Move Conventions

This guide outlines recommended conventions and best practices for writing Move smart contracts on Sui. Following these guidelines helps create more maintainable, secure, and composable code that aligns with ecosystem standards.

While these conventions are recommendations rather than strict rules, they represent patterns that have proven effective across many Sui projects. They help create consistency across the ecosystem and make code easier to understand and maintain.

A Sui package consists of:

For this reason, the Move.lock file should always be part of the package (don't add it to the .gitignore file). Use the automated address management instead of the old published-at field in the manifest file.

Optionally, you can add a tests directory to contain the tests for the package and an examples directory to provide use cases for the package. Neither directory is uploaded on chain when you publish the package.

In your package manifest, the package name should be in PascalCase: name = "MyPackage" . Ideally, the named address representing the package should be the same as the package name, but in snake_case: my_package = 0x0 .

Modules are the main building blocks of your Move code. They are used to organize and encapsulate related functionality. Design your modules around one object or data structure. A variant structure should have its own module to avoid complexity and bugs.

Module declarations don't need to use brackets anymore and the compiler provides default use statements for widely used modules, so you don't need to declare all of them.

Structure your code using comments to create sections for your Move code files. Structure your titles using ═══ on either side of the title.

Here, "public functions" are the functions modifying state, "view functions" are often on-chain getters or off-chain helpers. The latter are not necessary because you can query objects to read their data. The init function should be the first function in the module, if it exists.

Try to sort your functions by their purpose and according to the user flow to improve readability. You can also use explicit function names like admin_set_fees to make it clear what the function does.

Ideally, test functions should only consist of [test_only] helpers for the actual tests that are located in the tests directory.

Group imports by dependency, for example:

Adhering to naming conventions in your code helps readability and ultimately makes your codebase easier to maintain. The following sections outline the key naming conventions to follow when writing Move code.

Constants should be uppercase and formatted as snake case. Errors are specific constants that use PascalCase and start with an E. Make them descriptive.

Always declare struct abilities in this order: key , copy , drop , store .

Do not use 'potato' in the name of structs. The lack of abilities define it as a potato pattern.

Structs support positional fields that can be used for simple wrappers, dynamic field keys, or as tuples.

Use the Event suffix to name structs that emit events.

The following functions follow standard CRUD (Create, Read, Update, Delete) naming conventions:

Declare generics using single letter names or full names. By convention, developers use T and U for generic types, but you can use a more descriptive name if it is not confusing with other types. Always prioritize readability.

The following section covers common patterns and best practices specific to Move development on Sui, including object ownership models and function design principles.

Library modules that share objects should provide two functions: one to instantiate and return the object, and another one to share it. It allows the caller to pass it to other functions and run custom functionality before sharing it.

Keep your functions pure to maintain composability. Do not use transfer::transfer or transfer::public_transfer inside core functions, except in specific cases where the object is not transferable and shouldn't be modified.

Pass the Coin object by value with the exact right amount directly to improve transaction readability from the frontend.

To maintain composability, use capability objects instead of arrays of addresses for access control.

If your dApp data has a one to one relationship, it's best to use owned objects.

In admin-gated functions, the first parameter should be the capability. It helps the autocomplete with user types.

There is nothing more pleasant than a well-written and well-documented codebase. While some argue that clean code is self-documenting, well-documented code is self-explanatory.

Document your code by explaining functions and structs in simple terms using the /// syntax (doc comment). If you want to add technical insights for developers that might use your code, use the // syntax (regular comment).

Use field comments to describe the properties of your structs. In complex functions, you can also describe the parameters and return values.

Create a README.md file in the root of the package. Include a description of the package, the purpose of the package, and how to use it.

Organization principles

A Sui package consists of:

For this reason, the Move.lock file should always be part of the package (don't add it to the .gitignore file). Use the automated address management instead of the old published-at field in the manifest file.

Optionally, you can add a tests directory to contain the tests for the package and an examples directory to provide use cases for the package. Neither directory is uploaded on chain when you publish the package.

In your package manifest, the package name should be in PascalCase: name = "MyPackage" . Ideally, the named address representing the package should be the same as the package name, but in snake_case: my_package = 0x0 .

Modules are the main building blocks of your Move code. They are used to organize and encapsulate related functionality. Design your modules around one object or data structure. A variant structure should have its own module to avoid complexity and bugs.

Module declarations don't need to use brackets anymore and the compiler provides default use statements for widely used modules, so you don't need to declare all of them.

Structure your code using comments to create sections for your Move code files. Structure your titles using ═══ on either side of the title.

Here, "public functions" are the functions modifying state, "view functions" are often on-chain getters or off-chain helpers. The latter are not necessary because you can query objects to read their data. The init function should be the first function in the module, if it exists.

Try to sort your functions by their purpose and according to the user flow to improve readability. You can also use explicit function names like admin_set_fees to make it clear what the function does.

Ideally, test functions should only consist of [test_only] helpers for the actual tests that are located in the tests directory.

Group imports by dependency, for example:

Adhering to naming conventions in your code helps readability and ultimately makes your codebase easier to maintain. The following sections outline the key naming conventions to follow when writing Move code.

Constants should be uppercase and formatted as snake case. Errors are specific constants that use PascalCase and start with an E. Make them descriptive.

Always declare struct abilities in this order: key , copy , drop , store .

Do not use 'potato' in the name of structs. The lack of abilities define it as a potato pattern.

Structs support positional fields that can be used for simple wrappers, dynamic field keys, or as tuples.

Use the Event suffix to name structs that emit events.

The following functions follow standard CRUD (Create, Read, Update, Delete) naming conventions:

Declare generics using single letter names or full names. By convention, developers use T and U for generic types, but you can use a more descriptive name if it is not confusing with other types. Always prioritize readability.

The following section covers common patterns and best practices specific to Move development on Sui, including object ownership models and function design principles.

Library modules that share objects should provide two functions: one to instantiate and return the object, and another one to share it. It allows the caller to pass it to other functions and run custom functionality before sharing it.

Keep your functions pure to maintain composability. Do not use transfer::transfer or transfer::public_transfer inside core functions, except in specific cases where the object is not transferable and shouldn't be modified.

Pass the Coin object by value with the exact right amount directly to improve transaction readability from the frontend.

To maintain composability, use capability objects instead of arrays of addresses for access control.

If your dApp data has a one to one relationship, it's best to use owned objects.

In admin-gated functions, the first parameter should be the capability. It helps the autocomplete with user types.

There is nothing more pleasant than a well-written and well-documented codebase. While some argue that clean code is self-documenting, well-documented code is self-explanatory.

Document your code by explaining functions and structs in simple terms using the /// syntax (doc comment). If you want to add technical insights for developers that might use your code, use the // syntax (regular comment).

Use field comments to describe the properties of your structs. In complex functions, you can also describe the parameters and return values.

Create a README.md file in the root of the package. Include a description of the package, the purpose of the package, and how to use it.

Naming conventions

Adhering to naming conventions in your code helps readability and ultimately makes your codebase easier to maintain. The following sections outline the key naming conventions to follow when writing Move code.

Constants should be uppercase and formatted as snake case. Errors are specific constants that use PascalCase and start with an E. Make them descriptive.

Always declare struct abilities in this order: key , copy , drop , store .

Do not use 'potato' in the name of structs. The lack of abilities define it as a potato pattern.

Structs support positional fields that can be used for simple wrappers, dynamic field keys, or as tuples.

Use the Event suffix to name structs that emit events.

The following functions follow standard CRUD (Create, Read, Update, Delete) naming conventions:

Declare generics using single letter names or full names. By convention, developers use T and U for generic types, but you can use a more descriptive name if it is not confusing with other types. Always prioritize readability.

The following section covers common patterns and best practices specific to Move development on Sui, including object ownership models and function design principles.

Library modules that share objects should provide two functions: one to instantiate and return the object, and another one to share it. It allows the caller to pass it to other functions and run custom functionality before sharing it.

Keep your functions pure to maintain composability. Do not use transfer::transfer or transfer::public_transfer inside core functions, except in specific cases where the object is not transferable and shouldn't be modified.

Pass the Coin object by value with the exact right amount directly to improve transaction readability from the frontend.

To maintain composability, use capability objects instead of arrays of addresses for access control.

If your dApp data has a one to one relationship, it's best to use owned objects.

In admin-gated functions, the first parameter should be the capability. It helps the autocomplete with user types.

There is nothing more pleasant than a well-written and well-documented codebase. While some argue that clean code is self-documenting, well-documented code is self-explanatory.

Document your code by explaining functions and structs in simple terms using the /// syntax (doc comment). If you want to add technical insights for developers that might use your code, use the // syntax (regular comment).

Use field comments to describe the properties of your structs. In complex functions, you can also describe the parameters and return values.

Create a README.md file in the root of the package. Include a description of the package, the purpose of the package, and how to use it.

Code Structure

The following section covers common patterns and best practices specific to Move development on Sui, including object ownership models and function design principles.

Library modules that share objects should provide two functions: one to instantiate and return the object, and another one to share it. It allows the caller to pass it to other functions and run custom functionality before sharing it.

Keep your functions pure to maintain composability. Do not use transfer::transfer or transfer::public_transfer inside core functions, except in specific cases where the object is not transferable and shouldn't be modified.

Pass the Coin object by value with the exact right amount directly to improve transaction readability from the frontend.

To maintain composability, use capability objects instead of arrays of addresses for access control.

If your dApp data has a one to one relationship, it's best to use owned objects.

In admin-gated functions, the first parameter should be the capability. It helps the autocomplete with user types.

There is nothing more pleasant than a well-written and well-documented codebase. While some argue that clean code is self-documenting, well-documented code is self-explanatory.

Document your code by explaining functions and structs in simple terms using the /// syntax (doc comment). If you want to add technical insights for developers that might use your code, use the // syntax (regular comment).

Use field comments to describe the properties of your structs. In complex functions, you can also describe the parameters and return values.

Create a README.md file in the root of the package. Include a description of the package, the purpose of the package, and how to use it.

Documentation

There is nothing more pleasant than a well-written and well-documented codebase. While some argue that clean code is self-documenting, well-documented code is self-explanatory.

Document your code by explaining functions and structs in simple terms using the /// syntax (doc comment). If you want to add technical insights for developers that might use your code, use the // syntax (regular comment).

Use field comments to describe the properties of your structs. In complex functions, you can also describe the parameters and return values.

Create a README.md file in the root of the package. Include a description of the package, the purpose of the package, and how to use it.