T` type to create a `Instance<T>` instance.

> It is often the case that the witness struct is not stored, and for that the function may require the Drop ability for the type.

```
module book::witness {
    /// A struct that requires a witness to be created.
    public struct Instance<T> { t: T }

    /// Create a new instance of `Instance<T>` with the provided T.
    public fun new<T>(witness: T): Instance<T> {
        Instance { t: witness }
    }
}
```

The only way to construct an `Instance<T>` is to call the `new` function with an instance of the type `T`. This is a basic example of the witness pattern in Move. A module providing a witness often has a matching implementation, like the module `book::witness_source` below:

```
module book::witness_source {
    use book::witness::{Self, Instance};

    /// A struct used as a witness.
    public struct W {}

    /// Create a new instance of `Instance<W>`.
    public fun new_instance(): Instance<W> {
        witness::new(W {})
    }
}
```

The instance of the struct `W` is passed into the `new_instance` function to create an `Instance<W>`, thereby proving that the module `book::witness_source` owns the type `W`.

Instantiating a Generic Type

Witness allows generic types to be instantiated with a concrete type. This is useful for inheriting associated behaviors from the type with an option to extend them, if the module provides the ability to do so.

```
// File: sui-framework/sources/balance.move
/// A Supply of T. Used for minting and burning.
public struct Supply<phantom T> has key, store {
    id: UID,
    value: u64
}

/// Create a new supply for type T with the provided witness.
public fun create_supply<T: drop>(_w: T): Supply<T> {
    Supply { value: 0 }
}

/// Get the `Supply` value.
public fun supply_value<T>(supply: &Supply<T>): u64 {
    supply.value
}
```

In the example above, which is borrowed from the `balance` module of the Sui Framework, the `Supply` a generic struct that can be constructed only by supplying a witness of the type `T`. The witness is taken by value and *discarded* - hence the `T` must have the drop ability.

The instantiated `Supply<T>` can then be used to mint new `Balance<T>`'s, where `T` is the type of the supply.

```
// File: sui-framework/sources/balance.move
/// Storable balance.
struct Balance<phantom T> has store {
    value: u64
}

/// Increase supply by `value` and create a new `Balance<T>` with this value.
public fun increase_supply<T>(self: &mut Supply<T>, value: u64): Balance<T> {
    assert!(value < (18446744073709551615u64 - self.value), EOverflow);
    self.value = self.value + value;
    Balance { value }
}
```

One Time Witness

While a struct can be created any number of times, there are cases where a struct should be guaranteed to be created only once. For this purpose, Sui provides the "One-Time Witness" - a special witness that can only be used once. We explain it in more detail in the next section.

Summary

- Witness is a pattern of proving a certain property by constructing a proof.
- In Move, a module proves ownership of a type by constructing it.
- Witness is often used for generic type instantiation and authorization.

Next Steps

In the next section, we will learn about the One Time Witness pattern.

One Time Witness

While regular Witness is a great way to statically prove the ownership of a type, there are cases where we need to ensure that a Witness is instantiated only once. And this is the purpose of the One Time Witness (OTW).

Definition

The OTW is a special type of Witness that can be used only once. It cannot be manually created and it is guaranteed to be unique per module. Sui Adapter treats a type as an OTW if it follows these rules:

1. Has only `drop` ability.
2. Has no fields.
3. Is not a generic type.
4. Named after the module with all uppercase letters.

Here is an example of an OTW:

```
module book::one_time {
    /// The OTW for the `book::one_time` module.
    /// Only `drop`, no fields, no generics, all uppercase.
    public struct ONE_TIME has drop {}

    /// Receive the instance of `ONE_TIME` as the first argument.
    fun init(otw: ONE_TIME, ctx: &mut TxContext) {
        // do something with the OTW
    }
}
```

The OTW cannot be constructed manually, and any code attempting to do so will result in a compilation error. The OTW can be received as the first argument in the module initializer. And because the `init` function is called only once per module, the OTW is guaranteed to be instantiated only once.

Enforcing the OTW

To check if a type is an OTW, `sui::types` module of the Sui Framework offers a special function `is_one_time_witness` that can be used to check if the type is an OTW.

```
use sui::types;

const ENotOneTimeWitness: u64 = 1;

/// Takes an OTW as an argument, aborts if the type is not OTW.
public fun takes_witness<T: drop>(otw: T) {
    assert!(types::is_one_time_witness(&otw), ENotOneTimeWitness);
}
```

Summary

The OTW pattern is a great way to ensure that a type is used only once. Most of the developers should understand how to define and receive the OTW, while the OTW checks and enforcement is mostly needed in libraries and frameworks. For example, the `sui::coin` module requires an OTW in the `coin::create_currency` method, therefore enforcing that the `coin::TreasuryCap` is created only once.

OTW is a powerful tool that lays the foundation for the Publisher object, which we will cover in the next section.

Publisher Authority

In application design and development, it is often needed to prove publisher authority. This is especially important in the context of digital assets, where the publisher may enable or disable certain features for their assets. The Publisher Object is an object, defined in the Sui Framework, that allows the publisher to prove their *authority over a type*.

Definition

The Publisher object is defined in the `sui::package` module of the Sui Framework. It is a very simple, non-generic object that can be initialized once per module (and multiple times per package) and is used to prove the authority of the publisher over a type. To claim a Publisher object, the publisher must present a One Time Witness to the `package::claim` function.

```
// File: sui-framework/sources/package.move
public struct Publisher has key, store {
    id: UID,
    package: String,
    module_name: String,
}
```

> If you're not familiar with the One Time Witness, you can read more about it here.

Here's a simple example of claiming a `Publisher` object in a module:

```
module book::publisher {
    /// Some type defined in the module.
    public struct Book {}

    /// The OTW for the module.
    public struct PUBLISHER has drop {}

    /// Uses the One Time Witness to claim the Publisher object.
    fun init(otw: PUBLISHER, ctx: &mut TxContext) {
        // Claim the Publisher object.
        let publisher = sui::package::claim(otw, ctx);

        // Usually it is transferred to the sender.
        // It can also be stored in another object.
        transfer::public_transfer(publisher, ctx.sender())
    }
}
```

Usage

The Publisher object has two functions associated with it which are used to prove the publisher's authority over a type:

```
// Checks if the type is from the same module, hence the `Publisher` has the
// authority over it.
assert!(publisher.from_module<Book>(), 0);

// Checks if the type is from the same package, hence the `Publisher` has the
// authority over it.
assert!(publisher.from_package<Book>(), 0);
```

Publisher as Admin Role

For small applications or simple use cases, the Publisher object can be used as an admin capability. While in the broader context, the Publisher object has control over system configurations, it can also be used to manage the application's state.

```
/// Some action in the application gated by the Publisher object.
public fun admin_action(cap: &Publisher, /* app objects... */ param: u64) {
    assert!(cap.from_module<Book>(), ENotAuthorized);

    // perform application-specific action
}
```

However, Publisher misses some native properties of Capabilities, such as type safety and expressiveness. The signature for the `admin_action` is not very explicit, can be called by anyone else. And due to `Publisher` object being standard, there now is a risk of unauthorized access if the `from_module` check is not performed. So it's important to be cautious when using the `Publisher` object as an admin role.

Role on Sui

Publisher is required for certain features on Sui. Object Display can be created only by the Publisher, and TransferPolicy - an important component of the Kiosk system - also requires the Publisher object to prove ownership of the type.

Next Steps

In the next chapter we will cover the first feature that requires the Publisher object - Object Display - a way to describe objects for clients, and standardize metadata. A must-have for user-friendly applications.

Object Display

Objects on Sui are explicit in their structure and behavior and can be displayed in an understandable way. However, to support richer metadata for clients, there's a standard and efficient way of "describing" them to the client - the `Display` object defined in the Sui Framework.

Background

Historically, there were different attempts to agree on a standard structure of an object so it can be displayed in a user interface. One of the approaches was to define certain fields in the object struct which, when present, would be used in the UI. This approach was not flexible enough and required developers to define the same fields in every object, and sometimes the fields did not make sense for the object.

```
/// An attempt to standardize the object structure for display.
public struct CounterWithDisplay has key {
    id: UID,
    /// If this field is present it will be displayed in the UI as `name`.
    name: String,
    /// If this field is present it will be displayed in the UI as `description`.
    description: String,
    // ...
    image: String,
    /// Actual fields of the object.
    counter: u64,
    // ...
}
```

If any of the fields contained static data, it would be duplicated in every object. And, since Move does not have interfaces, it is not possible to know if an object has a specific field without "manually" checking the object's type, which makes the client fetching more complex.

Object Display

To address these issues, Sui introduces a standard way of describing an object for display. Instead of defining fields in the object struct, the display metadata is stored in a separate object, which is associated with the type. This way, the display metadata is not duplicated, and it is easy to extend and maintain.

Another important feature of Sui Display is the ability to define templates and use object fields in those templates. Not only it allows for a more flexible display, but it also frees the developer from the need to define the same fields with the same names and types in every object.

> The Object Display is natively supported by the Sui Fullnode, and the client can fetch the display metadata for any object if the object type has a Display associated with it.

```
module book::arena {
    use std::string::String;
    use sui::package;
    use sui::display;

    /// The One Time Witness to claim the `Publisher` object.
    public struct ARENA has drop {}

    /// Some object which will be displayed.
    public struct Hero has key {
        id: UID,
        class: String,
        level: u64,
    }

    /// In the module initializer we create the `Publisher` object, and then
    /// the Display for the `Hero` type.
    fun init(otw: ARENA, ctx: &mut TxContext) {
        let publisher = package::claim(otw, ctx);
        let mut display = display::new<Hero>(&publisher, ctx);

        display.add(
            b"name".to_string(),
            b"{class} (lvl. {level})".to_string()
        );

        display.add(
            b"description".to_string(),
            b"One of the greatest heroes of all time. Join us!".to_string()
        );

        display.add(
            b"link".to_string(),
            b"https://example.com/hero/{id}".to_string()
        );

        display.add(
            b"image_url".to_string(),
            b"https://example.com/hero/{class}.jpg".to_string()
        );

        // Update the display with the new data.
        // Must be called to apply changes.
        display.update_version();
```

```
        transfer::public_transfer(publisher, ctx.sender());
        transfer::public_transfer(display, ctx.sender());
    }
}
```

Creator Privilege

While the objects can be owned by accounts and may be a subject to True Ownership, the Display can be owned by the creator of the object. This way, the creator can update the display metadata and apply the changes globally without the need to update every object. The creator can also transfer Display to another account or even build an application around the object with custom functionality to manage the metadata.

Standard Fields

The fields that are supported most widely are:

- `name` - A name for the object. The name is displayed when users view the object.
- `description` - A description for the object. The description is displayed when users view the object.
- `link` - A link to the object to use in an application.
- `image_url` - A URL or a blob with the image for the object.
- `thumbnail_url` - A URL to a smaller image to use in wallets, explorers, and other products as a preview.
- `project_url` - A link to a website associated with the object or creator.
- `creator` - A string that indicates the object creator.

> Please, refer to the Sui Documentation for the most up-to-date list of supported fields.

While there's a standard set of fields, the Display object does not enforce them. The developer can define any fields they need, and the client can use them as they see fit. Some applications may require additional fields, and omit other, and the Display is flexible enough to support them.

Working with Display

The `Display` object is defined in the `sui::display` module. It is a generic struct that takes a phantom type as a parameter. The phantom type is used to associate the `Display` object with the type it describes. The `fields` of the `Display` object are a `VecMap` of key-value pairs, where the key is the field name and the value is the field value. The `version` field is used to version the display metadata, and is updated on the `update_display` call.

File: sui-framework/sources/display.move

```
struct Display<phantom T: key> has key, store {
    id: UID,
    /// Contains fields for display. Currently supported
    /// fields are: name, link, image and description.
    fields: VecMap<String, String>,
    /// Version that can only be updated manually by the Publisher.
    version: u16
}
```

The Publisher object is required to a new Display, since it serves as the proof of ownership of type.

Template Syntax

Currently, Display supports simple string interpolation and can use struct fields (and paths) in its templates. The syntax is trivial - `{path}` is replaced with the value of the field at the path. The path is a dot-separated list of field names, starting from the root object in case of nested fields.

```
/// Some common metadata for objects.
public struct Metadata has store {
    name: String,
    description: String,
    published_at: u64
}

/// The type with nested Metadata field.
public struct LittlePony has key, store {
    id: UID,
    image_url: String,
    metadata: Metadata
}
```

The Display for the type `LittlePony` above could be defined as follows:

```
{
  "name": "Just a pony",
  "image_url": "{image_url}",
  "description": "{metadata.description}"
}
```

Multiple Display Objects

There's no restriction to how many `Display<T>` objects can be created for a specific `T`. However, the most recently updated `Display<T>` will be used by the fullnode.

Further Reading

- Sui Object Display is Sui Documentation
- Publisher - the representation of the creator

Events

Events are a way to notify off-chain listeners about on-chain events. They are used to emit additional information about the transaction that is not stored - and, hence, can't be accessed - on-chain. Events are emitted by the `sui::event` module located in the Sui Framework.

> Any custom type with the copy and drop abilities can be emitted as an event. Sui Verifier requires the type to be internal to the module.

```
// File: sui-framework/sources/event.move
module sui::event {
    /// Emit a custom Move event, sending the data offchain.
    ///
    /// Used for creating custom indexes and tracking onchain
    /// activity in a way that suits a specific application the most.
    ///
    /// The type `T` is the main way to index the event, and can contain
    /// phantom parameters, eg `emit(MyEvent<phantom T>)`.
    public native fun emit<T: copy + drop>(event: T);
}
```

Emitting Events

Events are emitted using the `emit` function in the `sui::event` module. The function takes a single argument - the event to be emitted. The event data is passed by value,

```
module book::events {
    use sui::coin::Coin;
    use sui::sui::SUI;
    use sui::event;

    /// The item that can be purchased.
    public struct Item has key { id: UID }

    /// Event emitted when an item is purchased. Contains the ID of the item and
    /// the price for which it was purchased.
    public struct ItemPurchased has copy, drop {
        item: ID,
        price: u64
    }

    /// A marketplace function which performs the purchase of an item.
    public fun purchase(coin: Coin<SUI>, ctx: &mut TxContext) {
        let item = Item { id: object::new(ctx) };

        // Create an instance of `ItemPurchased` and pass it to `event::emit`.
        event::emit(ItemPurchased {
            item: object::id(&item),
            price: coin.value()
        });

        // Omitting the rest of the implementation to keep the example simple.
        abort 0
    }
}
```

The Sui Verifier requires the type passed to the `emit` function to be *internal to the module*. So emitting a type from another module will result in a compilation error. Primitive types, although they match the *copy* and *drop* requirement, are not allowed to be emitted as events.

Event Structure

Events are a part of the transaction result and are stored in the *transaction effects*. As such, they natively have the `sender` field which is the address who sent the transaction. So adding a "sender" field to the event is not necessary. Similarly, event metadata contains the timestamp. But it is important to note that the timestamp is relative to the node and may vary a little from node to node.

Sui Framework

Sui Framework is a default dependency set in the Package Manifest. It depends on the Standard Library and provides Sui-specific features, including the interaction with the storage, and Sui-specific native types and modules.

For convenience, we grouped the modules in the Sui Framework into multiple categories. But they're still part of the same framework.

Core

Module	Description	Chapter
sui::address	Adds conversion methods to the address type	Address
sui::transfer	Implements the storage operations for Objects	It starts with an Object
sui::tx_context	Contains the `TxContext` struct and methods to read it	Transaction Context
sui::object	Defines the `UID` and `ID` type, required for creating objects	It starts with an Object
sui::clock	Defines the `Clock` type and its methods	Epoch and Time
sui::dynamic_field	Implements methods to add, use and remove dynamic fields	Dynamic Fields
sui::dynamic_object_field	Implements methods to add, use and remove dynamic object fields	Dynamic Object Fields
sui::event	Allows emitting events for off-chain listeners	Events
sui::package	Defines the `Publisher` type and package upgrade methods	Publisher, Package Upgrades
sui::display	Implements the `Display` object and ways to create and update it	Display

Collections

Module	Description	Chapter
sui::vec_set	Implements a set type	Collections
sui::vec_map	Implements a map with vector keys	Collections
sui::table	Implements the `Table` type and methods to interact with it	Dynamic Collections
sui::linked_table	Implements the `LinkedTable` type and methods to interact with it	Dynamic Collections
sui::bag	Implements the `Bag` type and methods to interact with it	Dynamic Collections
sui::object_table	Implements the `ObjectTable` type and methods to interact with it	Dynamic Collections
sui::object_bag	Implements the `ObjectBag` type and methods to interact with it	Dynamic Collections

Utilities

Module	Description	Chapter
sui::bcs	Implements the BCS encoding and decoding functions	Binary Canonical Serialization

Exported Addresses

Sui Framework exports two named addresses: `sui = 0x2` and `std = 0x1` from the std dependency.

```
[addresses]
sui = "0x2"

# Exported from the MoveStdlib dependency
std = "0x1"
```

Implicit Imports

Just like with Standard Library, some of the modules and types are imported implicitly in the Sui Framework. This is the list of modules and types that are available without explicit `use` import:

- sui::object
- sui::object::ID
- sui::object::UID
- sui::tx_context
- sui::tx_context::TxContext
- sui::transfer

Source Code

The source code of the Sui Framework is available in the Sui repository.

Pattern: Hot Potato

A case in the abilities system - a struct without any abilities - is called *hot potato*. It cannot be stored (not as an object nor as a field in another struct), it cannot be copied or discarded. Hence, once constructed, it must be gracefully unpacked by its module, or the transaction will abort due to unused value without drop.

> If you're familiar with languages that support *callbacks*, you can think of a hot potato as an obligation to call a callback function. If you don't call it, the transaction will abort.

The name comes from the children's game where a ball is passed quickly between players, and none of the players want to be the last one holding it when the music stops, or they are out of the game. This is the best illustration of the pattern - the instance of a hot-potato struct is passed between calls, and none of the modules can keep it.

Defining a Hot Potato

A hot potato can be any struct with no abilities. For example, the following struct is a hot potato:

```
public struct Request {}
```

Because the `Request` has no abilities and cannot be stored or ignored, the module must provide a function to unpack it. For example:

```
/// Constructs a new `Request`
public fun new_request(): Request { Request {} }

/// Unpacks the `Request`. Due to the nature of the hot potato, this function
/// must be called to avoid aborting the transaction.
public fun confirm_request(request: Request) {
    let Request {} = request;
}
```

Example Usage

In the following example, the `Promise` hot potato is used to ensure that the borrowed value, when taken from the container, is returned back to it. The `Promise` struct contains the ID of the borrowed object, and the ID of the container, ensuring that the borrowed value was not swapped for another and is returned to the correct container.

```
/// A generic container for any Object with `key + store`. The Option type
/// is used to allow taking and putting the value back.
public struct Container<T: key + store> has key {
    id: UID,
    value: Option<T>,
}

/// A Hot Potato struct that is used to ensure the borrowed value is returned.
public struct Promise {
    /// The ID of the borrowed object. Ensures that there wasn't a value swap.
    id: ID,
    /// The ID of the container. Ensures that the borrowed value is returned to
    /// the correct container.
    container_id: ID,
}

/// A module that allows borrowing the value from the container.
public fun borrow_val<T: key + store>(container: &mut Container<T>): (T, Promise) {
    assert!(container.value.is_some());
    let value = container.value.extract();
    let id = object::id(&value);
    (value, Promise { id, container_id: object::id(container) })
}

/// Put the taken item back into the container.
public fun return_val<T: key + store>(
    container: &mut Container<T>, value: T, promise: Promise
) {
    let Promise { id, container_id } = promise;
    assert!(object::id(container) == container_id);
    assert!(object::id(&value) == id);
    container.value.fill(value);
}
```

Applications

Below we list some of the common use cases for the hot potato pattern.

Borrowing

As shown in the example above, the hot potato is very effective for borrowing with a guarantee that the borrowed value is returned to the correct container. While the example focuses on a value stored inside an `Option`, the same pattern can be applied to any other storage type, say a dynamic field.

Flash Loans

Canonical example of the hot potato pattern is flash loans. A flash loan is a loan that is borrowed and repaid in the same transaction. The borrowed funds are used to perform some operations, and the repaid funds are returned to the lender. The hot potato pattern ensures that the borrowed funds are returned to the lender.

An example usage of this pattern may look like this:

```
// Borrow the funds from the lender.
let (asset_a, potato) = lender.borrow(amount);

// Perform some operations with the borrowed funds.
let asset_b = dex.trade(loan);
let proceeds = another_contract::do_something(asset_b);

// Keep the commission and return the rest to the lender.
let pay_back = proceeds.split(amount, ctx);
lender.repay(pay_back, potato);
transfer::public_transfer(proceeds, ctx.sender());
```

Variable-path execution

The hot potato pattern can be used to introduce variation in the execution path. For example, if there is a module which allows purchasing a `Phone` for some "Bonus Points" or for USD, the hot potato can be used to decouple the purchase from the payment. The approach is very similar to how some shops work - you take the item from the shelf, and then you go to the cashier to pay for it.

```
/// A `Phone`. Can be purchased in a store.
public struct Phone has key, store { id: UID }

/// A ticket that must be paid to purchase the `Phone`.
public struct Ticket { amount: u64 }

/// Return the `Phone` and the `Ticket` that must be paid to purchase it.
public fun purchase_phone(ctx: &mut TxContext): (Phone, Ticket) {
    (
        Phone { id: object::new(ctx) },
        Ticket { amount: 100 }
    )
}

/// The customer may pay for the `Phone` with `BonusPoints` or `SUI`.
public fun pay_in_bonus_points(ticket: Ticket, payment: Coin<BONUS>) {
    let Ticket { amount } = ticket;
    assert!(payment.value() == amount);
    abort 0 // omitting the rest of the function
}

/// The customer may pay for the `Phone` with `USD`.
public fun pay_in_usd(ticket: Ticket, payment: Coin<USD>) {
    let Ticket { amount } = ticket;
    assert!(payment.value() == amount);
    abort 0 // omitting the rest of the function
}
```

This decoupling technique allows separating the purchase logic from the payment logic, making the code more modular and easier to maintain. The `Ticket` could be split into its own module, providing a basic interface for the payment, and the shop implementation could be extended to support other goods without changing the payment logic.

Compositional Patterns

Hot potato can be used to link together different modules in a compositional way. Its module may define ways to interact with the hot potato, for example, stamp it with a type signature, or to extract some information from it. This way, the hot potato can be passed between different modules, and even different packages within the same transaction.

The most important compositional pattern is the Request Pattern, which we will cover in the next section.

Usage in the Sui Framework

The pattern is used in various forms in the Sui Framework. Here are some examples:

- `sui::borrow` - uses hot potato to ensure that the borrowed value is returned to the correct container.
- `sui::transfer_policy` - defines a `TransferRequest` - a hot potato which can only be consumed if all conditions are met.
- `sui::token` - in the Closed Loop Token system, an `ActionRequest` carries the information about the performed action and collects approvals similarly to `TransferRequest`.

Summary

- A hot potato is a struct without abilities, it must come with a way to create and destroy it.
- Hot potatoes are used to ensure that some action is taken before the transaction ends, similar to a callback.
- Most common use cases for hot potato are borrowing, flash loans, variable-path execution, and compositional patterns.

Binary Canonical Serialization

Binary Canonical Serialization (BCS) is a binary encoding format for structured data. It was originally designed in Diem, and became the standard serialization format for Move. BCS is simple, efficient, deterministic, and easy to implement in any programming language.

> The full format specification is available in the BCS repository.

Format

BCS is a binary format that supports unsigned integers up to 256 bits, options, booleans, unit (empty value), fixed and variable-length sequences, and maps. The format is designed to be deterministic, meaning that the same data will always be serialized to the same bytes.

> "BCS is not a self-describing format. As such, in order to deserialize a message, one must know the message type and layout ahead of time" from the README

Integers are stored in little-endian format, and variable-length integers are encoded using a variable-length encoding scheme. Sequences are prefixed with their length as ULEB128, enumerations are stored as the index of the variant followed by the data, and maps are stored as an ordered sequence of key-value pairs.

Structs are treated as a sequence of fields, and the fields are serialized in the order they are defined in the struct. The fields are serialized using the same rules as the top-level data.

Using BCS

The Sui Framework includes the sui::bcs module for encoding and decoding data. Encoding functions are native to the VM, and decoding functions are implemented in Move.

Encoding

To encode data, use the `bcs::to_bytes` function, which converts data references into byte vectors. This function supports encoding any types, including structs.

```
// File: move-stdlib/sources/bcs.move
public native fun to_bytes<T>(t: &T): vector<u8>;
```

The following example shows how to encode a struct using BCS. The `to_bytes` function can take any value and encode it as a vector of bytes.

```
use sui::bcs;

// 0x01 - a single byte with value 1 (or 0 for false)
let bool_bytes = bcs::to_bytes(&true);
// 0x2a - just a single byte
let u8_bytes = bcs::to_bytes(&42u8);
// 0x2a00000000000000 - 8 bytes
let u64_bytes = bcs::to_bytes(&42u64);
// address is a fixed sequence of 32 bytes
// 0x0000000000000000000000000000000000000000000000000000000000000002
let addr = bcs::to_bytes(&@sui);
```

Encoding a Struct

Structs encode similarly to simple types. Here is how to encode a struct using BCS:

```
let data = CustomData {
    num: 42,
    string: b"hello, world!".to_string(),
    value: true
};

let struct_bytes = bcs::to_bytes(&data);

let mut custom_bytes = vector[];
custom_bytes.append(bcs::to_bytes(&42u8));
custom_bytes.append(bcs::to_bytes(&b"hello, world!".to_string()));
custom_bytes.append(bcs::to_bytes(&true));

// struct is just a sequence of fields, so the bytes should be the same!
assert!(&struct_bytes == &custom_bytes, 0);
```

Decoding

Because BCS does not self-describe and Move is statically typed, decoding requires prior knowledge of the data type. The `sui::bcs` module provides various functions to assist with this process.

Wrapper API

BCS is implemented as a wrapper in Move. The decoder takes the bytes by value, and then allows the caller to *peel off* the data by calling different decoding functions, prefixed with `peel_*`. The data is split off the bytes, and the remainder bytes are kept in the wrapper until the `into_remainder_bytes` function is called.

```
use sui::bcs;

// BCS instance should always be declared as mutable
let mut bcs = bcs::new(x"0100000000000000");

// Same bytes can be read differently, for example: Option<u64>
let value: Option<u64> = bcs.peel_option_u64();

assert!(value.is_some(), 0);
assert!(value.borrow() == &0, 1);

let remainder = bcs.into_remainder_bytes();

assert!(remainder.length() == 0, 2);
```

There is a common practice to use multiple variables in a single `let` statement during decoding. It makes code a little bit more readable and helps to avoid unnecessary copying of the data.

```
let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// mind the order!!!
// handy way to peel multiple values
let (bool_value, u8_value, u64_value) = (
    bcs.peel_bool(),
    bcs.peel_u8(),
    bcs.peel_u64()
);
```

Decoding Vectors

While most of the primitive types have a dedicated decoding function, vectors need special handling, which depends on the type of the elements. For vectors, first you need to decode the length of the vector, and then decode each element in a loop.

```
let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

// bcs.peel_vec_length() peels the length of the vector :)
let mut len = bcs.peel_vec_length();
let mut vec = vector[];

// then iterate depending on the data type
while (len > 0) {
    vec.push_back(bcs.peel_u64()); // or any other type
    len = len - 1;
};

assert!(vec.length() == 1, 0);
```

For most common scenarios, `bcs` module provides a basic set of functions for decoding vectors:

- `peel_vec_address(): vector<address>`
- `peel_vec_bool(): vector<bool>`
- `peel_vec_u8(): vector<u8>`
- `peel_vec_u64(): vector<u64>`
- `peel_vec_u128(): vector<u128>`
- `peel_vec_vec_u8(): vector<vector<u8>>` - vector of byte vectors

Decoding Option

Option is represented as a vector of either 0 or 1 element. To read an option, you would treat it like a vector and check its length (first byte - either 1 or 0).

```
let mut bcs = bcs::new(x"00");
let is_some = bcs.peel_bool();

assert!(is_some == false, 0);

let mut bcs = bcs::new(x"0101");
let is_some = bcs.peel_bool();
let value = bcs.peel_u8();

assert!(is_some == true, 1);
assert!(value == 1, 2);
```

> If you need to decode an option of a custom type, use the method in the code snippet above.

The most common scenarios, `bcs` module provides a basic set of functions for decoding Option's:

- `peel_option_address(): Option<address>`
- `peel_option_bool(): Option<bool>`
- `peel_option_u8(): Option<u8>`
- `peel_option_u64(): Option<u64>`
- `peel_option_u128(): Option<u128>`

Decoding Structs

Structs are decoded field by field, and there is no standard function to automatically decode bytes into a Move struct, and it would have been a violation of the Move's type system. Instead, you need to decode each field manually.

```
// some bytes...
let mut bcs = bcs::new(x"0101010F0000000000F00000000000");

let (age, is_active, name) = (
    bcs.peel_u8(),
    bcs.peel_bool(),
    bcs.peel_vec_u8().to_string()
);

let user = User { age, is_active, name };
```

Summary

Binary Canonical Serialization is an efficient binary format for structured data, ensuring consistent serialization across platforms. The Sui Framework provides comprehensive tools for working with BCS, allowing extensive functionality through built-in functions.

Move 2024 Migration Guide

Move 2024 is the new edition of the Move language that is maintained by Mysten Labs. This guide is intended to help you understand the differences between the 2024 edition and the previous version of the Move language.

> This guide provides a high-level overview of the changes in the new edition. For a more detailed and exhaustive list of changes, refer to the Sui Documentation.

Using the New Edition

To use the new edition, you need to specify the edition in the `move` file. The edition is specified in the `move` file using the `edition` keyword. Currently, the only available edition is `2024.beta`.

```
edition = "2024.beta";
```

Migration Tool

The Move CLI has a migration tool that updates the code to the new edition. To use the migration tool, run the following command:

```
$ sui move migrate
```

The migration tool will update the code to use the `let mut` syntax, the new `public` modifier for strucs, and the `public(package)` function visibility instead of `friend` declarations.

Mutable bindings with `let mut`

Move 2024 introduces `let mut` syntax to declare mutable variables. The `let mut` syntax is used to declare a mutable variable that can be changed after it is declared.

> `let mut` declaration is now required for mutable variables. Compiler will emit an error if you try to reassign a variable without the `mut` keyword.

```
// Move 2020
let x: u64 = 10;
x = 20;

// Move 2024
let mut x: u64 = 10;
x = 20;
```

Additionally, the `mut` keyword is used in tuple destructuring and function arguments to declare mutable variables.

```
// takes by value and mutates
fun takes_by_value_and_mutates(mut v: Value): Value {
    v.field = 10;
    v
}

// `mut` should be placed before the variable name
fun destruct() {
    let (x, y) = point::get_point();
    let (mut x, y) = point::get_point();
    let (mut x, mut y) = point::get_point();
}

// in struct unpack
fun unpack() {
    let Point { x, mut y } = point::get_point();
    let Point { mut x, mut y } = point::get_point();
}
```

Friends are Deprecated

In Move 2024, the `friend` keyword is deprecated. Instead, you can use the `public(package)` visibility modifier to make functions visible to other modules in the same package.

```
// Move 2020
friend book::friend_module;
public(friend) fun protected_function() {}

// Move 2024
public(package) fun protected_function_2024() {}
```

Struct Visibility

In Move 2024, structs get a visibility modifier. Currently, the only available visibility modifier is `public`.

```
// Move 2020
struct Book {}

// Move 2024
public struct Book {}
```

Method Syntax

In the new edition, functions which have a struct as the first argument are associated with the struct. This means that the function can be called using the dot notation. Methods defined in the same module with the type are automatically exported.

> Methods are automatically exported if the type is defined in the same module as the method. It is impossible to export methods for types defined in other modules. However, you can create custom aliases for methods in the module scope.

```
public fun count(c: &Counter): u64 { /* ... */ }

fun use_counter() {
    // move 2020
    let count = counter::count(&c);

    // move 2024
    let count = c.count();
}
```

Methods for Built-in Types

In Move 2024, some of the native and standard types received associated methods. For example, the `vector` type has a `to_string` method that converts the vector into a UTF8 string.

```
fun aliases() {
    // vector to string and ascii string
    let str: String = b"Hello, World!".to_string();
    let ascii: ascii::String = b"Hello, World!".to_ascii_string();

    // address to bytes
    let bytes = @0xa11ce.to_bytes();
}
```

For the full list of built-in aliases, refer to the Standard Library and Sui Framework source code.

Borrowing Operator

Some of the built-in types support borrowing operators. The borrowing operator is used to get a reference to the element at the specified index. The borrowing operator is defined as `[]`.

```
fun play_vec() {
    let v = vector[1,2,3,4];
    let first = &v[0];         // calls vector::borrow(v, 0)
    let first_mut = &mut v[0]; // calls vector::borrow_mut(v, 0)
    let first_copy = v[0];     // calls *vector::borrow(v, 0)
}
```

Types that support the borrowing operator are:

- `vector`
- `sui::vec_map::VecMap`
- `sui::table::Table`
- `sui::bag::Bag`
- `sui::object_table::ObjectTable`
- `sui::object_bag::ObjectBag`
- `sui::linked_table::LinkedTable`

To implement the borrowing operator for a custom type, you need to add a `#[syntax(index)]` attribute to the methods.

```
#[syntax(index)]
public fun borrow(c: &List<T>, key: String): &T { /* ... */ }

#[syntax(index)]
public fun borrow_mut(c: &mut List<T>, key: String): &mut T { /* ... */ }
```

Method Aliases

In Move 2024, methods can be associated with types. The alias can be defined for any type locally to the module; or publicly, if the type is defined in the same module.

```
// my_module.move
// Local: type is foreign to the module
use fun my_custom_function as vector.do_magic;

// sui-framework/kiosk/kiosk.move
// Exported: type is defined in the same module
public use fun kiosk_owner_cap_for as KioskOwnerCap.kiosk;
```

Upgradeability Practices

To talk about best practices for upgradeability, we need to first understand what can be upgraded in a package. The base premise of upgradeability is that an upgrade should not break public compatibility with the previous version. The parts of the module which can be used in dependent packages should not change their static signature. This applies to modules - a module can not be removed from a package, public structs - they can be used in function signatures and public functions - they can be called from other packages.

```
// module can not be removed from the package
module book::upgradable {
    // dependencies can be changed (if they are not used in public signatures)
    use std::string::String;
    use sui::event; // can be removed

    // public structs can not be removed and can't be changed
    public struct Book has key {
        id: UID,
        title: String,
    }

    // public structs can not be removed and can't be changed
    public struct BookCreated has copy, drop {
        /* ... */
    }

    // public functions can not be removed and their signature can never change
    // but the implementation can be changed
    public fun create_book(ctx: &mut TxContext): Book {
        create_book_internal(ctx)

        // can be removed and changed
        event::emit(BookCreated {
            /* ... */
        })
    }

    // package-visibility functions can be removed and changed
    public(package) fun create_book_package(ctx: &mut TxContext): Book {
        create_book_internal(ctx)
    }

    // entry functions can be removed and changed as long they're not public
    entry fun create_book_entry(ctx: &mut TxContext): Book {
        create_book_internal(ctx)
    }

    // private functions can be removed and changed
    fun create_book_internal(ctx: &mut TxContext): Book {
        abort 0
    }
}
```

Versioning objects

To discard previous versions of the package, the objects can be versioned. As long as the object contains a version field, and the code which uses the object expects and asserts a specific version, the code can be force-migrated to the new version. Normally, after an upgrade, admin functions can be used to update the version of the shared state, so that the new version of code can be used, and the old version aborts with a version mismatch.

```
module book::versioned_state {

    const EVersionMismatch: u64 = 0;

    const VERSION: u8 = 1;

    /// The shared state (can be owned too)
    public struct SharedState has key {
        id: UID,
        version: u8,
        /* ... */
    }

    public fun mutate(state: &mut SharedState) {
        assert!(state.version == VERSION, EVersionMismatch);
        // ...
    }
}
```

Versioning configuration with dynamic fields

There's a common pattern in Sui which allows changing the stored configuration of an object while retaining the same object signature. This is done by keeping the base object simple and versioned and adding an actual configuration object as a dynamic field. Using this *anchor* pattern, the configuration can be changed with package upgrades while keeping the same base object signature.

```
module book::versioned_config {
    use sui::vec_map::VecMap;
    use std::string::String;

    /// The base object
    public struct Config has key {
        id: UID,
        version: u16
    }

    /// The actual configuration
    public struct ConfigV1 has store {
        data: Bag,
        metadata: VecMap<String, String>
    }

    // ...
}
```

Modular architecture

This section is coming soon!

Building against Limits

To guarantee the safety and security of the network, Sui has certain limits and restrictions. These limits are in place to prevent abuse and to ensure that the network remains stable and efficient. This guide provides an overview of these limits and restrictions, and how to build your application to work within them.

The limits are defined in the protocol configuration and are enforced by the network. If any of the limits are exceeded, the transaction will either be rejected or aborted. The limits, being a part of the protocol, can only be changed through a network upgrade.

Transaction Size

The size of a transaction is limited to 128KB. This includes the size of the transaction payload, the size of the transaction signature, and the size of the transaction metadata. If a transaction exceeds this limit, it will be rejected by the network.

Object Size

The size of an object is limited to 256KB. This includes the size of the object data. If an object exceeds this limit, it will be rejected by the network. While a single object cannot bypass this limit, for more extensive storage options, one could use a combination of a base object with other attached to it using dynamic fields (eg Bag).

Single Pure Argument Size

The size of a single pure argument is limited to 16KB. A transaction argument bigger than this limit will result in execution failure. So in order to create a vector of more than ~500 addresses (given that a single address is 32 bytes), it needs to be joined dynamically either in Transaction Block or in a Move function. Standard functions like `vector::append()` can join two vectors of ~16KB resulting in a ~32KB of data as a single value.

Maximum Number of Objects created

The maximum number of objects that can be created in a single transaction is 2048. If a transaction attempts to create more than 2048 objects, it will be rejected by the network. This also affects dynamic fields, as both the key and the value are objects. So the maximum number of dynamic fields that can be created in a single transaction is 1024.

Maximum Number of Dynamic Fields created

The maximum number of dynamic fields that can be created in a single object is 1024. If an object attempts to create more than 1024 dynamic fields, it will be rejected by the network.

Maximum Number of Events

The maximum number of events that can be emitted in a single transaction is 1024. If a transaction attempts to emit more than 1024 events, it will be aborted.

Better error handling

Whenever execution encounters an abort, transaction fails and abort code is returned to the caller. Move VM returns the module name that aborted the transaction and the abort code. This behavior is not fully transparent to the caller of the transaction, especially when a single function contains multiple calls to the same function which may abort. In this case, the caller will not know which call aborted the transaction, and it will be hard to debug the issue or provide meaningful error message to the user.

```
module book::module_a {
    use book::module_b;

    public fun do_something() {
        let field_1 = module_b::get_field(1); // may abort with 0
        /* ... a lot of logic ... */
        let field_2 = module_b::get_field(2); // may abort with 0
        /* ... some more logic ... */
        let field_3 = module_b::get_field(3); // may abort with 0
    }
}
```

The example above illustrates the case when a single function contains multiple calls which may abort. If the caller of the `do_something` function receives an abort code `0`, it will be hard to understand which call to `module_b::get_field` aborted the transaction. To address this problem, there are common patterns that can be used to improve error handling.

Rule 1: Handle all possible scenarios

It is considered a good practice to provide a safe "check" function that returns a boolean value indicating whether an operation can be performed safely. If the `module_b` provides a function `has_field` that returns a boolean value indicating whether a field exists, the `do_something` function can be rewritten as follows:

```
module book::module_a {
    use book::module_b;

    const ENoField: u64 = 0;

    public fun do_something() {
        assert!(module_b::has_field(1), ENoField);
        let field_1 = module_b::get_field(1);
        /* ... */
        assert!(module_b::has_field(2), ENoField);
        let field_2 = module_b::get_field(2);
        /* ... */
        assert!(module_b::has_field(3), ENoField);
        let field_3 = module_b::get_field(3);
    }
}
```

By adding custom checks before each call to `module_b::get_field`, the developer of the `module_a` takes control over the error handling. And it allows implementing the second rule.

Rule 2: Abort with different codes

The second trick, once the abort codes are handled by the caller module, is to use different abort codes for different scenarios. This way, the caller module can provide a meaningful error message to the user. The `module_a` can be rewritten as follows:

```
module book::module_a {
    use book::module_b;

    const ENoFieldA: u64 = 0;
    const ENoFieldB: u64 = 1;
    const ENoFieldC: u64 = 2;

    public fun do_something() {
        assert!(module_b::has_field(1), ENoFieldA);
        let field_1 = module_b::get_field(1);
        /* ... */
        assert!(module_b::has_field(2), ENoFieldB);
        let field_2 = module_b::get_field(2);
        /* ... */
        assert!(module_b::has_field(3), ENoFieldC);
        let field_3 = module_b::get_field(3);
    }
}
```

Now, the caller module can provide a meaningful error message to the user. If the caller receives an abort code `0`, it can be translated to "Field 1 does not exist". If the caller receives an abort code `1`, it can be translated to "Field 2 does not exist". And so on.

Rule 3: Return bool instead of assert

A developer is often tempted to add a public function that would assert all the conditions and abort the execution. However, it is a better practice to create a function that returns a boolean value instead. This way, the caller module can handle the error and provide a meaningful error message to the user.

```
module book::some_app_assert {

    const ENotAuthorized: u64 = 0;

    public fun do_a() {
        assert_is_authorized();
        // ...
    }

    public fun do_b() {
        assert_is_authorized();
        // ...
    }

    /// Don't do this
    public fun assert_is_authorized() {
        assert!(/* some condition */ true, ENotAuthorized);
    }
}
```

This module can be rewritten as follows:

```
module book::some_app {
    const ENotAuthorized: u64 = 0;

    public fun do_a() {
        assert!(is_authorized(), ENotAuthorized);
        // ...
    }

    public fun do_b() {
        assert!(is_authorized(), ENotAuthorized);
        // ...
    }

    public fun is_authorized(): bool {
        /* some condition */ true
    }

    // a private function can still be used to avoid code duplication for a case
    // when the same condition with the same abort code is used in multiple places
    fun assert_is_authorized() {
        assert!(is_authorized(), ENotAuthorized);
    }
}
```

Utilizing these three rules will make the error handling more transparent to the caller of the transaction, and it will allow other developers to use custom abort codes in their modules.

Coding Conventions

Naming

Module

1. Module names should be in `snake_case`.
2. Module names should be descriptive and should not be too long.

```
module book::conventions { /* ... */ }
module book::common_pracices { /* ... */ }
```

Constant

1. Constants should be in `SCREAMING_SNAKE_CASE`.
2. Error constants should be in `EPascalCase`

```
const MAX_PRICE: u64 = 1000;
const EInvalidInput: u64 = 0;
```

Function

1. Function names should be in `snake_case`.
2. Function names should be descriptive.

```
public fun add(a: u64, b: u64): u64 { a + b }
public fun create_if_not_exists() { /* ... */ }
```

Struct

1. Struct names should be in `PascalCase`.
2. Struct fields should be in `snake_case`.
3. Capabilities should be suffixed with `Cap`.

```
public struct Hero has key {
    id: UID
    value: u64,
    another_value: u64,
}

public struct AdminCap has key { id: UID }
```

Struct Method

1. Struct methods should be in `snake_case`.
2. If there's multiple structs with the same method, the method should be prefixed with the struct name. In this case, an alias can be added to the method using `use fun`.

```
public fun value(h: &Hero): u64 { h.value }

public use fun hero_health as Hero.health;
public fun hero_health(h: &Hero): u64 { h.another_value }

public use fun boar_health as Boar.health;
public fun boar_health(b: &Boar): u64 { b.another_value }
```

Appendix A: Glossary

- Fast Path - term used to describe a transaction that does not involve shared objects, and can be executed without the need for consensus.
- Parallel Execution - term used to describe the ability of the Sui runtime to execute transactions in parallel, including the ones that involve shared objects.
- Internal Type - type that is defined within the module. Fields of this type can not be accessed from outside the module, and, in case of "key"-only abilities, can not be used in `public_*` transfer functions.

Abilities

- key - ability that allows the struct to be used as a key in the storage. On Sui, the key ability marks an object and requires the first field to be a `id: UID`.
- store - ability that allows the struct to be stored inside other objects. This ability relaxes restrictions applied to internal structs, allowing `public_*` transfer functions to accept them as arguments. It also enables the object to be stored as a dynamic field.
- copy - ability that allows the struct to be copied. On Sui, the `copy` ability conflicts with the `key` ability, and can not be used together with it.
- drop - ability that allows the struct to be ignored or discarded. On Sui, the `drop` ability cannot be used together with the `key` ability, as objects are not allowed to be ignored.

Appendix B: Reserved Addresses

Reserved addresses are special addresses that have a specific purpose on Sui. They stay the same between environments and are used for specific native operations.

- `0x1` - address of the Standard Library (alias `std`)
- `0x2` - address of the Sui Framework (alias `sui`)
- `0x5` - address of the `SuiSystem` object
- `0x6` - address of the system `Clock` object
- `0x8` - address of the system `Random` object
- `0x403` - address of the `DenyList` system object

Appendix C: Publications

This section lists publications related to Move and Sui.

- The Move Borrow Checker by Sam Blackshear, John Mitchell, Todd Nowacki, Shaz Qadeer.
- Resources: A Safe Language Abstraction for Money by Sam Blackshear, David L. Dill, Shaz Qadeer, Clark W. Barrett, John C. Mitchell, Oded Padon, Yoni Zohar.
- Robust Safety for Move by Marco Patrignani, Sam Blackshear

Appendix D: Contributing

To contribute to this book, please, submit a pull request to the GitHub repository. The repository contains the source files for the book, written in mdBook format.

Appendix E: Acknowledgements

The Rust Book has been a great inspiration for this book. I am personally grateful to the authors of the book, Steve Klabnik and Carol Nichols, for their work, as I have learned a lot from it. This book is a small tribute to their work and an attempt to bring a similar learning experience to the Move community.