The Move Book

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

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.

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

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

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 optional, but has to be a u64 value or an #[error] (see below for more information).

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 similar to other constants and do not have any special handling. However, they are commonly used to improve code readability and make abort scenarios easier to understand.

Move 2024 introduces a special type of error constant, marked with the #[error] attribute. This attribute allows the error constant to be of type vector and can be used to store an error message.

Abort

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

```bash 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 .

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 optional, but has to be a u64 value or an #[error] (see below for more information).

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

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

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 similar to other constants and do not have any special handling. However, they are commonly used to improve code readability and make abort scenarios easier to understand.

```bash /// 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);

/* ... */

} ```
```

Move 2024 introduces a special type of error constant, marked with the #[error] attribute. This attribute allows the error constant to be of type vector and can be used to store an error message.

```bash
# [error]

const ENotAuthorized: vector = b"The user is not authorized to perform this action";

# [error]

const EValueTooLow: vector = b"The value is too low, it should be at least 10";

/// Performs an action on behalf of the user. public fun update_value(user: &mut User, value: u64) { assert!(user.is_authorized, ENotAuthorized); assert!(value >= 10, EValueTooLow);

user.value = value;

} ```

## 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 optional, but has to be a u64 value or an #[error] (see below for more information).

``bash // aborts ifuser_has_accessisfalse` with abort code 0 assert!(user_has_access, 0);

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

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 similar to other constants and do not have any special handling. However, they are commonly used to improve code readability and make abort scenarios easier to understand.

```bash /// 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);

/* ... */

} ```

Move 2024 introduces a special type of error constant, marked with the #[error] attribute. This attribute allows the error constant to be of type vector and can be used to store an error message.

```bash
# [error]

const ENotAuthorized: vector = b"The user is not authorized to perform this action";

# [error]

const EValueTooLow: vector = b"The value is too low, it should be at least 10";

/// Performs an action on behalf of the user. public fun update_value(user: &mut User, value: u64) { assert!(user.is_authorized,
```

ENotAuthorized); assert!(value >= 10, EValueTooLow);

```
user.value = value;
```

} ```

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 similar to other constants and do not have any special handling. However, they are commonly used to improve code readability and make abort scenarios easier to understand.

```bash /// 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);

/* ... */

} ```

Move 2024 introduces a special type of error constant, marked with the #[error] attribute. This attribute allows the error constant to be of type vector and can be used to store an error message.

```bash

# [error]

const ENotAuthorized: vector = b"The user is not authorized to perform this action";

# [error]

const EValueTooLow: vector = b"The value is too low, it should be at least 10";

/// Performs an action on behalf of the user. public fun update_value(user: &mut User, value: u64) { assert!(user.is_authorized, ENotAuthorized); assert!(value >= 10, EValueTooLow);

```
user.value = value;
```

} ```

Error messages

Move 2024 introduces a special type of error constant, marked with the #[error] attribute. This attribute allows the error constant to be of type vector and can be used to store an error message.

```bash

# [error]

const ENotAuthorized: vector = b"The user is not authorized to perform this action";

# [error]

const EValueTooLow: vector = b"The value is too low, it should be at least 10";

/// Performs an action on behalf of the user. public fun update_value(user: &mut User, value: u64) { assert!(user.is_authorized, ENotAuthorized); assert!(value >= 10, EValueTooLow);
```

```
user.value = value;

} ```

## Further reading
```