# The Move Book

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.

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 :: .

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

Note: Any item (struct, function, constant, etc.) that you want to import from another module must be marked with the public (or public(package) - see visibility modifiers ) keyword to make it accessible outside its defining module. For example, the Character struct and the new function in module_one are marked public so they can be used in module_two .

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.

Imports can be grouped into a single use statement using curly braces {} . This allows for cleaner and more organized code when importing multiple members from the same module or package.

Importing function names is 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.

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.

Move packages can depend on other packages; the dependencies are listed in the Package Manifest file called Move.toml .

Package dependencies are defined in the Package Manifest as follows:

The dependencies section contains an entry for each package dependency. The key of the entry is the name of the package ( Example or Local in the example), 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 qa package directory.

If you add a dependency, all of its dependencies also become available to your package.

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

Starting with version 1.45 of the sui CLI, the system packages are automatically included as dependencies for all packages if they are not present in Move.toml . Therefore, MoveStdlib , Sui , System , Bridge , and Deepbook are all available without an explicit import.

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.

## 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 :: .

```bash // 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.

```bash // 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(); } ```

Note: Any item (struct, function, constant, etc.) that you want to import from another module must be marked with the public (or public(package) - see visibility modifiers ) keyword to make it accessible outside its defining module. For example, the Character struct and the new function in module_one are marked public so they can be used in module_two .

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.

```bash 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() } ```

Imports can be grouped into a single use statement using curly braces {} . This allows for cleaner and more organized code when importing multiple members from the same module or package.

```bash 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() } ```

Importing function names is 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.

```bash 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() } ```

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.

```bash 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() } ```

Move packages can depend on other packages; the dependencies are listed in the Package Manifest file called Move.toml .

Package dependencies are defined in the Package Manifest as follows:

```
bash [dependencies] Example = { git = "https://github.com/Example/example.git", subdir = "path/to/package", rev = "v1.2.3" } Local = { local = "../my_other_package" }
```

The dependencies section contains an entry for each package dependency. The key of the entry is the name of the package ( Example or Local in the example), 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 qa package directory.

If you add a dependency, all of its dependencies also become available to your package.

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

Starting with version 1.45 of the sui CLI, the system packages are automatically included as dependencies for all packages if they are not present in Move.toml . Therefore, MoveStdlib , Sui , System , Bridge , and Deepbook are all available without an explicit import.

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

```bash 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
```

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.

## 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.

```bash 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() } ```

Imports can be grouped into a single use statement using curly braces {} . This allows for cleaner and more organized code when importing multiple members from the same module or package.

```bash 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() } ```

Importing function names is 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.

```bash 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() } ```

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.

```bash 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() } ```

Move packages can depend on other packages; the dependencies are listed in the Package Manifest file called Move.toml .

Package dependencies are defined in the Package Manifest as follows:

```
bash [dependencies] Example = { git = "https://github.com/Example/example.git", subdir =
"path/to/package", rev = "v1.2.3" } Local = { local = "../my_other_package" }
```

The dependencies section contains an entry for each package dependency. The key of the entry is the name of the package ( Example or Local in the example), 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 qa package directory.

If you add a dependency, all of its dependencies also become available to your package.

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

Starting with version 1.45 of the sui CLI, the system packages are automatically included as dependencies for all packages if they are not present in Move.toml . Therefore, MoveStdlib , Sui , System , Bridge , and Deepbook are all available without an explicit import.

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

```bash 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
```

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.

## Grouping Imports

Imports can be grouped into a single use statement using curly braces {} . This allows for cleaner and more organized code when importing multiple members from the same module or package.

```bash 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() } ```

Importing function names is 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.

```bash 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() } ```

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.

```bash 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() } ```

Move packages can depend on other packages; the dependencies are listed in the Package Manifest file called Move.toml .

Package dependencies are defined in the Package Manifest as follows:

```
bash [dependencies] Example = { git = "https://github.com/Example/example.git", subdir = "path/to/package", rev = "v1.2.3" } Local = { local = "../my_other_package" }
```

The dependencies section contains an entry for each package dependency. The key of the entry is the name of the package ( Example or Local in the example), 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 qa package directory.

If you add a dependency, all of its dependencies also become available to your package.

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

Starting with version 1.45 of the sui CLI, the system packages are automatically included as dependencies for all packages if they are not present in Move.toml . Therefore, MoveStdlib , Sui , System , Bridge , and Deepbook are all available without an explicit import.

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

```bash 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 ```

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.

## 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.
```

```bash 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() } ```

Move packages can depend on other packages; the dependencies are listed in the Package Manifest file called Move.toml .

Package dependencies are defined in the Package Manifest as follows:

```
bash [dependencies] Example = { git = "https://github.com/Example/example.git", subdir =
"path/to/package", rev = "v1.2.3" } Local = { local = "../my_other_package" }
```

The dependencies section contains an entry for each package dependency. The key of the entry is the name of the package ( Example or Local in the example), 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 qa package directory.

If you add a dependency, all of its dependencies also become available to your package.

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

Starting with version 1.45 of the sui CLI, the system packages are automatically included as dependencies for all packages if they are not present in Move.toml . Therefore, MoveStdlib , Sui , System , Bridge , and Deepbook are all available without an explicit import.

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

```bash 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
```

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.

## Adding an External Dependency

Move packages can depend on other packages; the dependencies are listed in the Package Manifest file called Move.toml .

Package dependencies are defined in the Package Manifest as follows:

```
bash [dependencies] Example = { git = "https://github.com/Example/example.git", subdir =
"path/to/package", rev = "v1.2.3" } Local = { local = "../my_other_package" }
```

The dependencies section contains an entry for each package dependency. The key of the entry is the name of the package ( Example or Local in the example), 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 qa package directory.

If you add a dependency, all of its dependencies also become available to your package.

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

Starting with version 1.45 of the sui CLI, the system packages are automatically included as dependencies for all packages if they are not present in Move.toml . Therefore, MoveStdlib , Sui , System , Bridge , and Deepbook are all available without an explicit import.
```

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

```bash 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
```

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.

## Importing a Module from Another Package

Normally, packages define their addresses in the [addresses] section. You can use aliases instead of full addresses. For example, instead of using 0x2::coin to reference the Sui coin module, you can use sui::coin . The sui alias is defined in the Sui Framework package's manifest. Similarly, the std alias is defined in the Standard Library package and can be used instead of 0x1 to access standard library modules.

To import a module from another package, 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 :: .

```bash 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
```

Note: Module address names come from the [addresses] section of the manifest file ( Move.toml ), not the names used in the [dependencies] section.