Introduction

Clarity of Mind is both an introductory as well as a reference book for the Clarity smart contract language. Clarity is developed as a joint effort of Hiro PBC, Algorand, and various other stakeholders, that originally targets the Stacks blockchain. A significant innovation in the field of smart contract development, Clarity allows you to write more safe and secure smart contracts. The language optimises for readability and predictability and is purpose-built for developers working applications with high-stakes transactions.

Target audience

This book is accessible for both beginners and experienced developers alike. Concepts are gradually introduced in a logical and steady pace. Nonetheless, the chapters lend themselves rather well to being read in a different order. More experienced developers might get the most benefit by jumping to the chapters that interest them most. **If you like to learn by example, then you should go straight to the chapter on Using Clarinet.**

It is assumed that you have a basic understanding of programming and the underlying logical concepts. The first chapter covers the general syntax of Clarity but it does not delve into what programming *itself* is all about. If this is what you are looking for, then you might have a more difficult time working through this book unless you have an (undiscovered) natural affinity for such topics. Do not let that dissuade you though, find an introductory programming book and press on! The straightforward design of Clarity makes it a great first language to pick up.

What are smart contracts & blockchains?

Clarity is a language for writing *smart contracts* that run on a *blockchain*. Before we can discuss the what smart contracts are, we must understand what a blockchain is. There is a wealth of information available on the topic, as well as what different kinds of blockchains exist. We will therefore only briefly touch upon the concept in a more generalised sense. A blockchain can be thought of as a special kind of *immutable distributed database*:

- It is *distributed* in the sense that all participants can get a complete copy of the database. Once you install a Bitcoin node, for example, it will start downloading the entire blockchain from the network. There is no single party that hosts or manages the entire blockchain on behalf of the users. Everyone that plays by the rules can participate in the network.
- *Immutability* comes from the fact that once information is added, it cannot (feasibly[^1]) be changed. The network rules prevent any one actor from making changes to data that has already been inserted. Blockchains are unique in that they leverage cryptography to ensure that all participants reach a consensus on the true state of the data. It allows the network to function without the need of trusting a central authority and it is why it is referred to as a "trustless system". It is where the "crypto" in "cryptocurrency" comes from.

The blockchain is therefore a kind of secure and resilient public record. Changes are made by submitting a properly formatted and digitally signed *transaction* onto the network. The different nodes in the network collect these transactions, assess their validity, and will then include them in the next block based on some conditions. For most blockchains, the nodes that create blocks are reimbursed by the transaction senders. Transactions come attached with a small fee that the node can claim when it includes the transaction in a block.

Smart contracts are programs that run on top of blockchains and can utilise their unique properties. Effectively, it means that you can build an application that does not run on a single system but is instead executed and verified across a distributed network. The same nodes that process transactions will execute the smart contract code and include the result in the next block. Users can *deploy* smart contracts—the act of adding a new one to the chain—and call into existing ones by sending a transaction. Because executing some code is more resource intensive, the transaction fee will go up depending on the complexity of the code.

What makes Clarity different

The number of smart contract languages grows by the year. Choosing a first language can be challenging, especially for a beginner. The choice is largely dictated by the ecosystem you are interested in, although some languages are applicable to more than just one platform. Each language has its own upsides and downsides and it is out of the scope of this book to look at all of them. Instead, we will focus on what sets Clarity apart and why it is a prime choice if you require the utmost security and transparency.

One of the core precepts of Clarity is that it is secure by design. The design process was guided by examining common pitfalls, mistakes, and vulnerabilities in the field of smart contract engineering as a whole. There are countless real world examples of where developer failure led to the loss or theft of vast amounts of tokens. To name two big ones: an issue that has become known as the *Parity bug* led to the irreparable loss of millions of dollars worth of Ethereum. Second, the hacking of The DAO (a *"Decentralised Autonomous Organisation"*) caused financial damage so great that the Ethereum Foundation decided to issue a contentious hard fork that undid the theft. These and many other mistakes could have been prevented in the design of the language itself.

Clarity is *interpreted*, not *compiled*

Clarity code is interpreted and committed to the chain exactly as written. Solidity and other languages are compiled to byte-code before it is submitted to the chain. The danger of compiled smart contract languages is two-fold: first, a compiler adds a layer of complexity. A bug in the compiler may lead to different byte-code than was intended and thus carries the risk of introducing a vulnerability. Second, byte-code is not human-readable, which makes it very hard to verify what the smart contract is actually doing. Ask yourself, *would you sign a contract you cannot read?* If your answer is no, then why should it be any different for smart contracts?[^2] With Clarity, what you see is what you get.

Clarity is *decidable*

A decidable language has the property that from the code itself, you can know with certainty what the program will do. This avoids issues like the halting problem. With Clarity you know for sure that given any input, the program will halt in a finite number of steps. In simple terms: it is guaranteed that program execution will end. Decidability also allows for complete static analysis of the call graph so you get an accurate picture of the exact cost before execution. There is no way for a Clarity call to *"run out of gas"* in the middle of the call. If you are unsure what this means, let it not worry you for now. The serious advantage of decidability will become more apparent over time.

Clarity does not permit *reentrancy*

Reentrancy is a situation where one smart contract calls into another, which then calls back into the first contract—the call *"re-enters"* the same logic. It may allow an attacker to trigger multiple token withdrawals before the contract has had a chance to update its internal balance sheet. Clarity's design considers reentrancy an anti-feature and disallows it on the language level.

Clarity guards against *overflow* and *underflows*

Overflows and underflows happen when a calculation results in a number that is either too large or too small to be stored, respectively. These events throw smart contracts into disarray and may intentionally be triggered in poorly written contracts by attackers. Usually this leads to a situation where the contract is either frozen or drained of tokens. Overflows and underflows of any kind automatically cause a transaction to be aborted in Clarity.

Support for custom tokens is built-in

Issuance of custom fungible and non-fungible tokens is a popular use-case for smart contracts. Custom token features are built into the Clarity language. Developers do not need to worry about creating an internal balance sheet, managing supply, and emitting token events. Creating custom tokens is covered in depth in later chapters.

On Stacks, transactions are secured by *post conditions*

In order to further safeguard user tokens, post conditions can be attached to transactions to assert the chain state has changed in a certain way once the transaction has completed. For example, a user calling into a smart contract may attach a post condition that states that after the call completes, exactly 500 STX should have been transferred from one address to another. If the post condition check fails, then the entire transaction is reverted. Since custom token support is built right into Clarity, post conditions can also be used to guard any other token in the same way.

Returned responses cannot be left unchecked

Public contract calls must return a so-called *response* that indicates success or failure. Any contract that calls another contract is required to properly handle the response. Clarity contracts that fail to do so are invalid and cannot be deployed on the network. Other languages like Solidity permit the use of low level calls without requiring the return value to be checked. For example, a token transfer can fail silently if the developer forgets to check the result. In Clarity it is not possible to ignore errors, although that obviously does prevent buggy error handling on behalf of the developer. Responses and error handling are covered extensively in the chapters on functions and control flow.

Composition over inheritance

Clarity adopts a composition over inheritance. It means that Clarity smart contracts do not inherit from one another like you see in languages like Solidity. Developers instead define traits which are then implemented by different smart contracts. It allows contracts to conform to different interfaces with greater flexibility. There is no need to worry about complex class trees and contracts with implicit inherited behaviour.

Access to the base chain: Bitcoin

Clarity smart contracts can read the state of the Bitcoin base chain. It means you can use Bitcoin transactions as a trigger in your smart contracts! Clarity also features a number of built-in functions to verify secp256k1 signatures and recover keys.

[^1]: The note on feasibility was added for correctness. Blockchains are designed to be highly resistant to change but there have been cases of rewrites. Weaker chains can be susceptible to so-called "51% attacks" that allow a powerful miner to rewrite that chain's history. On the other hand, influential factions may mandate a node upgrade that changes chain history by means of a hard fork. For example, the Ethereum Foundation "solved" the problem of the DAO hack with a hard fork.

[^2]: Although this characteristic makes it a lot easier to read Clarity smart contracts, it does not necessarily make it easy. A good grasp of Clarity is still required, but one can argue that it is still a lot better than a conventional paper contract written in legalese, which quite honestly can be considered a language in its own right. The actual difference is that Clarity permits only one interpretation, something that definitely cannot be said for legalese.

Getting started

Getting started with Clarity is an order of magnitude easier than some other smart contract languages. Although the tooling is still a bit experimental, you can already do some pretty great things. Want to test a quick snippet? We have a self-contained REPL (*"Read-Evaluate-Print Loop"*). Want to spin up a local simulated chain without all the hassles of running a full node? *Clarinet* has got you covered. How about generating a full-fledged client for your smart contract? *Clarigen* can make a difference there.

The book will cover installing the official Clarity REPL and Clarinet for end to end smart contract development. A lot of fully functioning Clarity snippets and example contracts will be provided as you work through the chapters. Learning by example is really the best way to master a new skill. The later chapters will challenge you to build a few common smart contracts by yourself using Clarinet. Extra attention will be given to best practices and proper testing.

You are in for a treat if you are reading this book online. Almost all Clarity code snippets contained herein are *fully interactive*. When you see a play button, you can click it to execute the Clarity code and be shown the results (or mistakes) in your browser. But the fun does not stop there. You can even play around with these snippets by editing them directly. If you are reading this on a different medium, then keep the Clarity REPL by your side.

Installing Clarinet

What is Clarinet?

Clarinet is a Clarity runtime packaged as a command line tool, designed to facilitate smart contract understanding, development, testing and deployment. Clarinet consists of a Clarity REPL and a testing harness, which, when used together allow you to rapidly develop and test a Clarity smart contract, with the need to deploy the contract to a local devnet or testnet.

Clarity is a decidable smart contract language that optimises for predictability and security, designed for the Stacks blockchain. Smart contracts allow developers to encode essential business logic on a blockchain.

Install on macOS (Homebrew)

This process relies on the macOS package manager called Homebrew. Using the `brew` command you can easily add powerful functionality to your mac, but first we have to install it.

To get started, launch your Terminal (/Applications/Utilities/Terminal) application. Terminal is a versatile command line system that comes with every Mac computer.

If you do not already have XCode installed, it's best to first install the command line tools as these will be used by homebrew:

```
xcode-select --install
```

When the XCode is complete, proceed with Homebrew installation:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Now you can test your installation to ensure you have installed brew correctly, simply type:

```
% brew --version
Homebrew 3.6.3
```

Finally, you can now install the Clarinet on your mac:

```
brew install clarinet
```

Just follow the terminal prompts if necessary.

Install on Windows

The easiest way to install Clarinet on Windows is to use the MSI installer, that can be downloaded from the releases page.

Clarinet is also available on Winget, the package manager that Microsoft started including in the latest Windows updates:

```
winget install clarinet
```

Install from a pre-built binary

To install Clarinet from pre-built binaries, download the latest release from the releases page. Unzip the binary, then copy it to a location that is already in your path, such as `/usr/local/bin`.

```
# note: you can change the v0.27.0 with version that are available in the releases page.
wget -nv https://github.com/hirosystems/clarinet/releases/download/v0.27.0/clarinet-linux-x64-glibc.tar.gz -O clarinet-linux-x64.tar.gz
tar -xf clarinet-linux-x64.tar.gz
chmod +x ./clarinet
mv ./clarinet /usr/local/bin
```

On MacOS, you may get security errors when trying to run the pre-compiled binary. You can resolve the security warning with this command:

```
xattr -d com.apple.quarantine /path/to/downloaded/clarinet/binary
```

Install from source using Cargo

Prerequisites

Install Rust for access to `cargo`, the Rust package manager.

On Debian and Ubuntu-based distributions, please install the following packages before building Clarinet.

```
sudo apt install build-essential pkg-config libssl-dev
```

Build Clarinet

You can build Clarinet from source using Cargo with the following commands:

```
git clone https://github.com/hirosystems/clarinet.git --recursive
cd clarinet
cargo clarinet-install
```

By default, you will be in our development branch, `develop`, with code that has not been released yet. If you plan to submit any changes to the code, then this is the right branch for you. If you just want the latest stable version, switch to the main branch:

```
git checkout main
```

If you have previously checked out the source, ensure you have the latest code (including submodules) before building using:

```
git pull
git submodule update --recursive
```

Verify Clarinet installation

You can verify Clarinet is installed properly by running `clarinet --version` in your favourite Terminal emulator.

```
% clarinet --version
clarinet-cli 2.6.0
```

More information about Clarinet: https://github.com/hirosystems/clarinet/blob/develop/README.md

Clarity basics

Clarity features *LISP-like* syntax. That means that you will see a lot of parentheses. Inside these parentheses you will find symbols, phrases, values, and more parentheses. People that are new to LISP-like languages may feel intimidated as it looks very foreign compared to both natural languages as many other programming languages.

A way to conceptualise Clarity code is to think of lists inside lists. (The technical term for these kinds of expressions is "S-expression".) Expressions do not differentiate between data and code, that makes things a bit easier! Here is an example of a Clarity expression (hint: click the *play* button or copy and paste it into the REPL):

```
(+ 4 5)
```

Did you guess correctly what the expression evaluates to? Even though it might look a little strange for those who are not used to *Polish notation* in math, it is easy to make a reasonable assumption. The `+` symbol defines an operation (in this case addition), and the `4` and the `5` are inputs. The end result: `9`.

Such expressions always follow the same basic pattern: an opening round brace `(`, followed by a *function name*, optionally followed by a number of *input parameters*, and closed by a closing round brace `)`. The different parts inside the braces are separated by whitespace.

The `+` symbol in the example above really has no special significance. It is just a function name. Here is another example:

```
(concat "Hello" " World!")
```

Is it starting to make sense? `concat` is the function name and it is provided with two strings as inputs. "Concat" is short for *concatenate*. Thus: it glues two strings together to form the classic `"Hello World!"`.

You might be able to intuit how more complex expressions go together at this point. Take the following calculation for example: `4 × (15 + 10)`. In Clarity, it looks like this:

```
(* 4 (+ 15 10))
```

The above is still pretty legible, but it is obvious that expressions with multiple levels of nesting become hard to read. Whitespace is only used to delineate symbols. It means that we can use tabs, spaces, and newlines to make Clarity code more readable.

```
(*
    (+ 2 3)
    (- 6 1)
)
```

As you start writing more Clarity code, you will discover which formatting is most comfortable for you. Looking at how other developers format their code also helps.

Finally, it is possible to add freeform commentary to your code. Never be sparse with your comments, especially if you are working on a project with multiple people! Comments are prefixed by a double

semi-colon (`;;`) and can appear on their own line or after an expression. They are also useful to temporarily deactivate some code during the development process.

```
;; A comment on its own line.

(+ 1 2) ;; The result will be 3.

;; The following line is not evaluated:
;; (+ 4 5)
```

Remember that Clarity contracts are committed to the chain as-is. Comments therefore increase the contract size and anything *creative* you put in there will be viewable by anyone inspecting the code.

```

```

Types

(If you are looking for *type signatures*, see the later chapter on variables.)

An important concept for many programming languages are the so-called *types*. A type defines what kind of information can be stored inside a *variable*. If you imagine a variable to be a container that can hold something, the type defines what kind of thing it holds.

Types are strictly enforced and cannot mix in Clarity. Type safety is key because type errors (mixing two different types) can lead to unexpected errors with grave consequences. Clarity therefore rejects any kind of type mixing. Here is an example:

```
(+ 2 u3)
```

The expression above results in an error:

> *Analysis error: expecting expression of type **'int'**, found **'uint'** (+ 2 u3)*

Before we can properly answer the question of *which* type it was expecting *where*, let us take a look at the different types in Clarity. Types fall in three categories: *primitives*, *sequences*, and *composites*.

- Primitives are the basic building blocks for the language. They include numbers and boolean values (true and false).
- Sequences hold multiple values in order.
- Composites are complex types that are made up of other types.

Primitives

Primitive types are the most basic components. These are: *signed and unsigned integers*, *booleans*, and *principals*.

Signed integers

`int`, short for *(signed) integer*. These are 128 bits numbers that can either be positive or negative. The minimum value is -2^127 and the maximum value is 2^127 - 1. Some examples: `0`, `5000`, `-45`.

Unsigned integers

`uint`, short for *unsigned integer*. These are 128 bits numbers that can only be positive. The minimum value is therefore 0 and the maximum value is 2^128 - 1. **Unsigned integers are always prefixed by the character `u`.** Some examples: `u0`, `u40935094534`.

Clarity has many built-in functions that accept either signed or unsigned integers.

Addition:

```
(+ u2 u3)
```

Subtraction:

```
(- 5 10)
```

Multiplication:

```
(* u2 u16)
```

Division:

```
(/ 100 4)
```

As you might have noticed by now, integers are always whole numbers—there are no decimal points. It is something to keep in mind when writing your code.

```
(/ u10 u3)
```

If you punch the above into a calculator, you will likely get `3.3333333333...`. Not with integers! The above expression evaluates to `u3`, the decimals are dropped.

There are many more functions that take integers as inputs. We will get to the rest later in the book.

Booleans

`bool`, short for *boolean*. A boolean value is either `true` or `false`. They are used to check if a certain condition is met or unmet (true or false). Some built-in functions that accept booleans:

`not` (inverts a boolean):

```
(not true)
```

`and` (returns `true` if all inputs are `true`):

```
(and true true true)
```

`or` (returns `true` if at least one input is `true`):

```
(or false true false)
```

Principals

A principal is a special type in Clarity and represents a Stacks address on the blockchain. It is a unique identifier you can roughly equate to an email address or bank account number—although definitely not the same! You might have also heard the term *wallet address* as well. Clarity admits two different kinds of principals: *standard principals* and *contract principals*. Standard principals are backed by a corresponding private key whilst contract principals point to a smart contract. Principals follow a specific structure and always start with the characters `SP` for the Stacks mainnet and `ST` for the testnet and mocknet[^1].

A literal principal value is prefixed by a single quote (`'`) in Clarity. Notice there is no closing single quote.

```
'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE
```

Contract principals are a compound of the standard principal that deployed the contract and the contract name, delimited by a dot:

```
'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.my-awesome-contract
```

You will use the principal type often when writing Clarity. It is used to check who is calling the contract, recording information about different principals, function calls across contracts, and much more.

To retrieve the current STX balance of a principal, we can pass it to the `stx-get-balance` function.

```
(stx-get-balance 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE)
```

Both kinds of principals can hold tokens, we can thus also check the balance of a contract.

```
(stx-get-balance 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.my-contract)
```

Zero balances are a little boring, so let us send some some STX to a principal:

```
(stx-transfer? u500 tx-sender 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE)
```

Knowing about primitives, and the fact that types can never mix, it is now clear why the example in the previous section does not work. Since the first number is a *signed integer* and the next one is an

unsigned integer—notice the `u` —the analyser rejects the code as invalid. We should provide it with two signed or two unsigned integers instead.

Incorrect:

```
(+ 2 u3)
```

Correct:

```
(+ u2 u3)
```

[^1]: More on the different kinds of networks in a later chapter.

Sequences

Sequences hold a sequence of data, as the name implies. Clarity provides three different kinds of sequences: *buffers*, *strings*, and *lists*.

Buffers

Buffers are unstructured data of a fixed maximum length. They always start with the prefix `0x` followed by a hexadecimal string. Each byte is thus represented by two so-called hexits.

```
0x68656c6c6f21
```

The buffer above spells out *"hello!"*. (Copy and paste it to this page to verify.)

Strings

A string is a sequence of characters. These can be defined as ASCII strings or UTF-8 strings. ASCII strings may only contain basic Latin characters whilst UTF-8 strings can contain fun stuff like emoji. Both strings are enclosed in double-quotes (`"`) but UTF-8 strings are also prefixed by a `u` . Just like buffers, strings always have a fixed maximum length in Clarity.

ASCII:

```
"This is an ASCII string"
```

UTF-8:

```
u"And this is an UTF-8 string \u{1f601}"
```

You can use strings to pass names and messages.

Lists

Lists are sequences of fixed length that contain another type. Since types cannot mix, a list can only contain items of the same type. Take this list of *signed integers* for example:

```
(list 4 8 15 16 23 42)
```

As you can see, a list is constructed using the `list` function. Here is a list of ASCII strings:

```
(list "Hello" "World" "!")
```

And just for completeness, here is a list that is invalid due to mixed types:

```
(list u5 10 "hello") ;; This list is invalid.
```

Lists are very useful and make it a lot easier to perform actions in bulk. (For example, sending some tokens to a list of people.) You can *iterate* over a list using the `map` or `fold` functions.

`map` applies an input function to each element and returns a new list with the updated values. The `not` function inverts a boolean (`true` becomes `false` and `false` becomes `true`). We can thus

invert a list of booleans like this:

```
(map not (list true true false false))
```

`fold` applies an input function to each element of the list *and* the output value of the previous application. It also takes an initial value to use for the second input for the first element. The returned result is the last value returned by the final application. This function is also commonly called *reduce*, because it reduces a list to a single value. We can use `fold` to sum numbers in a list by applying the `+` (addition) function with an initial value of `u0`:

```
(fold + (list u1 u2 u3) u0)
```

The snippet above can be expanded to the following:

```
(+ u3 (+ u2 (+ u1 u0)))
```

Working with sequences

Length

Sequences always have a specific length, which we can retrieve using the `len` function.

A buffer (remember that each byte is represented as two hexits):

```
(len 0x68656c6c6f21)
```

A string:

```
(len "How long is this string?")
```

And a list:

```
(len (list 4 8 15 16 23 42))
```

Retrieving elements

They also allow you to extract elements at a particular index. The following takes the *fourth* element from the list. (Counting starts at 0.)

```
(element-at? (list 4 8 15 16 23 42) u3)
```

You can also do the reverse and find the index of a particular item in a sequence. We can search the list to see if it contains the value `23`.

```
(index-of? (list 4 8 15 16 23 42) 23)
```

And we get `(some u4)`, indicating there is a value of `23` at index four. The attentive might now be wondering, what is this *"some"*? Read on and all will be revealed in the next section.

Composite types

These are more complex types that contain a number of other types. Composites make it a lot easier to create larger smart contracts.

Optionals

The type system in Clarity does not allow for empty values. It means that a boolean is always either `true` or `false`, and an integer always contains a number. But sometimes you want to be able to express a variable that could have *some* value, or *nothing*. For this you use the `optional` type. This type wraps a different type and can either be `none` or a value of that type. The optional type is very powerful and the tooling will perform checks to make sure they are handled properly in the code. Let us look at a few examples.

Wrapping a `uint`:

```
(some u5)
```

An ASCII string:

```
(some "An optional containing a string.")
```

Or even a principal:

```
(some 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE)
```

Nothing is represented by the keyword `none`:

```
none
```

Functions that might or might not return a value tend to return an optional type. As we saw in the previous section, both `element-at` and `index-of` returned a `(some ...)`. It is because for some inputs, no matching value can be found. We can take the same list but this time try to retrieve an element at an index *larger* than the total size of the list. We see that it results in a `none` value.

```
(element-at? (list 4 8 15 16 23 42) u5000)
```

When writing smart contracts, the developer must handle cases where `(some ...)` is returned differently from when `none` is returned.

In order to access the value contained within an optional, you have to *unwrap* it.

```
(unwrap-panic (some u10))
```

Trying to unwrap a `none` will result in an error because there is nothing to unwrap. The *"panic"* in `unwrap-panic` should give that away. (We use a trick with `if` to provide information about the type of the optional value, otherwise Clarity can't handle the expression.)

```
(unwrap-panic (if true none (some u10)))
```

Later chapters on error handling and defining custom functions will dive into how to deal with such errors and what effects they have on the chain state.

Tuples

Tuples are records that hold multiple values in named fields. Each field has its own type, making it very useful to pass along structured data in one go. Tuples have their own special formatting and use curly braces.

```
{
    id: u5, ;; a uint
    username: "ClarityIsAwesome", ;; an ASCI string
    address: 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE ;; and a principal
}
```

The members inside tuples are unordered. You retrieve them by name and cannot iterate over them. A specific member can be read using the `get` function.

```
(get username { id: 5, username: "ClarityIsAwesome" })
```

Tuples, like other values, are immutable once defined. It means that they cannot be changed. You can, however, merge two tuples together to form a new tuple. Merging is done from left to right and will overwrite values with the same key.

```
(merge
    {id: 5, score: 10, username: "ClarityIsAwesome"}
    {score: 20, winner: true}
)
```

The above expression will result in a tuple with both keys merged and the score set to `20`.

Since the merge function returns an *entirely new* tuple, member types can in fact be overwritten by a later tuple in the sequence.

```
(merge
    {id: u6, score: 10}
    {score: u50}
)
```

Responses

A response is a composite type that wraps another type just like an optional. What is different, however, is that a response type includes an indication of whether a specific action was successful or a failure. Responses have special effects when returned by public functions. We will cover those effects in the chapter on functions.

A response takes the concrete form of either `(ok ...)` or `(err ...)`. Wrapping a value in a concrete response is straightforward:

```
(ok true)
```

Developers usually come up with their own rules to indicate error status. You could for example use unsigned integers to represent a specific error code.

```
(err u5) ;; something went wrong.
```

There are no explicit rules on which types you should wrap for your responses. Standards are currently being proposed and we will touch upon a few in the chapter on *Stacks Improvement Proposals* (SIPs).

Responses can be unwrapped in the same way as optional types:

```
(unwrap-panic (ok true))
```

Although not necessary, private functions and read-only functions may also return a response type.

Keywords

Keywords are special terms that have an assigned meaning. We already came across a few keywords in the previous chapters: `true`, `false`, and `none`. There are a few others that demand extra attention.

block-height

Reflects the current block height of the Stacks blockchain as an unsigned integer. If we imagine the chain tip to be at height 5, we can read that number at any point in our code.

```
block-height
```

burn-block-height

Reflects the current block height of the underlying burn blockchain (in this case Bitcoin) as an unsigned integer.

```
burn-block-height
```

tx-sender

Contains the principal that sent the transaction. It can be used to validate the principal that is calling into a public function.

```
tx-sender
```

Note that it is possible for the `tx-sender` to be a contract principal if the special function `as-contract` was used to shift the sending context.

```
(as-contract tx-sender)
```

contract-caller

Contains the principal that called the function. It can be a standard principal or contract principal. If the contract is called via a signed transaction directly, then `tx-sender` and `contract-caller` will be equal. If the contract calls another contract in turn, then `contract-caller` will be equal to the previous contract in the chain.

```
contract-caller
```

Storing data

Smart contracts have their own private storage space. You can define different types of data members to use throughout your smart contract. These data members are committed to the chain and thus persist across transactions. For example, a first transaction can change a data member after which a second one reads the updated value.

All data members have to be defined on the top level of the contract and are identified by a unique name. No new data members can be introduced after the contract has been deployed. Clarity permits three different kinds of storage: *constants*, *variables*, and *data maps*.

- Constant values are unchangeable, defined on the top level of the contract. They are useful to define the contract owner, error codes, and other static values.
- Variables have an initial value and can be changed by means of future contract calls. One variable contains exactly one value of a predefined type.
- Maps are collections of data identified by other data. Think of variables where the variable names themselves are values. They are used to relate one value to another; for example, relating specific principals to unsigned integers to keep track of scores.

Although data members are _private_—meaning, only the current contract can use them—it does not mean that they are *hidden*. **Anything on the blockchain is inherently public so data members should never be used to store sensitive information like passwords or private keys.** The value of any data member can be extracted from the chain state effortlessly.

Constants

Constants are data members that cannot be changed once they are defined (hence the name *constant*). They are useful to define concrete configuration values, error codes, and more. The general form to define a constant looks like this:

```
(define-constant constant-name expression)
```

The `constant-name` can be any valid phrase and the `expression` any valid Clarity code.

The expression passed into the definition is evaluated at contract launch in the order that it is supplied. If one constant thus depends on another, they need to be defined in the right order.

```
(define-constant my-constant "This is a constant value")

(define-constant my-second-constant
    (concat my-constant " that depends on another")
)

(print my-constant)
(print my-second-constant)
```

A common pattern that you will come across is that of defining a constant to store the principal that deployed the contract:

```
(define-constant contract-owner tx-sender)

(print contract-owner)
```

Constants are also useful to give return values and errors meaningful names.

```
(define-constant err-something-failed (err u100))

;; And then use err-something-failed instead of (err u100) later in the code.
(print err-something-failed)
```

If you are curious about the `print` function by this point: it allows us to print something to the screen in the REPL. Interestingly enough, `print` actually triggers a custom event and can be used to emit any valid data structure. Custom applications scanning the chain could pick these events up and process them further. The Stacks genesis block contains a simple smart contract with a `print` expression to encode a nice message on the blockchain until the end of time:

> *... to be a completely separate network and separate block chain, yet share CPU power with Bitcoin`` - Satoshi Nakamoto*

The `print` function is used throughout the book to be able to show intermediary values.

Variables

Variables are data members that can be changed over time. They are only modifiable by the current smart contract. Variables have a predefined type and an initial value.

```
(define-data-var var-name var-type initial-value)
```

Where the `var-type` is a *type signature* and `initial-value` a valid value for the specified type. Although you can name a variable pretty much anything, you should be mindful of the built-in keywords. Do not use keywords as variable names.

Variables can be read using the function `var-get` and changed using `var-set`.

```
;; Define an unsigned integer data var with an initial value of u0.
(define-data-var my-number uint u0)

;; Print the initial value.
(print (var-get my-number))

;; Change the value.
(var-set my-number u5000)

;; Print the new value.
(print (var-get my-number))
```

Notice the `uint`? That is the type signature.

Type signatures

The chapter on types covered how to express a value of a specific type. Type signatures, on the other hand, define the admitted type for a variable or function argument. Let us take a look at what the signatures look like.

Type	Signature
Signed integer	`int`
Unsigned integer	`uint`
Boolean	`bool`
Principal	`principal`
Buffer	`(buff max-len)`, where `max-len` is a number defining the maximum length.
ASCII string	`(string-ascii max-len)`, where `max-len` is a number defining the maximum length.
UTF-8 string	`(string-utf8 max-len)`, where `max-len` is a number defining the maximum length.
List	`(list max-len element-type)`, where `max-len` is a number defining the maximum length and `element-type` a type signature. Example: `(list 10 principal)`.

Optional	`(optional some-type)`, where `some-type` is a type signature. Example: `(optional principal)`.
Tuple	`{key1: entry-type, key2: entry-type}`, where `entry-type` is a type signature. Every key can have its own type. Example: `{sender: principal, amount: uint}`.
Response	`(response ok-type err-type)`, where `ok-type` is the type of returned `ok` values and `err-type` is the type of returned `err` values. Example: `(response bool uint)`.

We can see that some types indicate a *maximum length*. It goes without saying that the length is strictly enforced. Passing a value that is too long will result in an analysis error. Try changing the following example by making the `"This works."` string too long.

```
(define-data-var message (string-ascii 15) "This works.")
```

Like other kinds of definition statements, `define-data-var` may only be used at the top level of a smart contract definition; that is, you cannot put a define statement in the middle of a function body.

Remember that whitespace can be used to make your code more readable. If you are defining a complicated tuple type, simply space it out:

```
(define-data-var high-score
    ;; Tuple type definition:
    {
        score: uint,
        who: (optional principal),
        at-height: uint
    }
    ;; Tuple value:
    {
        score: u0,
        who: none,
        at-height: u0
    }
)

;; Print the initial value.
(print (var-get high-score))

;; Change the value.
(var-set high-score
    {score: u10, who: (some tx-sender), at-height: block-height}
)

;; Print the new value.
(print (var-get high-score))
```

Maps

Data maps are so-called *hash tables*. It is a kind of data structure that allows you to map keys to specific values. Unlike tuple keys, data map keys are not hard-coded names. They are represented as a specific concrete values. You should use maps if you want to relate data to other data.

A map is defined using `define-map`:

```
(define-map map-name key-type value-type)
```

Both `key-type` and `value-type` can be any valid type signature, although tuples are normally used because of their versatility.

```
;; A map that creates a principal => uint relation.
(define-map balances principal uint)

;; Set the "balance" of the tx-sender to u500.
(map-set balances tx-sender u500)

;; Retrieve the balance.
(print (map-get? balances tx-sender))
```

Let us take a look at how we can use a map to store and read basic orders by ID. We will use an unsigned integer for the key type and a tuple for the value type. These fictional orders will hold a principal and an amount.

```
(define-map orders uint {maker: principal, amount: uint})

;; Set two orders.
(map-set orders u0 {maker: tx-sender, amount: u50})
(map-set orders u1 {maker: tx-sender, amount: u120})

;; retrieve order with ID u1.
(print (map-get? orders u1))
```

It is important to know that maps are not iterable. In other words, you cannot loop through a map and retrieve all values. The only way to access a value in a map is by specifying the right key.

Keys can be as simple or complex as you want them to be:

```
(define-map highest-bids
    {listing-id: uint, asset: (optional principal)}
    {bid-id: uint}
)

(map-set highest-bids {listing-id: u5, asset: none} {bid-id: u20})
```

Whilst tuples make the code more readable, remember that Clarity is interpreted. Using a tuple as a key incurs a higher execution cost than using a primitive type. If your tuple key has only one member, consider using the member type as the map key type directly.

Set and insert

The `map-set` function will overwrite existing values whilst `map-insert` will do nothing and return `false` if the specified key already exists. Entries may also be deleted using `map-delete`.

```
(define-map scores principal uint)

;; Insert a value.
(map-insert scores tx-sender u100)

;; This second insert will do nothing because the key already exists.
(map-insert scores tx-sender u200)

;; The score for tx-sender will be u100.
(print (map-get? scores tx-sender))

;; Delete the entry for tx-sender.
(map-delete scores tx-sender)

;; Will return none because the entry got deleted.
(print (map-get? scores tx-sender))
```

Reading from a map might fail

What we have seen from the previous examples is that `map-get?` returns an optional type. The reason is that reading from a map fails if the provided key does not exist. When that happens, `map-get?` returns a `none`. It also means that if you wish to use the retrieved value, you will have to unwrap it in most cases.

```
;; A map that creates a string-ascii => uint relation.
(define-map names (string-ascii 34) principal)

;; Point the name "Clarity" to the tx-sender.
(map-set names "Clarity" tx-sender)

;; Retrieve the principal related to the name "Clarity".
(print (map-get? names "Clarity"))

;; Retrieve the principal for a key that does not exist. It will return `none`.
(print (map-get? names "bogus"))

;; Unwrap a value:
(print (unwrap-panic (map-get? names "Clarity")))
```

The chapter that discusses the different unwrap flavours goes more into what unwrapping means.

Functions

Functions are portions of code that may take some input and produce an output. They are used to subdivide your program code into logical components.

Clarity features a plethora of built-in functions. We have already seen a few of these in the chapters leading up to this one. Providing a full reference for all of them is out of the scope of this book (for now), but you can refer to the official Clarity Language Reference to find a detailed list. Instead, we will focus on defining custom functions and examining what different kinds of functions exist; namely, *public functions*, *private functions*, and *read-only functions*.

- Public functions can be called externally. That means that another standard principal or contract principal can invoke the function. Public function calls require sending a transaction. The sender thus need to pay transaction fees.
- Private functions can only be called by the current contract—there is no outside access. (Although the source code can obviously still be inspected by reading the blockchain.)
- Read-only functions can be called externally but may not change the chain state. Sending a transaction is not necessary to call a read-only function.

Defining a custom function takes the following general form:

```
(define-public function-signature function-body)
```

If you count the input parameters for the `define-public` function, you will see that there are only two: the *function signature* and the *function body*.

Function signature

The function signature defines the *name* of the function and any *input parameters*. The input parameters themselves contain type signatures.

The pattern of function signatures is such:

```
(function-name (param1-name param1-type) (param2-name param2-type) ...)
```

It makes more sense if we look at a few examples. Here is a *"hello world"* function that takes no parameters:

```
(define-public (hello-world)
    (ok "Hello World!")
)

(print (hello-world))
```

A multiplication function that takes two parameters:

```
(define-public (multiply (a uint) (b uint))
    (ok (* a b))
)

(print (multiply u5 u10))
```

A *"hello [name]"* function that takes one string parameter:

```
(define-public (hello (name (string-ascii 30)))
    (ok (concat "Hello " name))
)

(print (hello "Clarity"))
```

Function body

The expressions that define functions take exactly one expression for the function body. The function body is what is executed when the function is called. If the body is limited to one expression only, then how can you create more complex functions that require multiple expression? For this, a special form exists. The variadic `begin` function takes an arbitrary amount of inputs and will return the result of the last expression.

```
(begin 3 4 5)
```

A multi-expression function may therefore be put together as follows:

```
(define-public (print-twice (first (string-ascii 40)) (second (string-ascii 40)))
    (begin
        (print first)
        (print second)
        (ok true)
    )
)

(print-twice "Hello world!" "Multiple prints!")
```

Continue to the next section to understand why that `ok` is there at the end.

Public functions

Public functions are callable from the outside by both standard principals and contract principals. Contracts that feature any interactivity will need at least one public function. The fact of them being callable does not imply that the *underlying functionality* is public. The developer can include assertions to make sure that only specific contract callers or inputs are valid.

Public functions ***must*** return a response type value. If the function returns an `ok` type, then the function call is considered valid, and any changes made to the blockchain state will materialise. It means that state changes such as updating variables or transferring tokens will ***only*** be committed to the chain if the contract call that triggered these changes returns an `ok`.

The effects of returning an `ok` or an `err` are illustrated by the example below. It is a basic function that takes an unsigned integer as an input and will return an `ok` if it is even or an `err` if it is uneven. It will also increment a variable called `even-values` at the start of the function. To check if the number is even, we calculate the remainder of a division by two and check that it is equal to zero using the `is-eq` function (*n mod 2 should equal 0*).

```
(define-data-var even-values uint u0)

(define-public (count-even (number uint))
    (begin
        ;; increment the "event-values" variable by one.
        (var-set even-values (+ (var-get even-values) u1))

        ;; check if the input number is even (number mod 2 equals 0).
        (if (is-eq (mod number u2) u0)
            (ok "the number is even")
            (err "the number is uneven")
        )
    )
)

;; Call count-even two times.
(print (count-even u4))
(print (count-even u7))

;; Will this return u1 or u2?
(print (var-get even-values))
```

Did you notice how the final print expression returned `u1` and not `u2`, even though the `count-even` function is called twice? If you are used to programming in a different language, for example JavaScript, then it might strike you as odd: the `even-values` variable is updated at the start of the function so it seems intuitive that the number should increment on every function call.

Edit the example above and try going through the following iterations:

- Replace the uneven number `u7` with an even number like `u8`. Does the printout of `even-values` contain `u1` or `u2`?
- Change the `err` type in the `if` expression to an `ok`. What is the value of `even-values` then?

What happens is that the entire public function call is rolled back or *reverted* as soon as it returns an `err` value. It is as if the function call never happened! It therefore does not matter if you update the variable at the start or end of the function.

Understanding responses and how they can affect the chain state is key to being a successful smart contract developer. The chapter on error handling will go in greater detail on how to guard your functions and the exact control flow.

```
(define-public (sum-three)

)

(print (sum-three u3 u5 u9))
```

Private functions

Private functions are defined in the same manner as public functions. The difference is that they can only be called by the current contract. They cannot be called from other smart contracts, nor can they be called directly by sending a transaction. Private functions are useful to create utility or helper functions to cut down on code repetition. If you find yourself repeating the similar expressions in multiple locations, then it is worth considering turning those expressions into a separate private function.

The contract bellow allows only the contract owner to update the `recipients` map via two public functions. Instead of having to repeat the `tx-sender` check, it is *abstracted away* to its own private function called `is-valid-caller`.

```
(define-constant contract-owner tx-sender)

;; Try removing the contract-owner constant above and using a different
;; one to see the example calls error out:
;; (define-constant contract-owner 'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2)

(define-constant err-invalid-caller (err u1))

(define-map recipients principal uint)

(define-private (is-valid-caller)
    (is-eq contract-owner tx-sender)
)

(define-public (add-recipient (recipient principal) (amount uint))
    (if (is-valid-caller)
        (ok (map-set recipients recipient amount))
        err-invalid-caller
    )
)

(define-public (delete-recipient (recipient principal))
    (if (is-valid-caller)
        (ok (map-delete recipients recipient))
        err-invalid-caller
    )
)

;; Two example calls to the public functions:
(print (add-recipient 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK u500))
(print (delete-recipient 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK))
```

Another good reason to define private functions is to reduce overall function complexity. Large public functions can be harder to maintain and are more prone to developer error. Splitting such functions up into a public function and a number of smaller private functions can alleviate these issues.

Private functions may return any type, including responses, although returning an `ok` or an `err` will have no effect on the materialised state of the chain.

```
(define-constant err-invalid-caller (err u1))

(define-map authorised-callers principal bool)
(define-map recipients principal bool)

(map-set recipients tx-sender true)
(map-set authorised-callers 'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2 true)

(define-private (is-valid-caller (caller principal))
    ;; Implement.
)

(define-public (delete-recipient (recipient principal))
    (if (is-valid-caller tx-sender)
        (ok (map-delete recipients recipient))
        err-invalid-caller
    )
)

(print (delete-recipient tx-sender))
```

Read-only functions

Read-only functions can be called by the contract itself, as well as from the outside. They can return any type, just like private functions.

As the name implies, *read-only* functions may only perform read operations. You can read from data variables and maps but you *cannot* write to them. Read-only functions can also be purely *functional;* that is to say, calculate some result based on an input and return it. This is fine:

```
(define-read-only (add (a uint) (b uint))
    (+ a b)
)

(print (add u5 u10))
```

And so is this:

```
(define-data-var counter uint u0)

(define-read-only (get-counter-value)
    (var-get counter)
)

(print (get-counter-value))
```

But this one ***not*** so much:

```
(define-data-var counter uint u0)

(define-read-only (increment-counter)
    (var-set (+ (var-get counter) u1))
)

(print (increment-counter))
```

As you can see, the analysis tells us it detected a writing operation inside a read-only function:

> *Analysis error: expecting read-only statements, detected a writing operation (define-data-var counter uint u0)*

Not to worry though, there is no way to mess that up. If analysis fails the contract is rendered invalid, which means it cannot be deployed on the network.

One thing that makes read-only functions very interesting is that they can be called *without* actually sending a transaction! By using read-only functions, you can read the contract state for your application without requiring your users to pay transaction fees. Stacks.js and the Web Wallet Extension have support for calling read-only functions built-in. You can try it yourself right now with the Stacks Sandbox. Find a contract with a read-only function and call it directly. Completely free!

```
(define-map counters principal uint)

(map-set counters 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK u5)
```

```
(map-set counters 'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2 u10)

(define-read-only (get-counter-of (who principal))
    ;; Implement.
)

;; These exist:
(print (get-counter-of 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK))
(print (get-counter-of 'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2))

;; This one does not:
(print (get-counter-of 'ST21HMSJATHZ888PD0S0SSTWP4J61TCRJYEVQ0STB))
```

Control flow & error handling

Errors, would it not be great if smart contracts were error free? Error handling in Clarity follows a very straightforward paradigm. We have already seen that returning an `err` from a public function triggers a revert. That is pretty significant, but understanding the *control flow* of your smart contract is even more important.

What is control flow? Put simply, it is the order in which expressions are evaluated. The functions introduced up until this point allow following a simple left-to-right rule. `begin` perfectly illustrates this:

```
(begin
    (print "First")
    (print "Second")
    (print "Third")
)
```

The first print expression is evaluated first, the second after that, and so on. But there are a few functions that actually influence the control flow. These are aptly named *control flow functions*. If understanding responses is key to becoming a successful smart contract developer, then understanding control functions is key to becoming a great smart contract developer. The names of the control flow functions are: `asserts!`, `try!`, `unwrap!`, `unwrap-err!`, `unwrap-panic`, and `unwrap-err-panic`.

Up until now, we used `if` expressions to either return an `ok` or an `err` response. Recall the return portion of the `count-even` function in the chapter on public function:

```
(if (is-eq (mod number u2) u0)
    (ok "the number is even")
    (err "the number is uneven")
)
```

One can argue that the structure is still decently legible, but imagine needing multiple conditionals that all return a different error code on failure. You will quickly end up with constructs that no sane developer can easily understand! Control flow functions are absolutely necessary to produce legible code once your contracts become more complex. They allow you to create short-circuits that immediately return a value from a function, ending execution early and thus skipping over any expressions that might have come after.

Another useful thing to understand with control flow functions is the difference between functions that end in an exclamation point (such as `unwrap!`), and those that do not (such as `unwrap-panic`). Those that end in an exclamation point allow for arbitrary early returns from a function. Those that do not terminate execution altogether and throw a runtime error.

asserts!

The `asserts!` function takes two parameters, the first being a boolean expression and the second a so-called *throw value*. If the boolean expression evaluates to `true`, then `asserts!` returns `true` and execution continues as expected, but if the expression evaluates to `false` then `asserts!` will return the throw value and *exit the current control flow*.

That sounds complicated, so let us take a look at some examples. Keep in mind that the basic form for `asserts!` as described looks like this:

```
(asserts! boolean-expression throw-value)
```

The following assertion is said to *pass*, as the boolean expression evaluates to `true`.

```
(asserts! true (err "failed"))
```

The next one is said to *fail*, as the boolean expression evaluates to `false`.

```
(asserts! false (err "failed"))
```

Notice how somewhere in that error message we find the `(err "failed")`? Let us make it more clear with a test function. The test function takes a boolean input value and asserts its truthiness. For a throw value we will use an `err` and the final expression will return an `ok`.

```
(define-public (asserts-example (input bool))
    (begin
        (asserts! input (err "the assertion failed"))
        (ok "end of the function")
    )
)

(print (asserts-example true))
(print (asserts-example false))
```

The first print gives us the `ok` as seen at the end of the `begin` expression. Nothing too strange there. But the second call gives us the `err` throw value!

Even though the `begin` function gives us the result of the final expression *under normal circumstances*, the `asserts!` control function has the ability to *override that behaviour and exit the current flow*. When `asserts!` fails, it short-circuits and returns the throw value from the function immediately. It makes `asserts!` really useful for creating guards by _asserting_—hence the name—that certain values are what you expect them to be.

Remember that `is-valid-caller` function in the chapter on private functions? The example used an `if` function to only allow the action if the `tx-sender` was equal to the principal that deployed the contract. Let us now rewrite that contract to use `asserts!` instead:

```
(define-constant contract-owner tx-sender)

;; Try removing the contract-owner constant above and using a different
```

```
;; one to see the example calls error out:
;; (define-constant contract-owner 'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2)

(define-constant err-invalid-caller (err u1))

(define-map recipients principal uint)

(define-private (is-valid-caller)
    (is-eq contract-owner tx-sender)
)

(define-public (add-recipient (recipient principal) (amount uint))
    (begin
        ;; Assert the tx-sender is valid.
        (asserts! (is-valid-caller) err-invalid-caller)
        (ok (map-set recipients recipient amount))
    )
)

(define-public (delete-recipient (recipient principal))
    (begin
        ;; Assert the tx-sender is valid.
        (asserts! (is-valid-caller) err-invalid-caller)
        (ok (map-delete recipients recipient))
    )
)

;; Two example calls to the public functions:
(print (add-recipient 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK u500))
(print (delete-recipient 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK))
```

That looks a lot more readable. If you disagree, wait until you get to the chapter on best practices.

try!

The `try!` function takes an `optional` or a `response` type and will attempt to *unwrap* it. Unwrapping is the act of extracting the inner value and returning it. Take the following example:

```
(try! (some "wrapped string"))
```

It will unwrap the `some` and return the inner `"wrapped string"`.

`try!` can only successfully unwrap `some` and `ok` values. If it receives a `none` or an `err`, it will return the input value and exit the current control flow. In other words:

- If it receives a `none`, it returns a `none` and exits.
- If it receives an `err`, it returns that `err` and exits. It *does not* unwrap the value inside!

The following test function allows us to experiment with this behaviour. It takes a response type as input which is passed to `try!`. We will then call the function with an `ok` and an `err` and print the results.

```
(define-public (try-example (input (response uint uint)))
    (begin
        (try! input)
        (ok "end of the function")
    )
)

(print (try-example (ok u1)))
(print (try-example (err u2)))
```

The first print gives us the `(ok "end of the function")` as seen at the end of the `begin` expression. But the second call that passes the `err` gives us back the original `(err u2)`. The `try!` function therefore allows you to *propagate* an error that occurs in a sub call, as we will see in the section on intermediary responses.

Unwrap flavours

The other unwrap functions are all variations that exit the current control flow in a slightly different manner.

`unwrap!` takes an `optional` or `response` as the first input and a throw value as the second input. It follows the same unwrapping behaviour of `try!` , but instead of propagating the the `none` or the `err` it will return the the throw value instead.

```
(unwrap! (some "wrapped string") (err "unwrap failed"))
```

`unwrap-panic` takes a single input which is either an `optional` or `response` . If it fails to unwrap the input, it throws a runtime error and exits the current flow.

```
(unwrap-panic (ok true))
```

`unwrap-err!` takes a `response` input and a throw value. If the first input is an `err` , it will return the wrapped value. Otherwise it returns the throw value and exit. It is the counterpart to `unwrap!` .

```
(unwrap-err! (err false) (err "unwrap failed"))
```

`unwrap-err-panic` is the counterpart to `unwrap-panic` , the input is unwrapped if it is an `err` , or else a runtime error is thrown.

```
(unwrap-err-panic (err false))
```

You should ideally not use the `-panic` variants unless you absolutely have to, because they confer no meaningful information when they fail. A transaction will revert with a vague "runtime error" and users as well as developers are left to figure out exactly what went wrong.

Unpacking assignments

The unwrap functions are particularly useful when assigning local variables using `let` . You can unwrap and assign a value if it exists or exit if it does not. It makes working with maps and lists a breeze.

```
;; Some error constants
(define-constant err-unknown-listing (err u100))
(define-constant err-not-the-maker (err u101))

;; Define an example map called listings, identified by a uint.
(define-map listings
    {id: uint}
    {name: (string-ascii 50), maker: principal}
)

;; Insert some sample data
(map-set listings {id: u1} {name: "First Listing", maker: tx-sender})
(map-set listings {id: u2} {name: "Second Listing", maker: tx-sender})
```

```
;; Simple function to get a listing
(define-read-only (get-listing (id uint))
    (map-get? listings {id: id})
)

;; Update name function that only the maker for a specific listing
;; can call.
(define-public (update-name (id uint) (new-name (string-ascii 50)))
    (let
        (
            ;; The magic happens here.
            (listing (unwrap! (get-listing id) err-unknown-listing))
        )
        (asserts! (is-eq tx-sender (get maker listing)) err-not-the-maker)
        (map-set listings {id: id} (merge listing {name: new-name}))
        (ok true)
    )
)

;; Two test calls
(print (update-name u1 "New name!"))
(print (update-name u9999 "Nonexistent listing..."))
```

Find the comment `;; The magic happens here.` inside the `update-name` function and study the next line closely. Here is what happens:

- It defines a variable called `listing`.
- The value will be equal to the *unwrapped result* of the `get-listing` function.
- `get-listing` returns the result of `map-get?`, which is either `some` listing or `none`.
- If the unwrap fails, `unwap!` exits with `err-unknown-listing`.

The first test call will therefore succeed and return `(ok true)` while the second call will error out with `(err u100)` (`err-unknown-listing`).

Response checking

We already learned that the response returned by a public function determines whether or not a state change materialises on the chain. This does not just hold true for the initial contract call, but also for subsequent calls. If a standard principal calls into *Contract A*, which in turn calls into *Contract B*, then the response of *Contract B* will only influence the internal state of *Contract B*. That is to say, if *Contract B* returns an `err`, then any modifications to its own data members are reverted, but *Contract A* can still modify its own data members and have those materialise if it returns an `ok` itself. However, the first contract in the call remains in ultimate control. If it returns an `err` then anything down the line will not materialise.

Committing or reverting changes is therefore determined sequentially.

It means that in a multi-contract call chain, the calling contract knows with absolute certainty that a sub call will not materialise on chain if it returns an `err` response. Nonetheless, a contract may depend on the success of the sub contract call. For example, a wallet contract is calling into a token contract to transfer a token. It happens all too often that developers forget to check the return value. To protect against such mistakes, Clarity forbids intermediary responses to be left unchecked. An intermediary response is a response that, while part of an expression, is not the one that is returned. We can illustrate it with the `begin` function:

```
(begin
    true        ;; this is a boolean, so it is fine.
    (err false) ;; this is an *intermediary response*.
    (ok true)   ;; this is the response returned by the begin.
)
```

Executing the snippet returns the following error:

> *Analysis error: intermediary responses in consecutive statements must be checked*

Since responses are meant to indicate the success or failure of an action, they cannot be left dangling. *Checking* the response simply means dealing with it; that is, unwrapping it or propagating it. We can therefore put a `try!` or another control flow function around it to fix the code.

Any function call that returns a response must either be returned by the calling function or checked. Usually, this takes the form of an inter-contract function call, but some built-in functions also return a response type. The `stx-transfer?` function is one of them.

We will see what this behaviour looks like with the following deposit contract. It contains a function that allows a user to deposit STX and will track individual user deposits. The function is invalid due to an unchecked intermediary response. Your challenge is to *try* to locate and fix it.

```
(define-map deposits principal uint)

(define-read-only (get-total-deposit (who principal))
    (default-to u0 (map-get? deposits who))
)

(define-public (deposit (amount uint))
    (begin
        (stx-transfer? amount tx-sender (as-contract tx-sender))
        (map-set deposits tx-sender (+ (get-total-deposit tx-sender) amount))
```

```
        (ok true)
    )
)

;; Try a test deposit
(print (deposit u500))
```

Did you figure it out? Analysis gives us a hint, indicating that the `begin` expression contains an intermediary response. In this case, it is the return value of `stx-transfer?`. The easiest way to check the response is by simply wrapping the transfer in a `try!`. That way, the result of the transfer is unwrapped if it is successful, and propagated if it is an `err`. We thus rewrite the line as follows:

```
(try! (stx-transfer? amount tx-sender (as-contract tx-sender)))
```

We could have also solved the issue by unwrapping the response and using conditional statements. However, such a structure would have made the function a lot more convoluted and would honestly overcomplicate things. The art of writing smart contracts is coming up with a straightforward application flow, achieving the desired functionality in the least amount of code.

The new function looks nice, but it is actually possible to simplify the function even more:

```
(define-public (deposit (amount uint))
    (begin
        (map-set deposits tx-sender (+ (get-total-deposit tx-sender) amount))
        (stx-transfer? amount tx-sender (as-contract tx-sender))
    )
)
```

This implementation is functionally equivalent. `stx-transfer?` returns a response of type `(response bool uint)`, we therefore do not need to reiterate our own `(ok true)` at the end of the `begin`. By moving the transfer expression to the last line, its response no longer intermediary and will be returned from the `deposit` function—whether it is an `ok` or an `err`.

The chapter on best practices will teach you some techniques on how to spot code that can be simplified in the same manner.

Using Clarinet

The Clarinet workflow will be introduced by working through a series of example projects. Since it is a command line tool, you need to have some familiarity with *CLIs* (*command-line interfaces*). Still, all commands will be carefully explained so you can type along and take notes. Our IDE of choice is Visual Studio Code, an open source code editor with a large following and tons of extensions to choose from. The official Clarity language support extension is called Clarity for Visual Studio Code. We suggest you install it alongside with the Rainbow Brackets extension for the best experience.

The chapters focusing on projects contain a good amount of screenshots and step by step explanations to make it easy to follow for beginning developers and people who are new to using Visual Studio Code itself. Full project source files are also made available on GitHub. The link can be found at the end of each chapter and in the links and resources section.

From this point, it is assumed that you already successfully installed Clarinet and that it is available in your system PATH (that is to say, typing `clarinet` in your Terminal emulator runs Clarinet). If you do not yet have it installed, head over to the introductory chapter on installing tools. You can verify Clarinet is installed properly by running `clarinet --version` in your favourite Terminal emulator.

```
% clarinet --version
clarinet-cli 2.6.0
```

Creating a new project

For our first project, we will be creating a *multiplayer counter contract*. The contract will have the following features:

- A data store that keeps a counter per principal.
- A public function `count-up` that increments the counter for `tx-sender`.
- A public function `get-count` that returns the current counter value for the passed principal.

Once the contract is finished, we will look at how we can interact with the project manually and how to write automated tests.

Initialising the project folder

Let us start by creating a new folder that will contain the various projects. We will simply call it `projects`. Open the folder in Visual Studio Code and open the built-in terminal emulator by going to the Terminal menu and choosing New Terminal.

We will create a new project called `counter` using Clarinet. Type the following in the Terminal screen:

```
clarinet new counter
```

Clarinet will create a number of directories and files. We will go over these files momentarily.

```
Creating directory counter
Creating directory counter/contracts
Creating directory counter/settings
Creating directory counter/tests
Creating file counter/Clarinet.toml
Creating file counter/settings/Devnet.toml
Creating file counter/settings/Mocknet.toml
Creating directory counter/.vscode
Creating file counter/.vscode/settings.json
```

The `counter` directory will be our working directory. You can either reopen the folder in Visual Studio Code or navigate into it in the terminal session with the command `cd counter`.

Adding a new contract

Inside the `counter` folder, we create a new Clarity contract with the command:

```
clarinet contract new counter
```

Clarinet will create a boilerplate Clarity and test file, and add the contract to the configuration file.

```
Creating file contracts/counter.clar
Creating file tests/counter_test.ts
Adding contract counter to Clarinet.toml
```

Let us now open `Clarinet.toml`. This is the Clarinet configuration file for the project. It contains a reference to all contract files and their dependencies. In our case, only one contract is listed, but for multi-contract projects you can specify their relationship to each other.

```
[project]
name = "counter"
requirements = []
[contracts.counter]
path = "contracts/counter.clar"
```

The files found in the `settings` directory are used for deployment:

- `Devnet.toml` contains configuration that is used when running tests and in console mode.
- `Testnet.toml` is a template for deploying on a testnet chain.
- `Mainnet.toml` is a template for deploying on a mainnet chain.

We will take a closer look at them later. Their contents are not important for our project but it is good to know what they are used for.

Here is what the end result looks like. You can see your project files on the left, the terminal window on the bottom right, and your editor on the top right. *Please note that the directories shown in the image may differ with the current version.*


Visual Studio Code interface

Writing your first smart contract

Now that we have the project setup out of the way, let us jump into writing our first contract.

Open up the `counter.clar` file inside the `contracts` directory.

Defining the data store

A map will be used to store the individual counter values. Maps are a great choice because data can be added as more principals call into the contract over time.

```
(define-map counters principal uint)
```

Our map is called `counters`, it is indexed by a key of the principal type and contains an unsigned integer. Since our counter will always count up, using an unsigned integer as opposed to a signed integer makes the most sense. The map thus relates a principal to a number.

We will also add a read-only function that returns the counter value for a specified principal. If the principal does not exist in the map, we return a default value of `u0`. Clarity has a built-in function called `default-to` that takes a default value and an optional type. If the optional type is a `(some ...)`, it will unwrap and return it. If it is a `none`, then it will return the specified default value.

```
(define-read-only (get-count (who principal))
    (default-to u0 (map-get? counters who))
)
```

Since `map-get?` returns either a `(some ...)` if the value is found or `none` otherwise, it is perfect to directly plug into `default-to`.

Creating the public function

The next step is to create the `count-up` function that will increment the counter for the `tx-sender`. We will simply have the function return a `true` status. (Remember, overflows cause an abort automatically, so we do not have to deal with it ourselves.) We already created a really useful read-only function that we can repurpose. All we got to do is to set the map value for the `tx-sender` to be equal to the current counter value incremented by `u1`.

```
(define-public (count-up)
    (begin
        (map-set counters tx-sender (+ (get-count tx-sender) u1))
        (ok true)
    )
)
```

Looks great! But we can simplify it further. The `map-set` actually returns a boolean value so we could wrap it in an `ok` to cut down on the lines of code. The refactored function therefore looks like this:

```
(define-public (count-up)
    (ok (map-set counters tx-sender (+ (get-count tx-sender) u1)))
)
```

Putting it together

Our first contract turned out to be surprisingly simple. That is the power of Clarity. Let us put the entire contract together so that we can start testing it.

```
;; Multiplayer Counter contract

(define-map counters principal uint)

(define-read-only (get-count (who principal))
    (default-to u0 (map-get? counters who))
)

(define-public (count-up)
    (ok (map-set counters tx-sender (+ (get-count tx-sender) u1)))
)
```

If we made any typos along the way, then the Clarity for VSCode Extension should have highlighted them. Still, we can use Clarinet to validate our contract by running the check command:

```
clarinet check
```

The command will output any errors it finds or nothing if the contract is in order. No errors? Great, it is high time to play around with our contract.

Interacting with your contract

Clarinet makes it easy run a local mock chain to manually interact with your contracts. You can start an interactive session just like you can with the Clarity REPL. When you initiate the session, Clarinet will automatically deploy your contracts based on the definitions in `Clarinet.toml`. To start the console:

```
clarinet console
```

The console will print a bunch of information. After the Clarinet version, it shows a table of your currently deployed contracts. It should show the counter contract that we just created. If it does not, then there is an error in the code and you should go back to the previous section to figure it out. **You can exit out of the console at any time by pressing CTRL + D on your keyboard.**

There is a second table that contains a list of addresses and balances. These are local test wallets that are initialised based on the contents of the `settings/Devnet.toml` file.

 Clarinet console output

Here is the contract output in text format:

```
Contracts
+--------------------------------------------------+-----------------------------+
| Contract identifier                              | Public functions            |
+--------------------------------------------------+-----------------------------+
| ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.counter | (count-up)                  |
|                                                  | (get-count (who principal)) |
+--------------------------------------------------+-----------------------------+
```

Contract calls

Since the console starts a REPL session, we can use the full power of Clarity to play around with our contracts. By default, the `tx-sender` is equal to the first address in the table. (In the image above, it is `ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE`.) You can of course verify it by entering `tx-sender` into the console.

Let us try to read the counter value of `tx-sender`. It should be `u0` because that is the default value we programmed into our contract. The `contract-call?` function is used to call a public or read-only function in a contract. Its function signature looks as follow:

```
(contract-call? contract-identifier function-name param-1 param-2 ...)
```

The *contract identifier* specifies which contract to call. There are three ways to pass in a contract:

1. A fully qualified contract principal. This is a Stacks address with a contract name, prefixed by a single quote (`'`). In our example, it is:

   ```
   'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.counter
   ```

2. A contract specified by shorthand notation. If multiple contracts are deployed by the same `tx-sender`, then they can refer to each other without including the Stacks address. It is a bit like a relative path:

   ```
   .counter
   ```

3. A trait reference. Those will be covered in the chapter on traits.

What follows is the function name and parameters, if any.

In order to call the `get-count` function, we therefore type the following:

```
(contract-call? .counter get-count tx-sender)
```

If everything went well, the console should respond with the expected `u0`.

Changing the tx-sender

Now that we know how to call functions in our contract, we can try calling `count-up` and seeing if the counter for the `tx-sender` changes:

```
(contract-call? .counter count-up)
```

Returns `(ok true)`, indicating the call was successful. Reading the counter status using the call in the last section indeed reveals that the count is now `u1`!

The console can switch the sending context to any other address. We will pick the second one, called *wallet 1*, and set it as the `tx-sender`. For this we use a management command. Those always start with a double colon (`::`):

```
::set_tx_sender ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK
```

Now we can call into the contract again to verify that the count for this `tx-sender` is still `u0`. Remember that now that we changed the sending address to a different one from which the contract was deployed, we have to specify the full contract principal.

```
(contract-call? 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.counter get-count tx-
sender)
```

Great, it works! Now try calling `count-up` a few times. You can play around by changing the sender a couple of times and by specifying principals for `get-count`.

Congratulations, you just made and manually tested your first Clarity smart contract.

Testing your contract

Testing software is important and smart contracts are no different. However, properly testing smart contracts is absolutely crucial as it is impossible to update them after they have been deployed to the chain. The mantra of the lazy developer, "we will fix it in post", definitely does not fly with smart contracts.

Clarinet features a testing framework that allows developers to automate tests. These tests are written in code and describe the intended functionality of the smart contract. In this case, that code is written in TypeScript. Covering TypeScript itself is out of scope for the book but most web developers will be familiar with it. If you have done some JavaScript then it should also be easy enough to pick up.

Unit tests

Clarinet will generate a test file for each contract you instantiate using the `clarinet new contract` command. In the folder `tests` you will find one for our counter contract called `counter_test.ts`. Depending on the version of Clarinet you are using, it should have added some imports and template code to the test file. You may remove the template code but make sure you leave the `import` statements intact.

 Clarinet test file template

Tests are defined using the `Clarinet.test` function. They have a name, which is used as a description, and a callback function that is executed on test. Before each test commences, Clarinet will instantiate a fresh local chain and then run the function. The function should run one or more contract calls against the contract to test it. The testing suite makes writing these calls easy but still uses `contract-call?` Behind the scenes.

In order to write successful tests, we should first think about the *intended behaviour* of the contract. That does not only include success states but also failure states. Remember, behaviour is observed from the outside. We care about which inputs produce what outputs. How it is implemented is a detail. Let us describe all of them in bullet point format.

- `get-count` returns `u0` for principals that have never called into `count-up`.
- `get-count` returns the number of times a specific principal called into `count-up`.
- `count-up` increments the counter for the `tx-sender` as observed by `get-count` and returns `(ok true)`.

Testing get-count

Tests should be kept as simple as possible. You want to test exactly one aspect of a function call. The reason being that if a bug is later introduced in the contract, only the tests that specifically target that section of code will fail. Complex tests that leverage a lot of functionality can fail because of many different reasons and may make identifying the actual bug a lot more difficult.

We will therefore start by writing a test for the default behaviour of `get-count`; namely, it returning `u0` for principals that have never called `count-up` before.

```
Clarinet.test({
  name: "get-count returns u0 for principals that never called count-up before",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    // Get the deployer account.
    let deployer = accounts.get("deployer")!;
```

```
    // Call the get-count read-only function.
    // The first parameter is the contract name, the second the
    // function name, and the third the function arguments as
    // an array. The final parameter is the tx-sender.
    let count = chain.callReadOnlyFn("counter", "get-count", [
      types.principal(deployer.address),
    ], deployer.address);

    // Assert that the returned result is a uint with a value of 0 (u0).
    count.result.expectUint(0);
  },
});
```

We are now ready to execute the test using `clarinet test`. If everything went well, our test should pass with an "ok" status.

```
* get-count returns u0 for principals that never called count-up before ... ok
(146ms)
```

Testing count-up

The next step is to test `count-up`. We will write a minimal test that first calls `count-up` once, and then check the count using `get-count`. As mentioned, Clarinet will refresh the chain state for every test so that the starting conditions are always the same. We therefore know that the first `count-up` call will always result in the counter for that `tx-sender` to be `u1`.

```
Clarinet.test({
  name: "count-up counts up for the tx-sender",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    // Get the deployer account.
    let deployer = accounts.get("deployer")!;

    // Mine a block with one transaction.
    let block = chain.mineBlock([
      // Generate a contract call to count-up from the deployer address.
      Tx.contractCall("counter", "count-up", [], deployer.address),
    ]);

    // Get the first (and only) transaction receipt.
    let [receipt] = block.receipts;

    // Assert that the returned result is a boolean true.
    receipt.result.expectOk().expectBool(true);

    // Get the counter value.
    let count = chain.callReadOnlyFn("counter", "get-count", [
      types.principal(deployer.address),
    ], deployer.address);

    // Assert that the returned result is a u1.
```

```
    count.result.expectUint(1);
  },
});
```

Clarinet simulates the mining process by calling `chain.mineBlock()` with an array of transactions. The contract call transactions themselves are constructed using the `Tx.contractCall()`. The Clarinet TypeScript library provides a lot of helper functions to mine blocks, construct transactions, and create function arguments. Inspect the source or see the Clarinet documentation for a full overview.

Testing the multiplayer aspect

Although the two tests we have written cover all code paths in our smart contract, it makes sense to add another test that explicitly tests the multiplayer aspect of our contract.

```
Clarinet.test({
  name: "counters are specific to the tx-sender",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    // Get some accounts
    let deployer = accounts.get("deployer")!;
    let wallet1 = accounts.get("wallet_1")!;
    let wallet2 = accounts.get("wallet_2")!;

    // Mine a few contract calls to count-up
    let block = chain.mineBlock([
      // The deployer account calls count-up zero times.

      // Wallet 1 calls count-up one time.
      Tx.contractCall("counter", "count-up", [], wallet1.address),

      // Wallet 2 calls count-up two times.
      Tx.contractCall("counter", "count-up", [], wallet2.address),
      Tx.contractCall("counter", "count-up", [], wallet2.address),
    ]);

    // Get and assert the counter value for deployer.
    let deployerCount = chain.callReadOnlyFn("counter", "get-count", [
      types.principal(deployer.address),
    ], deployer.address);
    deployerCount.result.expectUint(0);

    // Get and assert the counter value for wallet 1.
    let wallet1Count = chain.callReadOnlyFn("counter", "get-count", [
      types.principal(wallet1.address),
    ], wallet1.address);
    wallet1Count.result.expectUint(1);

    // Get and assert the counter value for wallet 2.
    let wallet2Count = chain.callReadOnlyFn("counter", "get-count", [
      types.principal(wallet2.address),
    ], wallet2.address);
    wallet2Count.result.expectUint(2);
```

```
  },
});
```

Let us again run our tests using `clarinet test`. Clarinet will execute all tests and show us the result. We should get three ok's. If, not then the output will show you where it went wrong. Edit your code and try again.

```
Running counter/tests/counter_test.ts
* get-count returns u0 for principals that never called count-up before ... ok
(142ms)
* count-up counts up for the tx-sender ... ok (144ms)
* counters are specific to the tx-sender ... ok (148ms)
```

That concludes our first project. You now know what the entire Clarity smart contract development flow looks like! The full source code of the counter project can be found here: https://github.com/clarity-lang/book/tree/main/projects/counter.

Practice projects

This chapter contains three practice projects to further hone your skills. They will cover different concepts and Clarity features.

- The time-locked wallet project provides a nice introduction to how blockchains change over time. You will use the block height (the length of the chain) to create a contract that unlocks when an amount of time has passed.
- The smart-claimant is a follow-up project to illustrate how inter-contract calls work and how you can you can build on top of other smart contracts.
- Finally, the multi-signature vault project teaches you how to build a basic voting mechanism and how to iterate over sequences.

Time-locked wallet

The block height can be used to perform actions over time. If you know the average block time, then you can calculate roughly how many blocks will be mined in a specific time frame. We will use this concept to create a wallet contract that unlocks at a specific block height. Such a contract can be useful if you want to bestow tokens to someone after a certain time period. Imagine that in the crypto-future you want to put some money aside for when your child comes of age. Naturally you would do this by means of a smart contract! Let us get started.

From our main `projects` folder, we create a new project.

```
clarinet new timelocked-wallet
```

Inside the `timelocked-wallet` into the folder, we create the contract files using the following command:

```
clarinet contract new timelocked-wallet
```

Features

Instead of starting to code straight away, let us take a moment to consider the features we want to have.

- A user can deploy the time-locked wallet contract.
- Then, the user specifies a block height at which the wallet unlocks and a beneficiary.
- Anyone, not just the contract deployer, can send tokens to the contract.
- The beneficiary can claim the tokens once the specified block height is reached.
- Additionally, the beneficiary can transfer the right to claim the wallet to a different principal. (For whatever reason.)

With the above in mind, the contract will thus feature the following public functions:

- `lock`, takes the principal, unlock height, and an initial deposit amount.
- `claim`, transfers the tokens to the `tx-sender` if and only if the unlock height has been reached and the `tx-sender` is equal to the beneficiary.
- `bestow`, allows the beneficiary to transfer the right to claim the wallet.

Constants & variables

Contracts should be as easy to read and maintainable as possible. We will therefore make generous use of constants to not only define the contract owner but also various error states. Errors can take the following forms:

- Somebody other than the contract owner called `lock`.
- The contract owner tried to call `lock` more than once.
- The passed unlock height is in the past.
- The owner called `lock` with an initial deposit of zero (`u0`).
- Somebody other than the beneficiary called `claim` or `lock`.
- The beneficiary called `claim` but the unlock height has not yet been reached.

Two data variables are needed to store the beneficiary and the unlock height as an unsigned integer. We will make the beneficiary an optional principal type to account for the uninitialised state of the contract. (That is, before the contract owner called `lock`.)

```
;; Owner
(define-constant contract-owner tx-sender)

;; Errors
(define-constant err-owner-only (err u100))
(define-constant err-already-locked (err u101))
(define-constant err-unlock-in-past (err u102))
(define-constant err-no-value (err u103))
(define-constant err-beneficiary-only (err u104))
(define-constant err-unlock-height-not-reached (err u105))

;; Data
(define-data-var beneficiary (optional principal) none)
(define-data-var unlock-height uint u0)
```

The error codes themselves are made up. They are meant to be processed by a frontend application for our contract. As long as we use the `(err ...)` response type, we know for sure that any possible changes will revert.

Implementing lock

The `lock` function does nothing more than transferring some tokens from the `tx-sender` to itself and setting the two variables. However, we must not forget to check if the proper conditions are set. Specifically:

- Only the contract owner may call `lock`.
- The wallet cannot be locked twice.
- The passed unlock height should be at some point in the future; that is, it has to be larger than the current height.
- The initial deposit should be larger than zero. Also, the deposit should succeed.

Most of those translate into assertions. The function is thus implemented as follows:

```
(define-public (lock (new-beneficiary principal) (unlock-at uint) (amount uint))
    (begin
        (asserts! (is-eq tx-sender contract-owner) err-owner-only)
        (asserts! (is-none (var-get beneficiary)) err-already-locked)
        (asserts! (> unlock-at block-height) err-unlock-in-past)
        (asserts! (> amount u0) err-no-value)
        (try! (stx-transfer? amount tx-sender (as-contract tx-sender)))
        (var-set beneficiary (some new-beneficiary))
        (var-set unlock-height unlock-at)
        (ok true)
    )
)
```

Notice how we can use the constants we defined before as the throw values for the assertions? That allows for some pretty legible code. The `(as-contract tx-sender)` part gives us the principal of the contract.

Implementing bestow

The `bestow` function will be straightforward. It checks if the `tx-sender` is the current beneficiary, and if so, will update the beneficiary to the passed principal. One side-note to keep in mind is that the principal is stored as an `(optional principal)`. We thus need to wrap the `tx-sender` in a `(some ...)` before we do the comparison.

```
(define-public (bestow (new-beneficiary principal))
    (begin
        (asserts! (is-eq (some tx-sender) (var-get beneficiary)) err-beneficiary-
only)
        (var-set beneficiary (some new-beneficiary))
        (ok true)
    )
)
```

Implementing claim

Finally, the `claim` function should check if both the `tx-sender` is the beneficiary and that the unlock height has been reached.

```
(define-public (claim)
    (begin
        (asserts! (is-eq (some tx-sender) (var-get beneficiary)) err-beneficiary-
only)
        (asserts! (>= block-height (var-get unlock-height)) err-unlock-height-not-
reached)
        (as-contract (stx-transfer? (stx-get-balance tx-sender) tx-sender (unwrap-
panic (var-get beneficiary))))
    )
)
```

Manual testing

Time to hop into a `clarinet console` session to try out the contract.

```
Contracts
+-----------------------------------------------------------------+---------------------
-----------------+
| Contract identifier                                             | Public functions
|
+-----------------------------------------------------------------+---------------------
-----------------+
| ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.timelocked-wallet | (bestow (new-
beneficiary principal)) |
|                                                                 | (claim)
|
|                                                                 | (lock
|
|                                                                 |     (new-beneficiary
principal)      |
|                                                                 |     (unlock-at uint)
```

```
|
|                                                                 |     (amount uint))
|
+-----------------------------------------------------------------+--------------------
-----------------+
```

If the contract does not show up, then there was a bug or syntax error. Use `clarinet check` to track them down.

For the first test, the wallet will be locked for the first principal that comes after the deployer (`wallet_1`). We can pick a block height that is really low as console sessions always starts at a block height of zero. Here is the console interaction to lock the wallet until height 10, with an initial deposit of 100 mSTX:

```
>> (contract-call? .timelocked-wallet lock
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK u10 u100)
Events emitted
{"type":"stx_transfer_event","stx_transfer_event":
{"sender":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE","recipient":"ST1HTBVD3JG9C05J7HBJ
wallet","amount":"100"}}
```

That worked! Pay close attention to the STX transfer event from the `tx-sender` to the contract. The balance of the contract can be verified using the management command `::get_assets_maps` .

```
>> ::get_assets_maps
+-----------------------------------------------------------------+---------+
| Address                                                         | STX     |
+-----------------------------------------------------------------+---------+
| ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE (deployer)            | 999900  |
+-----------------------------------------------------------------+---------+
| ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.timelocked-wallet | 100     |
+-----------------------------------------------------------------+---------+
```

We then assume the identity of the beneficiary and see if we can claim the wallet. (Remember the full contract principal has to be specified in this case.)

```
>> ::set_tx_sender ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK
tx-sender switched to ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK
>> (contract-call? 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.timelocked-wallet
claim)
(err u105)
```

Trying to claim returns a `(err u105)` . That is the error value we related to the unlock height not having been reached. So far so good.

The block height in the REPL does not increment by itself. Mining can be simulated by using `::advance_chain_tip` . Let us see if we can claim the wallet after incrementing the block height by ten.

```
>> ::advance_chain_tip 10
10 blocks simulated, new height: 10
>> (contract-call? 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.timelocked-wallet
claim)
Events emitted
{"type":"stx_transfer_event","stx_transfer_event":
{"sender":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.timelocked-
wallet","recipient":"ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK","amount":"100"}}
(ok true)
```

The `ok` and STX transfer event prove that it worked. Feel free to check the asset maps for good measure.

Unit tests

We identify the following cases in order to write comprehensive unit tests. The smart contract:

- Allows the contract owner to lock an amount.
- Does not allow anyone else to lock an amount.
- Cannot be locked more than once.
- Cannot set the unlock height to a value less than the current block height.
- Allows the beneficiary to bestow the right to claim to someone else.
- Does not allow anyone else to bestow the right to claim to someone else. (Not even the contract owner.)
- Allows the beneficiary to claim the balance when the block height is reached.
- Does not allow the beneficiary to claim the balance before the block-height is reached.
- Nobody but the beneficiary can claim the balance once the block height is reached.

Clarinet features built-in assertion functions to check if an expected STX transfer event actually happened. Those will be used to keep the unit tests succinct.

The test file for a contract is always found in the `tests` folder. It is named after the contract: `timelocked-wallet_test.ts`. Clear the file but be sure to keep the `import` statement at the top.

Testing lock

We start by writing the four tests that cover the different cases of `lock`.

```
Clarinet.test({
  name: "Allows the contract owner to lock an amount",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const amount = 10;
    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(10),
        types.uint(amount),
      ], deployer.address),
    ]);
```

```
    // The lock should be successful.
    block.receipts[0].result.expectOk().expectBool(true);
    // There should be a STX transfer of the amount specified.
    block.receipts[0].events.expectSTXTransferEvent(
      amount,
      deployer.address,
      `${deployer.address}.timelocked-wallet`,
    );
  },
});

Clarinet.test({
  name: "Does not allow anyone else to lock an amount",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const accountA = accounts.get("wallet_1")!;
    const beneficiary = accounts.get("wallet_2")!;
    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(10),
        types.uint(10),
      ], accountA.address),
    ]);

    // Should return err-owner-only (err u100).
    block.receipts[0].result.expectErr().expectUint(100);
  },
});

Clarinet.test({
  name: "Cannot lock more than once",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const amount = 10;
    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(10),
        types.uint(amount),
      ], deployer.address),
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(10),
        types.uint(amount),
      ], deployer.address),
    ]);

    // The first lock worked and STX were transferred.
    block.receipts[0].result.expectOk().expectBool(true);
    block.receipts[0].events.expectSTXTransferEvent(
      amount,
```

```
      deployer.address,
      `${deployer.address}.timelocked-wallet`,
    );

    // The second lock fails with err-already-locked (err u101).
    block.receipts[1].result.expectErr().expectUint(101);

    // Assert there are no transfer events.
    assertEquals(block.receipts[1].events.length, 0);
  },
});

Clarinet.test({
  name: "Unlock height cannot be in the past",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const targetBlockHeight = 10;
    const amount = 10;

    // Advance the chain until the unlock height plus one.
    chain.mineEmptyBlockUntil(targetBlockHeight + 1);

    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(targetBlockHeight),
        types.uint(amount),
      ], deployer.address),
    ]);

    // The second lock fails with err-unlock-in-past (err u102).
    block.receipts[0].result.expectErr().expectUint(102);

    // Assert there are no transfer events.
    assertEquals(block.receipts[0].events.length, 0);
  },
});
```

Testing bestow

`bestow` is a simple function that allows the beneficiary to transfer the right to claim. We therefore have to make sure that only the beneficiary can successfully call `bestow`.

```
Clarinet.test({
  name: "Allows the beneficiary to bestow the right to claim to someone else",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const newBeneficiary = accounts.get("wallet_2")!;
    const block = chain.mineBlock([
```

```
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(10),
        types.uint(10),
      ], deployer.address),
      Tx.contractCall("timelocked-wallet", "bestow", [
        types.principal(newBeneficiary.address),
      ], beneficiary.address),
    ]);

    // Both results are (ok true).
    block.receipts.map(({ result }) => result.expectOk().expectBool(true));
  },
});

Clarinet.test({
  name:
    "Does not allow anyone else to bestow the right to claim to someone else (not
even the contract owner)",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const accountA = accounts.get("wallet_3")!;
    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(10),
        types.uint(10),
      ], deployer.address),
      Tx.contractCall("timelocked-wallet", "bestow", [
        types.principal(deployer.address),
      ], deployer.address),
      Tx.contractCall("timelocked-wallet", "bestow", [
        types.principal(accountA.address),
      ], accountA.address),
    ]);

    // All but the first call fails with err-beneficiary-only (err u104).
    block.receipts.slice(1).map(({ result }) =>
      result.expectErr().expectUint(104)
    );
  },
});
```

Testing claim

For `claim`, we test the cases of the unlock height being reached or not, and that only the beneficiary can claim.

```
Clarinet.test({
  name:
```

```
    "Allows the beneficiary to claim the balance when the block-height is reached",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const targetBlockHeight = 10;
    const amount = 10;
    chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(targetBlockHeight),
        types.uint(amount),
      ], deployer.address),
    ]);

    // Advance the chain until the unlock height.
    chain.mineEmptyBlockUntil(targetBlockHeight);

    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "claim", [], beneficiary.address),
    ]);

    // The claim was successful and the STX were transferred.
    block.receipts[0].result.expectOk().expectBool(true);
    block.receipts[0].events.expectSTXTransferEvent(
      amount,
      `${deployer.address}.timelocked-wallet`,
      beneficiary.address,
    );
  },
});

Clarinet.test({
  name:
    "Does not allow the beneficiary to claim the balance before the block-height is
reached",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const targetBlockHeight = 10;
    const amount = 10;
    chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(targetBlockHeight),
        types.uint(amount),
      ], deployer.address),
    ]);

    // Advance the chain until the unlock height minus one.
    chain.mineEmptyBlockUntil(targetBlockHeight - 1);

    const block = chain.mineBlock([
```

```
      Tx.contractCall("timelocked-wallet", "claim", [], beneficiary.address),
    ]);

    // Should return err-unlock-height-not-reached (err u105).
    block.receipts[0].result.expectErr().expectUint(105);
    assertEquals(block.receipts[0].events.length, 0);
  },
});

Clarinet.test({
  name:
    "Does not allow anyone else to claim the balance when the block-height is
reached",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = accounts.get("wallet_1")!;
    const other = accounts.get("wallet_2")!;
    const targetBlockHeight = 10;
    const amount = 10;
    chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary.address),
        types.uint(targetBlockHeight),
        types.uint(amount),
      ], deployer.address),
    ]);

    // Advance the chain until the unlock height.
    chain.mineEmptyBlockUntil(targetBlockHeight);

    const block = chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "claim", [], other.address),
    ]);

    // Should return err-beneficiary-only (err u104).
    block.receipts[0].result.expectErr().expectUint(104);
    assertEquals(block.receipts[0].events.length, 0);
  },
});
```

The full source code of the project can be found here: https://github.com/clarity-lang/book/tree/main/projects/timelocked-wallet.

Smart claimant

The time-locked wallet we created in the previous section is delightfully simple. But imagine that after it is deployed, the beneficiary desires to split the balance over multiple distinct beneficiaries. Maybe it was deployed by an old relative years ago and the beneficiary now wants to share with the rest of the family. Whatever the reason, we obviously cannot simply go back and change or redeploy the time-locked wallet. At some point it is going to unlock, after which the sole beneficiary can claim the entire balance. The solution? We can create a minimal *ad hoc* smart contract to act as the beneficiary! It will call `claim`, and if successful, disburse the tokens to a list of principals equally.

The point of this exercise is to show how smart contracts can interact with each other and how one can augment functionality or mitigate issues of older contracts.

Example ad hoc contract

We will add the `smart-claimant` contract to the existing time-locked wallet project for ease of development and testing. Navigate into it and add it using `clarinet contract new smart-claimant`.

For this example, we will assume there to be four beneficiaries. We will take wallets 1 through 4 as defined in the Clarinet configuration. (Adjust the addresses if your Clarinet configuration differs.)

```
+-----------------------------------------------------+---------+
| ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK (wallet_1) | 1000000 |
+-----------------------------------------------------+---------+
| ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2 (wallet_2) | 1000000 |
+-----------------------------------------------------+---------+
| ST21HMSJATHZ888PD0S0SSTWP4J61TCRJYEVQ0STB (wallet_3) | 1000000 |
+-----------------------------------------------------+---------+
| ST2QXSK64YQX3CQPC530K79XWQ98XFAM9W3XKEH3N (wallet_4) | 1000000 |
+-----------------------------------------------------+---------+
```

Custom claim function

Clarity is well-suited for creating small ad hoc smart contracts. Instead of coming up with a complicated mechanism for adding and removing beneficiaries, we will keep it simple and imagine that the people that are supposed to receive a portion of the balance are in the same room and that the wallet is unlocking imminently. They witness the creation of the contract by the current beneficiary and provide their wallet addresses directly.

The custom claim function will:

- Call `claim` on the time-locked wallet, exiting if it fails.
- Read the balance of the current contract. We do not read the balance of the time-locked wallet because someone might have sent some tokens to the smart-claimant by mistake. We want to include those tokens as well.
- Calculate an equal share for each recipient by dividing the total balance by the number of recipients.
- Send the calculated share to each recipient.
- Transfer the remainder in case of a rounding error. (Remember that integers have no decimal point.)

Whipping this up as a single hard-coded function is effortless.

```
(define-public (claim)
    (begin
        (try! (as-contract (contract-call? .timelocked-wallet claim)))
        (let
            (
                (total-balance (as-contract (stx-get-balance tx-sender)))
                (share (/ total-balance u4))
            )
            (try! (as-contract (stx-transfer? share tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)))
            (try! (as-contract (stx-transfer? share tx-sender
'ST20ATRN26N9P05V2F1RHFRV24X8C8M3W54E427B2)))
            (try! (as-contract (stx-transfer? share tx-sender
'ST21HMSJATHZ888PD0S0SSTWP4J61TCRJYEVQ0STB)))
            (try! (as-contract (stx-transfer? (stx-get-balance tx-sender) tx-sender
'ST2QXSK64YQX3CQPC530K79XWQ98XFAM9W3XKEH3N)))
            (ok true)
        )
    )
)
```

That is it. Guarding `claim` is unnecessary because the recipients are hardcoded. If any of the token transfers fail then the whole call is reverted. (Curious how to reduce the code repetition seen above? You can find some tips and tricks in the chapter on best practices.)

Unit tests

The smart-claimant does not care for what reason the time-locked wallet would error out. We therefore only need to consider the state of a successful transfer.

```
Clarinet.test({
  name: "Disburses tokens once it can claim the time-locked wallet balance",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const beneficiary = `${deployer.address}.smart-claimant`;
    const wallet1 = accounts.get("wallet_1")!;
    const wallet2 = accounts.get("wallet_2")!;
    const wallet3 = accounts.get("wallet_3")!;
    const wallet4 = accounts.get("wallet_4")!;
    const unlock_height = 10;
    const amount = 1000; // be sure to pick a test amount that is divisible by 4 for
this test.
    const share = Math.floor(amount / 4);
    chain.mineBlock([
      Tx.contractCall("timelocked-wallet", "lock", [
        types.principal(beneficiary),
        types.uint(unlock_height),
        types.uint(amount),
      ], deployer.address),
    ]);
    chain.mineEmptyBlockUntil(unlock_height);
```

```
    const block = chain.mineBlock([
      Tx.contractCall("smart-claimant", "claim", [], deployer.address),
    ]);

    // Take the first receipt.
    const [receipt] = block.receipts;
    // The claim should be successful.
    receipt.result.expectOk().expectBool(true);

    // All wallets should have received their share.
    receipt.events.expectSTXTransferEvent(share, beneficiary, wallet1.address);
    receipt.events.expectSTXTransferEvent(share, beneficiary, wallet2.address);
    receipt.events.expectSTXTransferEvent(share, beneficiary, wallet3.address);
    receipt.events.expectSTXTransferEvent(share, beneficiary, wallet4.address);
  },
});
```

The full source code of the project can be found here: https://github.com/clarity-lang/book/tree/main/projects/timelocked-wallet.

Multi-signature vault

Blockchain technology has enabled us to decentralise more than just digital asset management. Another popular field of study is that of decentralised governance. Participating in a vote of any kind can be a very opaque process. Whether it is a vote for the most popular song on the radio or for an elected government official, participants cannot verify whether the process is fair or that the results are genuine. DAOs (*Decentralised Autonomous Organisations*) can change all that. A DAO is a smart contract that organises some type of decision-making power, usually on behalf of its members.

DAOs can be very complex, featuring multiple levels of management, asset delegation, and member management. Some even have their own tokens that function as an ownership stake or right to access! Most, if not all, aspects of a conventional corporate structure can be translated into a set of smart contracts that mandate corporate bylaws with the integrity that a blockchain provides. The potential of DAOs cannot be underestimated.

For this project, we will create a simplified DAO that allows its members to vote on which principal is allowed to withdraw the DAO's token balance. The DAO will be initialised once when deployed, after which members can vote in favour or against specific principals.

Features

The contract deployer will only have the ability to initialise the contract and will then run its course. The initialising call will define the members (a list of principals) and the number of votes required to be allowed to withdraw the balance.

The voting mechanism will work as follows:

- Members can issue a yes/no vote for any principal.
- Voting for the same principal again replaces the old vote.
- Anyone can check the status of a vote.
- Anyone can tally all the votes for a specific principal.

Once a principal reaches the number of votes required, it may withdraw the tokens.

Constants & variables

We begin with the usual constants to define the contract owner and error codes. When it comes to errors, we can foresee three failures on the initialising step.

1. Someone other than the owner is trying to initialise.
2. The vault is already locked.
3. The initialising call specifies an amount of votes required that is larger the number of members.

The voting process itself will only fail if a non-member tries to vote. Finally, the withdrawal function will only succeed if the voting threshold has been reached.

```
;; Owner
(define-constant contract-owner tx-sender)

;; Errors
(define-constant err-owner-only (err u100))
(define-constant err-already-locked (err u101))
(define-constant err-more-votes-than-members-required (err u102))
```

```
(define-constant err-not-a-member (err u103))
(define-constant err-votes-required-not-met (err u104))
```

The members will be stored in a list with a given maximum length. The votes themselves will be stored in a map that uses a tuple key with two values: the principal of the member issuing the vote and the principal being voted for.

```
;; Variables
(define-data-var members (list 100 principal) (list))
(define-data-var votes-required uint u1)
(define-map votes {member: principal, recipient: principal} {decision: bool})
```

For a simple voting contract, storing the members in a list is acceptable. It also allows us to practice iterating over a list in a few interesting ways. However, it is important to note that such member lists are not sufficient for larger projects as they can quickly become expensive to use. The chapter on best practices covers some uses and possible misuses of lists.

Implementing start

The `start` function will be called by the contract owner to initialise the vault. It is a simple function that updates the two variables with the proper guards in place.

```
(define-public (start (new-members (list 100 principal)) (new-votes-required uint))
    (begin
        (asserts! (is-eq tx-sender contract-owner) err-owner-only)
        (asserts! (is-eq (len (var-get members)) u0) err-already-locked)
        (asserts! (>= (len new-members) new-votes-required) err-more-votes-than-
members-required)
        (var-set members new-members)
        (var-set votes-required new-votes-required)
        (ok true)
    )
)
```

Implementing vote

The vote function is even more straightforward. All we have to do is make sure the `tx-sender` is one of the members. We can do that by checking if the `tx-sender` is present in the members list by using the built-in `index-of` function. It returns an optional type, so we can simply check if it returns a `(some ...)`, rather than a `none`.

```
(define-public (vote (recipient principal) (decision bool))
    (begin
        (asserts! (is-some (index-of? (var-get members) tx-sender)) err-not-a-
member)
        (ok (map-set votes {member: tx-sender, recipient: recipient} {decision:
decision}))
    )
)
```

While we are at it, let us also add a read-only function to retrieve a vote. If a member never voted for a specific principal before, we will default to a negative vote of `false`.

```
(define-read-only (get-vote (member principal) (recipient principal))
    (default-to false (get decision (map-get? votes {member: member, recipient: recipient})))
)
```

There is a lot going on in this function. Here is what happens step by step:

- Use `map-get?` to retrieve the vote tuple. The function will return a `some` or a `none`.
- `get` returns the value of the specified key in a tuple. If `get` is supplied with a `(some tuple)`, it will return a `(some value)`. If `get` is supplied `none`, it returns `none`.
- `default-to` attempts to unwrap the result of `get`. If it is a `some`, it returns the wrapped value. If it is `none`, it returns the default value, in this case `false`.

Tallying the votes

The challenge now is to create a function that can calculate the number of positive votes for a principal. We will have to iterate over the members, retrieve their votes, and increment a counter if the vote equals `true`. Since Clarity is non-Turing complete, unbounded *for-loops* are impossible. In the chapter on sequences we learned that the only two ways to iterate over a list are by using the `map` or `fold` function.

The choice of whether to use `map` or `fold` comes down to a simple question: should the result be another list or a singular value?

We want to *reduce* the list of members to a *number* that represents the total amount of positive votes; meaning, we need `fold`. First we look at the function signature again.

```
(fold accumulator-function input-list initial-value)
```

`fold` will iterate over `input-list`, calling `accumulator-function` for every element in the list. The accumulator function receives two parameters: the next member in the list and the previous accumulator value. The value returned by the accumulator function is used as the input for the next accumulator call.

Since we want to count the number of positive votes, we should increment the accumulator value only when the vote for the principal is `true`. There is no built-in function that can do that so we have to create a custom accumulator as a private function.

```
(define-private (tally (member principal) (accumulator uint))
    (if (get-vote member tx-sender) (+ accumulator u1) accumulator)
)

(define-read-only (tally-votes)
    (fold tally (var-get members) u0)
)
```

The `tally-votes` function returns the result of folding over the members list. Our custom accumulator function `tally` calls the `get-vote` read-only function we created earlier with the

current current member from the list and the `tx-sender`. The result of this call will be either `true` or `false`. If the result is `true`, then `tally` returns the accumulator incremented by one. Otherwise, it returns just the current accumulator value.

Unpacking the `if` expression:

```
(if
    (get-vote member tx-sender) ;; The condition (boolean expression).
    (+ accumulator u1)          ;; Value to return if the condition is true.
    accumulator                 ;; Value to return if the condition is false.
)
```

Since `tally-votes` is a read-only function, it can be called with any `tx-sender` principal without having to send a transaction. Very convenient.

Implementing withdraw

We have everything we need to create the withdraw function. It will tally the votes for `tx-sender` and check if it is larger than or equal to the number of votes required. If the transaction sender passes the bar, the contract shall transfer all its holdings to the `tx-sender`.

```
(define-public (withdraw)
    (let
        (
            (recipient tx-sender)
            (total-votes (tally-votes))
        )
        (asserts! (>= total-votes (var-get votes-required)) err-votes-required-not-
met)
        (try! (as-contract (stx-transfer? (stx-get-balance tx-sender) tx-sender
recipient)))
        (ok total-votes)
    )
)
```

The total votes are returned for convenience so that it can be recorded on the blockchain and perhaps used by the calling application.

Deposit convenience function

Finally, we will add a convenience function to deposit tokens into the contract. It is definitely not required as users can transfer tokens to a contract principal directly. The function will be useful when writing unit tests later.

```
(define-public (deposit (amount uint))
    (stx-transfer? amount tx-sender (as-contract tx-sender))
)
```

Unit tests

It is about time we start making our unit tests a bit more manageable by adding reusable parts. We will define a bunch of standard values and create a setup function to initialise the contract. The function can then be called at the beginning of various tests to take care of calling `start` and making an initial STX token deposit by calling `deposit`.

In our example, we named the contract `multisig-vault`. Make sure to change the `contractName` variable below to your contract name if you gave your contract a different name.

```
const contractName = "multisig-vault";

const defaultStxVaultAmount = 5000;
const defaultMembers = [
  "deployer",
  "wallet_1",
  "wallet_2",
  "wallet_3",
  "wallet_4",
];
const defaultVotesRequired = defaultMembers.length - 1;

type InitContractOptions = {
  chain: Chain;
  accounts: Map<string, Account>;
  members?: Array<string>;
  votesRequired?: number;
  stxVaultAmount?: number;
};

function initContract({
  chain,
  accounts,
  members = defaultMembers,
  votesRequired = defaultVotesRequired,
  stxVaultAmount = defaultStxVaultAmount,
}: InitContractOptions) {
  const deployer = accounts.get("deployer")!;
  const contractPrincipal = `${deployer.address}.${contractName}`;
  const memberAccounts = members.map((name) => accounts.get(name)!);
  const nonMemberAccounts = Array.from(accounts.keys())
    .filter((key) => !members.includes(key))
    .map((name) => accounts.get(name)!);
  const startBlock = chain.mineBlock([
    Tx.contractCall(
      contractName,
      "start",
      [
        types.list(
          memberAccounts.map((account) => types.principal(account.address))
        ),
        types.uint(votesRequired),
      ],
      deployer.address
```

```
    ),
    Tx.contractCall(
      contractName,
      "deposit",
      [types.uint(stxVaultAmount)],
      deployer.address
    ),
  ]);
  return {
    deployer,
    contractPrincipal,
    memberAccounts,
    nonMemberAccounts,
    startBlock,
  };
}
```

Testing start

Let us get the tests for `start` out of the way first:

- The contract owner can initialise the vault.
- Nobody else can initialise the vault.
- The vault can only be initialised once.

```
Clarinet.test({
  name: "Allows the contract owner to initialise the vault",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const memberB = accounts.get("wallet_1")!;
    const votesRequired = 1;
    const memberList = types.list([
      types.principal(deployer.address),
      types.principal(memberB.address),
    ]);
    const block = chain.mineBlock([
      Tx.contractCall(
        contractName,
        "start",
        [memberList, types.uint(votesRequired)],
        deployer.address
      ),
    ]);
    block.receipts[0].result.expectOk().expectBool(true);
  },
});

Clarinet.test({
  name: "Does not allow anyone else to initialise the vault",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
```

```
    const memberB = accounts.get("wallet_1")!;
    const votesRequired = 1;
    const memberList = types.list([
      types.principal(deployer.address),
      types.principal(memberB.address),
    ]);
    const block = chain.mineBlock([
      Tx.contractCall(
        contractName,
        "start",
        [memberList, types.uint(votesRequired)],
        memberB.address
      ),
    ]);
    block.receipts[0].result.expectErr().expectUint(100);
  },
});

Clarinet.test({
  name: "Cannot start the vault more than once",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const memberB = accounts.get("wallet_1")!;
    const votesRequired = 1;
    const memberList = types.list([
      types.principal(deployer.address),
      types.principal(memberB.address),
    ]);
    const block = chain.mineBlock([
      Tx.contractCall(
        contractName,
        "start",
        [memberList, types.uint(votesRequired)],
        deployer.address
      ),
      Tx.contractCall(
        contractName,
        "start",
        [memberList, types.uint(votesRequired)],
        deployer.address
      ),
    ]);
    block.receipts[0].result.expectOk().expectBool(true);
    block.receipts[1].result.expectErr().expectUint(101);
  },
});

Clarinet.test({
  name: "Cannot require more votes than members",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { startBlock } = initContract({
      chain,
```

```
      accounts,
      votesRequired: defaultMembers.length + 1,
    });
    startBlock.receipts[0].result.expectErr().expectUint(102);
  },
});
```

Testing vote

Only members should be allowed to successfully call `vote`. It should also return the right error response if a non-member calls the function.

```
Clarinet.test({
  name: "Allows members to vote",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { memberAccounts, deployer } = initContract({ chain, accounts });
    const votes = memberAccounts.map((account) =>
      Tx.contractCall(
        contractName,
        "vote",
        [types.principal(deployer.address), types.bool(true)],
        account.address
      )
    );
    const block = chain.mineBlock(votes);
    block.receipts.map((receipt) => receipt.result.expectOk().expectBool(true));
  },
});

Clarinet.test({
  name: "Does not allow non-members to vote",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { nonMemberAccounts, deployer } = initContract({ chain, accounts });
    const votes = nonMemberAccounts.map((account) =>
      Tx.contractCall(
        contractName,
        "vote",
        [types.principal(deployer.address), types.bool(true)],
        account.address
      )
    );
    const block = chain.mineBlock(votes);
    block.receipts.map((receipt) => receipt.result.expectErr().expectUint(103));
  },
});
```

Testing get-vote

`get-vote` is a simple read-only function that returns the boolean vote status for a member-recipient combination.

```
Clarinet.test({
  name: "Can retrieve a member's vote for a principal",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { memberAccounts, deployer } = initContract({ chain, accounts });
    const [memberA] = memberAccounts;
    const vote = types.bool(true);
    chain.mineBlock([
      Tx.contractCall(
        contractName,
        "vote",
        [types.principal(deployer.address), vote],
        memberA.address
      ),
    ]);
    const receipt = chain.callReadOnlyFn(
      contractName,
      "get-vote",
      [types.principal(memberA.address), types.principal(deployer.address)],
      memberA.address
    );
    receipt.result.expectBool(true);
  },
});
```

Testing withdraw

The `withdraw` function returns an `ok` response containing the total number of votes for the `tx-sender` if the threshold is met. Otherwise it returns an `(err u104)` (`err-votes-required-not-met`).

```
Clarinet.test({
  name: "Principal that meets the vote threshold can withdraw the vault balance",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { contractPrincipal, memberAccounts } = initContract({
      chain,
      accounts,
    });
    const recipient = memberAccounts.shift()!;
    const votes = memberAccounts.map((account) =>
      Tx.contractCall(
        contractName,
        "vote",
        [types.principal(recipient.address), types.bool(true)],
        account.address
      )
    );
    chain.mineBlock(votes);
    const block = chain.mineBlock([
      Tx.contractCall(contractName, "withdraw", [], recipient.address),
    ]);
    block.receipts[0].result.expectOk().expectUint(votes.length);
```

```
    block.receipts[0].events.expectSTXTransferEvent(
      defaultStxVaultAmount,
      contractPrincipal,
      recipient.address
    );
  },
});

Clarinet.test({
  name: "Principals that do not meet the vote threshold cannot withdraw the vault
balance",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { memberAccounts, nonMemberAccounts } = initContract({
      chain,
      accounts,
    });
    const recipient = memberAccounts.shift()!;
    const [nonMemberA] = nonMemberAccounts;
    const votes = memberAccounts
      .slice(0, defaultVotesRequired - 1)
      .map((account) =>
        Tx.contractCall(
          contractName,
          "vote",
          [types.principal(recipient.address), types.bool(true)],
          account.address
        )
      );
    chain.mineBlock(votes);
    const block = chain.mineBlock([
      Tx.contractCall(contractName, "withdraw", [], recipient.address),
      Tx.contractCall(contractName, "withdraw", [], nonMemberA.address),
    ]);
    block.receipts.map((receipt) => receipt.result.expectErr().expectUint(104));
  },
});
```

Testing changing votes

Members have the ability to change their votes at any time. We will therefore add a final test where a vote change causes a recipient to no longer be eligible to claim the balance.

```
Clarinet.test({
  name: "Members can change votes at-will, thus making an eligible recipient
uneligible again",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const { memberAccounts } = initContract({ chain, accounts });
    const recipient = memberAccounts.shift()!;
    const votes = memberAccounts.map((account) =>
      Tx.contractCall(
        contractName,
```

```
        "vote",
        [types.principal(recipient.address), types.bool(true)],
        account.address
      )
    );
    chain.mineBlock(votes);
    const receipt = chain.callReadOnlyFn(
      contractName,
      "tally-votes",
      [],
      recipient.address
    );
    receipt.result.expectUint(votes.length);
    const block = chain.mineBlock([
      Tx.contractCall(
        contractName,
        "vote",
        [types.principal(recipient.address), types.bool(false)],
        memberAccounts[0].address
      ),
      Tx.contractCall(contractName, "withdraw", [], recipient.address),
    ]);
    block.receipts[0].result.expectOk().expectBool(true);
    block.receipts[1].result.expectErr().expectUint(104);
  },
});
```

The full source code of the project can be found here: https://github.com/clarity-lang/book/tree/main/projects/multisig-vault.

Traits

Traits are a bit like templates for smart contracts. They are a collection of public function definitions that describe names, input types, and output types. The purpose of a trait is to define a *public interface* to which a contract can conform either *implicitly* or *explicitly*. Implicit conformity is reached by implementing the functions defined in the trait. Explicit conformity requires not only implementing the trait, but also asserting the implementation.

Traits are used to ensure compatibility of your smart contracts and are deployed as separate contracts. Other contracts can then refer to these trait contracts and assert conformity. Basically, if there is a trait `my-trait` defined in a contract `my-contract` that is deployed on mainnet, another contract `my-implementation` can point to `my-contract.my-trait`. Traits are integral for dynamic inter-contract calls.

Defining traits

Traits are defined using the `define-trait` function. It takes a trait name as a parameter followed by a series of function signatures in short form. The function signatures looks a bit different from those defining a custom function in that they only contain the name, input types, and output types. Function argument names are not included because they are not important in terms of functionality. Think about it, when performing a contract call, you pass a list of parameters and never specify the argument names themselves. Only the order and types are important.

The general form for defining a trait looks like this:

```
(define-trait my-trait
    (
        (function-name-1 (param-types-1) response-type-1)
        (function-name-2 (param-types-2) response-type-2)
        ;; And so on...
    )
)
```

Time-locked wallet trait

Earlier, we created a time-locked wallet contract that allows someone to lock a number of tokens until a certain block height is reached. We can define a trait for it, allowing us to standardise the interface.

```
(define-trait locked-wallet-trait
    (
        (lock (principal uint uint) (response bool uint))
        (bestow (principal) (response bool uint))
        (claim () (response bool uint))
    )
)
```

Once this trait is deployed on-chain, future contracts can point to it to *implement* the trait.

Implementing traits

Trait conformance is just a fancy way of saying that a specific smart contract implements the functions defined in the trait. Take the following example trait:

```
(define-trait multiplier
    (
        (multiply (uint uint) (response uint uint))
    )
)
```

If we want to implement the trait, all we have to do is make sure that our contract contains a function `multiply` that takes two `uint` parameters and returns a response that is either a `(ok uint)` or `(err uint)`. The following contract will do just that:

```
(define-read-only (multiply (a uint) (b uint))
    (ok (* a b))
)

(define-read-only (divide (a uint) (b uint))
    (ok (/ a b))
)
```

Notice how the contract has another function `divide` that is not present in the `multiplier` trait. That is completely fine because any system that is looking for contracts that implement `multiplier` do not care what other functions those contracts might implement. Reliance on the trait ensures that conforming contracts have a compatible `multiply` function implementation, nothing more.

Asserting trait implementations

In the opening paragraph of this chapter we talked about implicit and explicit conformity. Under normal circumstances you always want to explicitly assert that your contract implements a trait.

Imagine that the `multiplier` trait is deployed in a contract called `multiplier-trait` by the same principal. To assert that the example contract implements the trait, the `impl-trait` function is used.

```
(impl-trait .multiplier-trait.multiplier)
```

By adding this expression, the analyser will check if the contract implements the trait specified when the contract is deployed. It will reject the transaction if the contract is not a full implementation. It is therefore recommended to always use `impl-trait` because it prevent accidental non-conformity.

The `impl-trait` function takes a single *trait reference* parameter. A trait reference is a contract principal plus the name of the trait. Trait references can be written in short form as seen above, or as a fully qualified reference.

```
(impl-trait 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.multiplier-trait.multiplier)
```

The introduction mentioned Clarity favours composition over inheritance. Smart contracts can implement multiple traits, leading to more complex composite behaviour when required. For example,

say that there also exists a `divider` trait that describes the `divide` function, deployed in another contract called `divider-trait`.

```
(define-trait divider
    (
        (divide (uint uint) (response uint uint))
    )
)
```

A contract that implements both traits simply contains multiple `impl-trait` expressions.

```
(impl-trait .multiplier-trait.multiplier)
(impl-trait .divider-trait.divider)
```

Passing traits as arguments

The `contract-call?` function allows contracts to call each other. Since the REPL and Clarinet start an interactive Clarity session, we have also been using it to manually interact with our contracts. The first two arguments are the *contract identifier* and the *function name*, followed by zero or more parameters. We learned that the contract identifier can either be a short form or a fully qualified contract principal. However, there is a third kind of contract identifier; namely, a *trait reference*.

Static dispatch

Up until now we have always been *hardcoding* the contract identifier; that is to say, we directly enter a contract principal as the first argument to `contract-call?`. Remember our smart-claimant? It called into the time-locked wallet contract to claim the balance. Here is the relevant snippet:

```
(begin
    (try! (as-contract (contract-call? .timelocked-wallet claim)))
    (let ;; ...
```

The contract identifier `.timelocked-wallet` is invariant—it is hardcoded into the contract. When the contract is deployed, the analyser will check if the specified contract exists on-chain and whether it contains a public or read-only function called `claim`. Contract calls with an invariant contract identifier are said to dispatch *statically*.

Dynamic dispatch

Traits enable us to pass contract identifiers as function arguments. It enables *dynamic* dispatch of contract calls, meaning that the actual contract called by `contract-call?` depends on the initial call.

You can either define a trait in the same contract or import it using `use-trait`. The latter allows you to bring a trait defined in another contract to the current contract.

```
(use-trait trait-alias trait-identifier)
```

The *trait identifier* is the same as the one used in `impl-trait`. The *trait alias* defines the local name to use in the context of the current contract.

If the locked-wallet trait we created in the previous section were to be deployed in a contract called `locked-wallet-trait`, then importing it with an alias of the same name would look like this:

```
(use-trait locked-wallet-trait 'ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.locked-
wallet-trait.locked-wallet-trait)
```

(`locked-wallet-trait` is repeated because both the contract and the trait bear the same name. Remember that a trait reference follows the pattern `address.contract-name.trait-name`.)

Trait references in function arguments use their own notion. The trait alias as a type is enclosed in angle brackets (`<` and `>`).

```
(define-public (claim-wallet (wallet-contract <locked-wallet-trait>))
    (ok (try! (as-contract (contract-call? wallet-contract claim))))
)
```

The example `claim-wallet` function can then be called with a contract principal that implements the `locked-wallet-trait` like so:

```
(contract-call? .example-contract claim-wallet .timelocked-wallet)
```

The upcoming marketplace practice project contains practical examples of dynamically dispatching contract calls.

Stacks Improvement Proposals

Stacks Improvement Proposals—or SIPs for short—are formal documents that describe the design, implementation, and governance of the Stacks 2.0 blockchain. Although this book is about Clarity, not mentioning SIPs would be an oversight as the Clarity language itself was first formalised by means of a SIP. The SIP process allows for a transparent way to propose and ratify new ideas. Even the SIP process itself is described in one; namely, SIP000. They can be found in a specially designated GitHub repository: https://github.com/stacksgov/sips.

Covering all SIPs is out of the scope of this book, but here are a few of interest for those that want to do a true deep dive:

- SIP000: The Stacks Improvement Proposal Process
- SIP002: The Clarity Smart Contract Language
- SIP006: Clarity Cost Execution Assessment
- SIP009: Standard Trait Definition for Non-Fungible Tokens
- SIP010: Standard Trait Definition for Fungible Tokens

The ones we will cover in this book are SIP009 and SIP010, the NFT and FT standards on Stacks.

SIP009: the NFT standard

(The full SIP can be found here: https://github.com/stacksgov/sips/blob/main/sips/sip-009/sip-009-nft-standard.md.)

NFTs—short for _non-fungible tokens_—are digital assets with unique identifiers. Non-fungible means that the tokens are not equivalent to one another. Think of a stamp or trading card collection. Even though they are all stamps, one particular stamp can be rarer and thus more valuable compared to a more common one. A concert ticket is another example. They all give you access to the venue but some are for seats in the front whilst others put you in the back. Trading your front-row ticket for one in the back is not an equal exchange. NFTs are the crypto equivalent, they are digital collectibles.

The concept of NFTs as we know them today were first pioneered on the Ethereum blockchain. You have probably already come across the acronym *ERC-721*, which is the technical designation for an EIP721-compliant NFT on Ethereum[^1]. Reading up on its history is a fun exercise in itself.

NFTs are first-class members in Clarity. Built-in functions exist to define, mint, transfer, and burn NFTs. However, these built-ins do not define the public interface of NFT contracts. That is where SIP009 comes in. The standard enables interoperability so that different contracts and apps can interact with NFTs contracts seamlessly. Trustless marketplaces are made possible thanks to token standards. (In fact, we will build one in a later chapter.)

The SIP009 NFT trait

Creating a SIP009-compliant NFT comes down implementing a trait. We start by taking a look at the trait and then dissecting it piece by piece.

```
(define-trait sip009-nft-trait
    (
        ;; Last token ID, limited to uint range
        (get-last-token-id () (response uint uint))

        ;; URI for metadata associated with the token
        (get-token-uri (uint) (response (optional (string-ascii 256)) uint))

        ;; Owner of a given token identifier
        (get-owner (uint) (response (optional principal) uint))

        ;; Transfer from the sender to a new principal
        (transfer (uint principal principal) (response bool uint))
    )
)
```

get-last-token-id

A read-only function that returns the token ID of the last NFT that was created by the contract. The ID can be used to iterate through the list and get an idea of how many NFTs are in existence—as long as the contract does not burn tokens. It should never return an `err` response.

get-token-uri

A read-only function that takes a token ID and returns a valid URI (a link) which resolves to a metadata file for that particular NFT. The idea is that you might want to provide some information about an NFT that you cannot or do not want to store on-chain. For example, the URI for an NFT that represents a

piece of digital artwork could contain a link to the image file, the author, and so on. The SIP009 standard does not provide a specification for what should exist at the URI returned by `get-token-uri`. It is supposed to be covered by a future SIP.

If the supplied token ID does not exist, it returns `none`.

get-owner

A read-only function that takes a token ID and returns the principal that owns the specified NFT. If the supplied token ID does not exist, it returns `none`.

transfer

A public function that transfers ownership from one principal to another. If the token ID does not exist, then it must return an `err` response.

[^1]: EIP, as you might have guessed, stands for Ethereum Improvement Proposal.

Creating a SIP009 NFT

About time we create our own SIP009-compliant NFT! We will quickly go over the aforementioned built-in NFT functions to understand how they work and what their limitations are, and then use them to implement the SIP009 trait.

Built-in NFT functions

A new non-fungible token class is defined using the `define-non-fungible-token` function. NFTs have a unique asset name (per contract) and are individually identified by an asset identifier. The developer is free to choose the identifier type although an incrementing unsigned integer is most common.

```
(define-non-fungible-token asset-name asset-identifier-type)
```

The naming schedule is the same as those for variables or function names. The asset identifier type is a normal type signature. We can create an NFT class `my-awesome` token, identified by an unsigned integer as follows:

```
(define-non-fungible-token my-awesome-token uint)
```

One can then use the functions `nft-mint?`, `nft-transfer?`, `nft-get-owner?`, and `nft-burn?` to manage the NFT.

```
;; Define my-awesome-token
(define-non-fungible-token my-awesome-token uint)

;; Mint NFT with ID u1 and give it to tx-sender.
(nft-mint? my-awesome-token u1 tx-sender)

;; Transfer the NFT with ID u1 from tx-sender to another principal.
(nft-transfer? my-awesome-token u1 tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)

;; Get and print the new owner of the NFT with ID u1.
(print (nft-get-owner? my-awesome-token u1))

;; Burn the NFT with ID u1 (destroys it)
(nft-burn? my-awesome-token u1 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)
```

Click the play button on the example above or copy it to the REPL to see the NFT events emitted for each function. Try defining and minting multiple NFTs, doing transfers and burns. Like the others, these functions are safe by design. It is not possible to transfer NFTs a principal does not own, minting an NFT with the same ID twice, and burning tokens that do not exist.

Project setup

Let us create a new Clarinet project for our custom NFT contract.

```
clarinet new sip009-nft
```

The `nft-trait` trait that we will implement has an official mainnet deployment address as detailed in the SIP document. Inside the `sip009-nft` project folder, we first specify a dependency on this trait, using Clarinet's `requirements`:

```
clarinet requirements add SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait
```

That command adds the following to `Clarinet.toml`:

```
[[project.requirements]]
contract_id = "SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait"
```

Clarinet uses this information to download the contract from the network and takes care of deploying it to local or test networks for your testing.

We then create the contract that will implement our custom NFT. Give it a flashy name if you like.

```
clarinet contract new stacksies
```

Preparation work

We have dealt with traits before, so we know that we should explicitly assert conformity.

```
(impl-trait 'SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait.nft-trait)
```

Adding this line makes it impossible to deploy the contract if it does not fully implement the SIP009 trait.

Since the SIP requires the asset identifier type to be an unsigned integer, we add our NFT definition next.

```
(define-non-fungible-token stacksies uint)
```

The asset identifier should be an incrementing unsigned integer. The easiest way to implement it is to increment a counter variable each time a new NFT is minted. Let us define a data variable for it.

```
(define-data-var last-token-id uint u0)
```

Finally, we will add a constant for the contract deployer and two error codes. Here is everything put together:

```
(impl-trait 'SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait.nft-trait)

(define-constant contract-owner tx-sender)
(define-constant err-owner-only (err u100))
(define-constant err-not-token-owner (err u101))

(define-non-fungible-token stacksies uint)
```

```
(define-data-var last-token-id uint u0)
```

Implementing the SIP009 NFT trait

The implementation is rather simple, thanks to the built-in NFT functionality.

get-last-token-id

We use a variable to track the last token ID.

```
(define-read-only (get-last-token-id)
    (ok (var-get last-token-id))
)
```

get-token-uri

The idea of `get-token-uri` is to return a link to metadata for the specified NFT. Our practice NFT does not have a website so we can return `none`.

```
(define-read-only (get-token-uri (token-id uint))
    (ok none)
)
```

However, even if we did have a website, there are a few challenges to implementing this seemingly straightforward function. Usually, these functions return a base URL with the token ID stuck behind it. The current version of Clarity (2.0) does not feature a intuitive way to do this. Something like the following is impossible because there is no `to-ascii` function to turn a number into an ASCII string type.

```
(concat "https://domain.tld/metadata/" (to-ascii token-id))
```

It does not mean that turning a number into a string cannot be done. It is just way more strenuous than it should be. Clarity 2.1 will solve the issue.

get-owner

The `get-owner` function only has to wrap the built-in `nft-get-owner?`.

```
(define-read-only (get-owner (token-id uint))
    (ok (nft-get-owner? stacksies token-id))
)
```

transfer

The `transfer` function should assert that the `sender` is equal to the `tx-sender` to prevent principals from transferring tokens they do not own.

```
(define-public (transfer (token-id uint) (sender principal) (recipient principal))
    (begin
        (asserts! (is-eq tx-sender sender) err-not-token-owner)
```

```
        (nft-transfer? stacksies token-id sender recipient)
    )
)
```

mint

We will also add a convenience function to mint new tokens. A simple guard to check if the `tx-sender` is equal to the `contract-owner` constant will prevent others from minting new tokens. The function will increment the last token ID and then mint a new token for the recipient.

```
(define-public (mint (recipient principal))
    (let
        (
            (token-id (+ (var-get last-token-id) u1))
        )
        (asserts! (is-eq tx-sender contract-owner) err-owner-only)
        (try! (nft-mint? stacksies token-id recipient))
        (var-set last-token-id token-id)
        (ok token-id)
    )
)
```

Manual testing

Check if the contract conforms to SIP009 with `clarinet check`. We then enter a console session `clarinet console` and try to mint a token for ourselves.

```
>> (contract-call? .stacksies mint tx-sender)
Events emitted
{"type":"nft_mint_event","nft_mint_event":
{"asset_identifier":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.stacksies::stacksies","r
(ok u1)
```

You can see the NFT mint event and the resulting `ok` response. We can transfer the newly minted token with ID `u1` to a different principal.

```
>> (contract-call? .stacksies transfer u1 tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.stacksies::stacksies","s
(ok true)
```

`get-owner` confirms that the token is now owned by the specified principal.

```
>> (contract-call? .stacksies get-owner u1)
(ok (some ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK))
```

That is all there is to it! NFTs in Clarity are really quite easy to do. The full source code of the project can be found here: https://github.com/clarity-lang/book/tree/main/projects/sip009-nft.

SIP010: the FT standard

(The full SIP can be found here: https://github.com/stacksgov/sips/blob/main/sips/sip-010/sip-010-fungible-token-standard.md.)

Fungible tokens came after for Stacks. Indicative of the numbering, NFTs were ratified first (in SIP009), and FTs followed in SIP010. We discussed what non-fungibility meant in the last section and can thus deduce what a fungible token is supposed to be. These are Bitcoins, STX tokens, US dollars, and so on. If you trade 1 BTC for 1 BTC with somebody else, you end up just where you started (apart from perhaps a loss in transaction fees); in other words, trading the same amount of fungible tokens is an equivalent exchange.

These kinds of tokens are known as *ERC-20* tokens in the Ethereum space. Because they are so fundamental, Clarity also features built-in functions to define fungible tokens. The SIP010 standard defines the interface that allows for interoperability, just like how SIP009 does it for NFTs.

The SIP010 FT trait

Creating a SIP010-compliant fungible token also comes down implementing a trait. The trait has a few more features as fungible tokens may be divisible—just like how you have cents to a dollar—and may have a maximum supply.

```
(define-trait sip010-ft-trait
  (
    ;; Transfer from the caller to a new principal
    (transfer (uint principal principal (optional (buff 34))) (response bool uint))

    ;; the human readable name of the token
    (get-name () (response (string-ascii 32) uint))

    ;; the ticker symbol, or empty if none
    (get-symbol () (response (string-ascii 32) uint))

    ;; the number of decimals used, e.g. 6 would mean 1_000_000 represents 1 token
    (get-decimals () (response uint uint))

    ;; the balance of the passed principal
    (get-balance (principal) (response uint uint))

    ;; the current total supply (which does not need to be a constant)
    (get-total-supply () (response uint uint))

    ;; an optional URI that represents metadata of this token
    (get-token-uri () (response (optional (string-utf8 256)) uint))
    )
  )
```

transfer

A public function that transfers an amount of tokens from one principal to another. If the balance is insufficient, then it must return an `err` response. The transfer may optionally include a memo which is to be emitted using `print`. Memos are useful for off-chain indexers and apps like exchanges. If a memo is present, it should be unwrapped and emitted *after* the token transfer.

get-name

A read-only function that returns a human-readable name of the token. The name may then be used in other contracts or off-chain apps.

get-symbol

A read-only functions that returns the ticker symbol for the token. Like how the ticker symbol for Stacks is STX.

get-decimals

Another read-only function that returns the number of decimals for the fungible token. For example, setting it to `u6` would indicate that the token is divisible up to six decimal spaces. One unit of the token should then be rendered as `0.000001`. This is for display purposes only as the token amount is always represented as an unsigned integer internally.

get-balance

A read-only functions that returns the the token balance of the provided principal.

get-total-supply

A read-only function that returns the current total supply of the token. The total supply need not be a static amount.

get-token-uri

A read-only function that returns a valid URI (a link) which resolves to a metadata file for the token. A preliminary structure for the metadata file defined in the SIP is as follows:

```
{
  "title": "Asset Metadata",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "Identifies the asset to which this token represents"
    },
    "description": {
      "type": "string",
      "description": "Describes the asset to which this token represents"
    },
    "image": {
      "type": "string",
      "description": "A URI pointing to a resource with mime type image/*
representing the asset to which this token represents. Consider making any images at
a width between 320 and 1080 pixels and aspect ratio between 1.91:1 and 4:5
inclusive."
    }
  }
}
```

The function should return `none` if the token does not have any off-chain metadata. It is also important to note that the return type for the function contains a `(string-utf8 256)` as opposed to

a `(string-ascii 256)` string type like we saw in SIP009.

Creating a SIP010 fungible token

We are getting the hang of this implementation stuff. Let us now implement a SIP010 fungible token.

Built-in FT functions

Fungible tokens can also easily be defined using built-in functions. The counterpart for fungible tokens is the `define-fungible-token` function. Just like NFTs, fungible tokens have a unique asset name per contract.

```
(define-fungible-token asset-name maximum-supply)
```

An optional maximum total supply can be defined by provider an unsigned integer as the second parameter. If left out, the token has no maximum total supply. Setting a maximum total supply ensures that no more than the provided amount can ever be minted.

The expected functions to manage fungible tokens follow the same naming schedule:

```
;; Define clarity-coin with a maximum of 1,000,000 tokens.
(define-fungible-token clarity-coin u1000000)

;; Mint 1,000 tokens and give them to tx-sender.
(ft-mint? clarity-coin u1000 tx-sender)

;; Transfer 500 tokens from tx-sender to another principal.
(ft-transfer? clarity-coin u500 tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)

;; Get and print the token balance of tx-sender.
(print (ft-get-balance clarity-coin tx-sender))

;; Burn 250 tokens (destroys them)
(ft-burn? clarity-coin u250 'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)

;; Get and print the circulating supply
(print (ft-get-supply clarity-coin))
```

Click the play button on the example above or copy it to the REPL to see the FT events emitted for each function. Play around with the code by minting, transferring, and burning tokens. You will notice that built-in safeties prevent you from transferring more tokens than a principal owns and minting more than the maximum supply.

Project setup

Let us create a new Clarinet project for our custom NFT contract.

```
clarinet new sip010-ft
```

Inside the `sip010-ft` project folder, we first add the requirement for the trait, using the official mainnet deployment address.

```
clarinet requirements add SP3FBR2AGK5H9QBDH3EEN6DF8EK8JY7RX8QJ5SVTE.sip-010-trait-ft-standard
```

We then create the contract that will implement our custom fungible token. Another flashy name is welcome.

```
clarinet contract new clarity-coin
```

Preparation work

Asserting explicit conformity with the trait is the first step as usual.

```
(impl-trait 'SP3FBR2AGK5H9QBDH3EEN6DF8EK8JY7RX8QJ5SVTE.sip-010-trait-ft-standard.sip-010-trait)
```

We can then add the token definition, a constant for the contract deployer, and two error codes:

```
(define-constant contract-owner tx-sender)
(define-constant err-owner-only (err u100))
(define-constant err-not-token-owner (err u101))

;; No maximum supply!
(define-fungible-token clarity-coin)
```

Implementing the SIP010 FT trait

transfer

The `transfer` function should assert that the `sender` is equal to the `tx-sender` to prevent principals from transferring tokens they do not own. It should also unwrap and print the `memo` if it is not `none`. We use `match` to conditionally call `print` if the passed `memo` is a `some`.

```
(define-public (transfer (amount uint) (sender principal) (recipient principal)
(memo (optional (buff 34))))
    (begin
        (asserts! (is-eq tx-sender sender) err-not-token-owner)
        (try! (ft-transfer? clarity-coin amount sender recipient))
        (match memo to-print (print to-print) 0x)
        (ok true)
    )
)
```

Play with this snippet to see how `match` works.

```
(match (some "inner string")
    inner-str (print inner-str)
    (print "got nothing")
)
```

get-name

A static function that returns a human-readable name for our token.

```
(define-read-only (get-name)
    (ok "Clarity Coin")
)
```

get-symbol

A static function that returns a human-readable symbol for our token.

```
(define-read-only (get-symbol)
    (ok "CC")
)
```

get-decimals

As was established in the previous section, the value returned by this function is purely for display reasons. Let us follow along with STX and introduce 6 decimals.

```
(define-read-only (get-decimals)
    (ok u6)
)
```

get-balance

This function returns the balance of a specified principal. We simply wrap the built-in function that retrieves the balance.

```
(define-read-only (get-balance (who principal))
    (ok (ft-get-balance clarity-coin who))
)
```

get-total-supply

This function returns the total supply of our custom token. We again simply wrap the built-in function for it.

```
(define-read-only (get-total-supply)
    (ok (ft-get-supply clarity-coin))
)
```

get-token-uri

This function has the same purpose as `get-token-uri` in SIP009. It should return a link to a metadata file for the token. Our practice fungible token does not have a website so we can return `none`.

```
(define-read-only (get-token-uri)
    (ok none)
```

```
)
```

mint

Just like our custom NFT, we will add a convenience function to mint new tokens that only the contract deployer can successfully call.

```
(define-public (mint (amount uint) (recipient principal))
    (begin
        (asserts! (is-eq tx-sender contract-owner) err-owner-only)
        (ft-mint? clarity-coin amount recipient)
    )
)
```

Manual testing

Check if the contract conforms to SIP010 with `clarinet check`. We then enter a console session `clarinet console` and try to mint some tokens for ourselves.

```
>> (contract-call? .clarity-coin mint u1000 tx-sender)
Events emitted
{"type":"ft_mint_event","ft_mint_event":
{"asset_identifier":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.clarity-
coin::clarity-
coin","recipient":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE","amount":"1000"}}
(ok true)
```

You can see the FT mint event and the resulting `ok` response. We minted a thousand tokens for `tx-sender`. Let us try to transfer some of those to a different principal.

```
>> (contract-call? .clarity-coin transfer u250 tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK none)
Events emitted
{"type":"ft_transfer_event","ft_transfer_event":
{"asset_identifier":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.clarity-
coin::clarity-
coin","sender":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE","recipient":"ST1J4G6RR643BCG
(ok true)
```

The `none` at the end is the (absence of a) memo. We can do another transfer with a memo to see if it is printed to the screen.

```
>> (contract-call? .clarity-coin transfer u100 tx-sender
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK (some 0x123456))
Events emitted
{"type":"ft_transfer_event","ft_transfer_event":
{"asset_identifier":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.clarity-
coin::clarity-
coin","sender":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE","recipient":"ST1J4G6RR643BCG
{"type":"contract_event","contract_event":
```

```
{"contract_identifier":"ST1HTBVD3JG9C05J7HBJTHGR0GGW7KXW28M5JS8QE.clarity-
coin","topic":"print","value":"0x123456"}}
(ok true)
```

Great! We see an FT transfer event followed by a print event of `0x123456`. The recipient principal should have a total of 350 tokens by now. We can verify the balance by querying the contract:

```
>> (contract-call? .clarity-coin get-balance
'ST1J4G6RR643BCG8G8SR6M2D9Z9KXT2NJDRK3FBTK)
(ok u350)
```

The full source code of the project can be found here: https://github.com/clarity-lang/book/tree/main/projects/sip010-ft.

Building a marketplace

We now have all the tools we need to build our first full-fledged project. With everything that is happening in the crypto space right now, what better to build than an NFT marketplace? Marketplaces are immensely popular as they democratise access to digital collectibles: art, in-game items, virtual vouchers, and so much more. Blockchains provide open membership ledgers and marketplaces provide an interface to trade assets on these ledgers freely.

Features

Our marketplace, which we shall call Tiny Market, will be a minimal implementation that allows people to trustlessly list NFTs for sale. An NFT offering will contain the following information:

- The NFT being offered for sale.
- A block height at which the listing expires.
- The payment asset, either STX or a SIP010 fungible token. This allows the seller to request payment in a fungible token other than STX.
- The NFT price in said payment asset.
- An optional intended taker. If set, only that principal will be able to fulfil the listing. This is useful if someone found a buyer outside of the platform and wants to trustlessly conduct a trade with just that person.

There will be no bidding or other matters that complicate the project. A listing is created by calling into the marketplace contract with the above information. The contract will then transfer the NFT and keep it in escrow for the duration of the listing. The seller can at any point decide to cancel the listing and retrieve the NFT. If someone decides to purchase the NFT, the marketplace will atomically take the payment from the buyer and transfer the NFT. Once a listing has expired, only the seller will be able to retrieve the NFT from the marketplace.

The full source code of the project can be found here: https://github.com/clarity-lang/book/tree/main/projects/tiny-market.

Setup

We start our project setup as usual: `clarinet new tiny-market`. Once inside the project folder, we will create the tiny market contract using `clarinet contract new tiny-market`. Since our marketplace revolves around selling NFTs, the first thing we have to do is add the SIP009 trait and create a SIP009 NFT. We already made a few of these so we will leave it as a challenge to the reader. Make sure the SIP009 contract is called `sip009-nft-trait` and the NFT contract `sip009-nft`. You may implement the contract any way you like as long it has a mint function that the contract deployer can call. We will use it later for our unit tests. Here is an example:

```
(define-public (mint (recipient principal))
    (let ((token-id (+ (var-get token-id-nonce) u1)))
        (asserts! (is-eq tx-sender contract-owner) err-owner-only)
        (try! (nft-mint? stacksies token-id recipient))
        (asserts! (var-set token-id-nonce token-id) err-token-id-failure)
        (ok token-id)
    )
)
```

Once you have done that, let us add the SIP010 trait and create a SIP010 token as well. We will call these `sip010-ft-trait` and `sip010-token`. The SIP010 token should also have a mint function that only the contract deployer can call.

```
(define-public (mint (amount uint) (recipient principal))
    (begin
        (asserts! (is-eq tx-sender contract-owner) err-owner-only)
        (ft-mint? amazing-coin amount recipient)
    )
)
```

Trait imports & constants

Let us now work on `tiny-market`. We will start off by importing the SIP traits. Be sure to check the naming and change them if you used different names. Do not forget to also add the corresponding `requirements` to the project like you have learned. We also define a constant for the contract owner.

```
(use-trait nft-trait 'SP2PABAF9FTAJYNFZH93XENAJ8FVY99RRM50D2JG9.nft-trait.nft-trait)
(use-trait ft-trait 'SP3FBR2AGK5H9QBDH3EEN6DF8EK8JY7RX8QJ5SVTE.sip-010-trait-ft-
standard.sip-010-trait)

(define-constant contract-owner tx-sender)
```

We will then think about the various error states that exist in our marketplace. The act of listing an NFT may fail under a number of circumstances; namely, the expiry block height is in the past, or the listing price is zero (we will not allow free listings). Additionally, there is the possibility that a user may try to list an NFT for sale without actually owning it. However, this issue will be addressed by the NFT contract's built-in safeguards. You may remember that the built-in NFT functions fail with an error code if the NFT does not exist or if it is not owned by `tx-sender`. We will simply propagate those errors using control flow functions. We therefore only define two listing error codes:

```
;; listing errors
(define-constant err-expiry-in-past (err u1000))
(define-constant err-price-zero (err u1001))
```

When it comes to cancelling and fulfilling, there are a few more error conditions we can identify:

- The listing the `tx-sender` wants to cancel or fulfil does not exist.
- The `tx-sender` tries to cancel a listing it did not create.
- The listing the `tx-sender` tries to fill has expired.
- The provided NFT asset trait reference does not match the NFT contract of the listing. Since trait references cannot be stored directly in Clarity, they will have to be provided again when the buyer is trying to purchase an NFT. We have to make sure that the trait reference provided by the buyer matches the NFT contract provided by the seller.
- The provided payment asset trait reference does not match the payment asset contract of the listing. The same as the above but for the SIP010 being used to purchase the NFT.
- The maker and the taker (seller and the buyer) are equal. We will not permit users to purchase tokens from themselves using the same principal.
- The buyer is not the intended taker. If the seller defines an intended taker (buyer) for the listing, then only that principal can fulfil the listing.

Finally, we will implement a whitelist for NFT and payment asset contracts that the contract deployer controls. It makes for two additional error conditions:

- The NFT asset the seller is trying to list is not whitelisted.
- The requested payment asset is not whitelisted.

Turning all of these into unique error constants, we get something like the following:

```
;; cancelling and fulfilling errors
(define-constant err-unknown-listing (err u2000))
(define-constant err-unauthorised (err u2001))
(define-constant err-listing-expired (err u2002))
(define-constant err-nft-asset-mismatch (err u2003))
(define-constant err-payment-asset-mismatch (err u2004))
(define-constant err-maker-taker-equal (err u2005))
(define-constant err-unintended-taker (err u2006))
(define-constant err-asset-contract-not-whitelisted (err u2007))
(define-constant err-payment-contract-not-whitelisted (err u2008))
```

Data storage

The marketplace itself only has to store a little information regarding the listing. The most efficient way to store the individual listings is by using a data map that uses an unsigned integer as a key. The integer functions as a unique identifier and will increment for each new listing. We will never reuse a value. To track the latest listing ID, we will use a simple data variable.

```
(define-map listings
    uint
    {
        maker: principal,
        taker: (optional principal),
```

```
        token-id: uint,
        nft-asset-contract: principal,
        expiry: uint,
        price: uint,
        payment-asset-contract: (optional principal)
    }
)

(define-data-var listing-nonce uint u0)
```

It is important to utilise the native types in Clarity to the fullest extent possible. A listing does not need to have an intended taker, so we make it `optional`. The same goes for the payment asset. If the seller wants to be paid in STX, then there is no payment asset. If the seller wants to be paid using a SIP010 token, then its token contract will be stored.

Asset whitelist

We will implement an asset whitelist to keep our marketplace safe. Only the contract owner will have the ability to modify the whitelist. The whitelist itself is a simple map that stores a boolean for a given contract principal. A guarded public function `set-whitelisted` is used to update the whitelist and a read-only function `is-whitelisted` allows anyone to check if a particular contract is whitelisted or not. We will also use `is-whitelisted` to guard other public functions later.

```
(define-map whitelisted-asset-contracts principal bool)

(define-read-only (is-whitelisted (asset-contract principal))
    (default-to false (map-get? whitelisted-asset-contracts asset-contract))
)

(define-public (set-whitelisted (asset-contract principal) (whitelisted bool))
    (begin
        (asserts! (is-eq contract-owner tx-sender) err-unauthorised)
        (ok (map-set whitelisted-asset-contracts asset-contract whitelisted))
    )
)
```

Listing & cancelling sales

Our marketplace functions as an escrow contract at its bare essence. It takes someone's NFT, along with some conditions, and releases it when these conditions are met. One of those conditions is providing right payment tokens. Most of the functionality thus involves tokens, which means it useful to create some helper functions that transfer NFTs and fungible tokens. The functions will take a trait reference (either SIP009 or SIP010) and then do the proper `contract-call?` to transfer the token.

```
(define-private (transfer-nft (token-contract <nft-trait>) (token-id uint) (sender principal) (recipient principal))
    (contract-call? token-contract transfer token-id sender recipient)
)

(define-private (transfer-ft (token-contract <ft-trait>) (amount uint) (sender principal) (recipient principal))
    (contract-call? token-contract transfer amount sender recipient none)
)
```

We will use these functions to implement the rest of the marketplace.

Listing an NFT

Principals will call into a function `list-asset` to put their NFT up for sale. The call will have to include a trait reference and a tuple that contains the information to store in the `listing` map we came up with in the previous section. The flow of the listing function will go something like this:

1. Retrieve the current listing ID to use by reading the `listing-nonce` variable.
2. Assert that the NFT asset is whitelisted.
3. Assert that the provided expiry height is somewhere in the future.
4. Assert that the listing price is larger than zero.
5. If a payment asset is given, assert that it is whitelisted.
6. Transfer the NFT from the `tx-sender` to the marketplace.
7. Store the listing information in the `listings` data map.
8. Increment the `listing-nonce` variable.
9. Return an `ok` to materialise the changes.

We will return the listing ID when everything goes well as a convenience for frontends and other contracts that interact with the marketplace.

```
(define-public (list-asset (nft-asset-contract <nft-trait>) (nft-asset {taker: (optional principal), token-id: uint, expiry: uint, price: uint, payment-asset-contract: (optional principal)}))
    (let ((listing-id (var-get listing-nonce)))
        (asserts! (is-whitelisted (contract-of nft-asset-contract)) err-asset-contract-not-whitelisted)
        (asserts! (> (get expiry nft-asset) block-height) err-expiry-in-past)
        (asserts! (> (get price nft-asset) u0) err-price-zero)
        (asserts! (match (get payment-asset-contract nft-asset) payment-asset (is-whitelisted payment-asset) true) err-payment-contract-not-whitelisted)
        (try! (transfer-nft nft-asset-contract (get token-id nft-asset) tx-sender (as-contract tx-sender)))
        (map-set listings listing-id (merge {maker: tx-sender, nft-asset-contract:
```

```
(contract-of nft-asset-contract)} nft-asset))
        (var-set listing-nonce (+ listing-id u1))
        (ok listing-id)
    )
)
```

Pay close attention to the `map-set` expression. Notice the `merge`? It merges two tuples together, which in this case is the information provided by the user, plus a tuple that sets the `maker` key to `tx-sender` and the `nft-asset-contract` to a value equal to `(contract-of nft-asset-contract)`. The `contract-of` expression takes a principal passed via a trait reference and turns it into a generic `principal` type. We do this because a trait reference `<nft-trait>` cannot be stored in a data map. You will see why this is useful later.

Finally, we create a read-only function that returns a listing by ID as usual.

```
(define-read-only (get-listing (listing-id uint))
    (map-get? listings listing-id)
)
```

Cancelling a listing

A listing is available until it either expires or is cancelled by the maker. When the maker cancels the listing, all that has to happen is for the marketplace to send the NFT back and delete the listing from the data map. The maker only has to provide the listing ID and the NFT asset contract trait reference. The rest can be read from the data map.

```
(define-public (cancel-listing (listing-id uint) (nft-asset-contract <nft-trait>))
    (let (
        (listing (unwrap! (map-get? listings listing-id) err-unknown-listing))
        (maker (get maker listing))
        )
        (asserts! (is-eq maker tx-sender) err-unauthorised)
        (asserts! (is-eq (get nft-asset-contract listing) (contract-of nft-asset-
contract)) err-nft-asset-mismatch)
        (map-delete listings listing-id)
        (as-contract (transfer-nft nft-asset-contract (get token-id listing) tx-
sender maker))
    )
)
```

See how we check if the trait reference is equal to the contract principal stored in the listing? Even though we cannot store trait references, we can still verify that the expected trait reference was passed.

Testing listing and cancelling

Let us perform a quick manual test to see if it all works as expected. We can drop into a `clarinet console` session and perform the steps manually.

We first mint an NFT for ourselves.

```
>> (contract-call? .sip009-nft mint tx-sender)
Events emitted
{"type":"nft_mint_event","nft_mint_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","recipient":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","value":"u1"}}
(ok u1)
```

We will also verify if we actually own it.

```
>> (contract-call? .sip009-nft get-owner u1)
(ok (some ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM))
```

Before we can list it, we have to whitelist the NFT contract.

```
>> (contract-call? .tiny-market set-whitelisted .test-sip009 true)
(ok true)
```

Then we try to list the NFT for sale on the marketplace for an amount of `u1000`. We will set the expiry to block 500 and have no intended taker or payment asset contract.

```
>> (contract-call? .tiny-market list-asset .sip009-nft {taker: none, token-id: u1,
expiry: u500, price: u1000, payment-asset-contract: none})
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","sender":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","recipient":"ST1PQ
market","value":"u1"}}
(ok u0)
```

Great! Looks like that worked. It returned an `ok` value with the listing ID, which for the first one is naturally `u0`. We can then retrieve that listing:

```
>> (contract-call? .tiny-market get-listing u0)
(some {expiry: u500, maker: ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM, nft-asset-
contract: ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-nft, payment-asset-
contract: none, price: u1000, taker: none, token-id: u1})
```

Who now owns the NFT? We can check by querying the NFT contract again.

```
>> (contract-call? .sip009-nft get-owner u1)
(ok (some ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-market))
```

Looks like it is owned by the marketplace contract as expected!

For fun, let us see what happens if we try to list an NFT we do not own. For example with a bogus token ID of 555.

```
>> (contract-call? .tiny-market list-asset .sip009-nft {taker: none, token-id: u555,
expiry: u500, price: u1000, payment-asset-contract: none})
(err u3)
```

And finally, we can cancel the listing and get the NFT back.

```
>> (contract-call? .tiny-market cancel-listing u0 .sip009-nft)
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","sender":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-
market","recipient":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","value":"u1"}}
(ok true)
```

And we can see it was transferred back to us.

Fulfilling listings

Now we get to the most exciting part, the ability to purchase NFTs. There are only a few steps the markeplace goes through to validate a purchase and send the tokens to the proper recipients. We have to handle two different payment methodologies, payment in STX and in SIP010 tokens. The logical flow will go like this:

1. Retrieve the listing from the `listings` data map and abort if it does not exist.
2. Assert that the taker is not equal to the maker.
3. Assert that the expiry block height has not been reached.
4. Assert that the provided NFT trait reference is equal to the principal stored in the listing.
5. Assert that the payment asset trait reference, if any, is equal to the one stored in the listing.
6. Transfer the NFT from the contract to the buyer and the payment asset from the buyer to the seller and revert if either transfer fails.
7. Delete the listing from the `listings` data map.

Most of the steps above are shared between both payment methedologie (step 7). It therefore makes sense to create a single function that checks if all fulfilment conditions are met. We will call this function `assert-can-fulfil`. It will take the NFT asset contract principal, payment asset contract principal, and listing tuple as parameters in order to validate all conditions.

```
(define-private (assert-can-fulfil (nft-asset-contract principal) (payment-asset-
contract (optional principal)) (listing {maker: principal, taker: (optional
principal), token-id: uint, nft-asset-contract: principal, expiry: uint, price:
uint, payment-asset-contract: (optional principal)}))
    (begin
        (asserts! (not (is-eq (get maker listing) tx-sender)) err-maker-taker-equal)
        (asserts! (match (get taker listing) intended-taker (is-eq intended-taker
tx-sender) true) err-unintended-taker)
        (asserts! (< block-height (get expiry listing)) err-listing-expired)
        (asserts! (is-eq (get nft-asset-contract listing) nft-asset-contract) err-
nft-asset-mismatch)
        (asserts! (is-eq (get payment-asset-contract listing) payment-asset-
contract) err-payment-asset-mismatch)
        (ok true)
    )
)
```

The function signature looks a bit unwieldly due to the listing tuple type definition. Remember that you can use whitespace to make it a bit more readable if you like.

Fulfilment in STX

With our helper function, implementing the fulfilment functions is easy. We can call into it at the start and propagate any errors using `try!`. If `assert-can-fulfil` returns an `ok` then we know we can move on to tranferring the assets to the buyer and the seller. We have to make sure to delete the listing from the data map to prevent people from trying to fulfil the listing more than once. It is just for good form, because even if that were to happen, the contract call would not complete because the marketplace would no longer possess the NFT.

```
(define-public (fulfil-listing-stx (listing-id uint) (nft-asset-contract <nft-
trait>))
```

```
    (let (
        (listing (unwrap! (map-get? listings listing-id) err-unknown-listing))
        (taker tx-sender)
        )
        (try! (assert-can-fulfil (contract-of nft-asset-contract) none listing))
        (try! (as-contract (transfer-nft nft-asset-contract (get token-id listing)
tx-sender taker)))
        (try! (stx-transfer? (get price listing) taker (get maker listing)))
        (map-delete listings listing-id)
        (ok listing-id)
    )
)
```

Returning the listing ID is again useful for a contract or frontend interacting with the marketplace.

Fulfilment in a SIP010 fungible token

We can now pretty much copy the previous function to create the SIP010 version. Instead of calling `stx-transfer?`, we call the `transfer-ft` function we made earlier.

```
(define-public (fulfil-listing-ft (listing-id uint) (nft-asset-contract <nft-trait>)
(payment-asset-contract <ft-trait>))
    (let (
        (listing (unwrap! (map-get? listings listing-id) err-unknown-listing))
        (taker tx-sender)
        )
        (try! (assert-can-fulfil (contract-of nft-asset-contract) (some (contract-of
payment-asset-contract)) listing))
        (try! (as-contract (transfer-nft nft-asset-contract (get token-id listing)
tx-sender taker)))
        (try! (transfer-ft payment-asset-contract (get price listing) taker (get
maker listing)))
        (map-delete listings listing-id)
        (ok listing-id)
    )
)
```

Testing STX order fulfilment

We are now feature-complete. Let us test order fulfilment manually before we start working on unit tests. We again drop into a `clarinet console` session and set the stage to conduct a trade.

First we mint an NFT and list it on the marketplace for 150 mSTX.

```
>> (contract-call? .sip009-nft mint tx-sender)
Events emitted
{"type":"nft_mint_event","nft_mint_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","recipient":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","value":"u1"}}
(ok u1)
>> (contract-call? .tiny-market list-asset .sip009-nft {taker: none, token-id: u1,
```

```
expiry: u500, price: u150, payment-asset-contract: none})
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","sender":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","recipient":"ST1PQ
market","value":"u1"}}
(ok u0)
```

You might remember that we cannot purchase an NFT from ourselves. Let us try that out first by trying to buy it right now. As seen above, the listing ID for our NFT is `u0`. The function we should call into is `fulfil-listing-stx`.

```
>> (contract-call? .tiny-market fulfil-listing-stx u0 .sip009-nft)
(err u2005)
```

An error was emitted; namely, `u2005`! Looking at our list of error constants, we find that it is indeed the maker-taker equality check that has failed: `err-maker-taker-equal`.

Clarinet makes it really easy to assume any `tx-sender`. We can pick the next wallet in the list and set it as the `tx-sender` using `::set_tx_sender`. Once we have done that we will again try to purchase the NFT from the marketplace. Remember that once we change the `tx-sender`, the shorthand contract notation for the marketplace and NFT contract will no longer work! You will have to write out the fully qualified contract principal.

```
>> ::set_tx_sender ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5
tx-sender switched to ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5
>> (contract-call? 'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-market fulfil-
listing-stx u0 'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-nft)
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","sender":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-
market","recipient":"ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5","value":"u1"}}
{"type":"stx_transfer_event","stx_transfer_event":
{"sender":"ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5","recipient":"ST1PQHQKV0RJXZFY1DGX
(ok u0)
```

Will you look at that! There is an NFT asset transfer from the marketplace contract to the `tx-sender`, and a STX transfer from the `tx-sender` to the original maker of the NFT listing. We can check the asset maps to verify that the balance of the maker has increased by 150 mSTX, the balance of the taker has decreased by the same amount, and that the taker now also owns one of our test NFTs. The marketplace no longer owns any tokens.

```
>> ::get_assets_maps
+-----------------------------------------------------+----------------------+---
--------------+
| Address                                             | .sip009-nft.stacksies |
STX             |
+-----------------------------------------------------+----------------------+---
--------------+
```

```
| ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM (deployer)  | 0                     |
100000000000150 |
+-----------------------------------------------------+-----------------------+---
--------------+
| ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-market | 0                     | 0
|
+-----------------------------------------------------+-----------------------+---
--------------+
| ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5 (wallet_1)  | 1                     |
99999999999850  |
+-----------------------------------------------------+-----------------------+---
--------------+
```

Testing SIP010 order fulfilment

Let us also test fulfilling an order with SIP010 tokens for good measure. Start with a fresh `clarinet session` and mint another NFT as the contract deployer.

```
>> (contract-call? .sip009-nft mint tx-sender)
Events emitted
{"type":"nft_mint_event","nft_mint_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","recipient":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","value":"u1"}}
(ok u1)
```

Next, we mint some SIP010 tokens to *another* standard principal. That other principal is going to be the taker of the order. We will mint 1,000 tokens.

```
>> (contract-call? .sip010-token mint u1000
'ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5)
Events emitted
{"type":"ft_mint_event","ft_mint_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip010-
token::amazing-
coin","recipient":"ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5","amount":"1000"}}
(ok true)
```

Now we are ready to list the NFT. We will charge 800 tokens in our test. Be sure to pass the SIP010 token contract as the `payment-asset-contract`. (And remember, it is an `optional` type, so wrap the contract principal in a `some`.)

```
>> (contract-call? .tiny-market list-asset .sip009-nft {taker: none, token-id: u1,
expiry: u500, price: u800, payment-asset-contract: (some .sip010-token)})
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","sender":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM","recipient":"ST1PQ
market","value":"u1"}}
(ok u0)
```

Looking good so far, our listing was accepted and the NFT transferred to the marketplace. The listing ID is again `u0` because we started a new Clarinet session.

We can now switch the `tx-sender` to the principal we minted SIP010 tokens to earlier. For fun, we can see if we can fulfil the listing using STX tokens instead.

```
>> ::set_tx_sender ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5
tx-sender switched to ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5
>> (contract-call? 'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-market fulfil-
listing-stx u0 'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-nft)
(err u2004)
```

We get an error `u2004` which is `err-payment-asset-mismatch`, and no asset transfer events. Good news! Now let us actually buy it with the proper payment asset by calling into `fulfil-listing-ft`.

```
>> (contract-call? 'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-market fulfil-
listing-ft u0 'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-nft
'ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip010-token)
Events emitted
{"type":"nft_transfer_event","nft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip009-
nft::stacksies","sender":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-
market","recipient":"ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5","value":"u1"}}
{"type":"ft_transfer_event","ft_transfer_event":
{"asset_identifier":"ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.sip010-
token::amazing-
coin","sender":"ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5","recipient":"ST1PQHQKV0RJXZF
(ok u0)
```

A bunch of transfers happened and we got a positive response. Taking a peek at the asset maps show us that all went well. The maker received 800 SIP010 tokens while the taker kept 200 and received the NFT. All STX balances remained unaffected.

```
>> ::get_assets_maps
+-------------------------------------------------------+----------------------+---
------------------------+-----------------+
| Address                                               | .sip009-nft.stacksies |
.sip010-token.amazing-coin | STX             |
+-------------------------------------------------------+----------------------+---
------------------------+-----------------+
| ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM (deployer)  | 0                    |
800                        | 100000000000000 |
+-------------------------------------------------------+----------------------+---
------------------------+-----------------+
| ST1PQHQKV0RJXZFY1DGX8MNSNYVE3VGZJSRTPGZGM.tiny-market | 0                    | 0
| 0                |
+-------------------------------------------------------+----------------------+---
------------------------+-----------------+
| ST1SJ3DTE5DN7X54YDH5D64R3BCB6A2AG2ZQ8YPD5 (wallet_1)  | 1                    |
200                        | 100000000000000 |
```

```
+-------------------------------------------------------+----------------------+---
-------------------------+-----------------+
```

Unit tests

The `tiny-market` contract is more sizeable than what we have built before. We will make an effort to structure the unit tests to avoid repetition. Common actions are turned into helper functions so that we can call them when needed. These include:

- Turning an `Account` and contract name into a contract principal string.
- Minting a new token for testing, both for NFTs and payment assets.
- Asserting that an NFT token transfer has happened. (Clarinet does not support this yet.)
- Creating an order tuple.
- Creating a transaction for common actions like whitelisting an asset contract or listing an NFT.

Token minting helpers

To prevent hard-coding contract names in our tests, we define a constant for each. The helper functions will construct the contract call to the `mint` function and mine a block to include it. The chain and deployer parameters will be passed in by the tests themselves. The functions then return some helpful information like the NFT asset contract principal, the token ID or amount, and the block data itself.

```
const contractName = "tiny-market";

const defaultNftAssetContract = "sip009-nft";
const defaultPaymentAssetContract = "sip010-token";

const contractPrincipal = (deployer: Account) =>
  `${deployer.address}.${contractName}`;

function mintNft(
  { chain, deployer, recipient, nftAssetContract = defaultNftAssetContract }: {
    chain: Chain;
    deployer: Account;
    recipient: Account;
    nftAssetContract?: string;
  },
) {
  const block = chain.mineBlock([
    Tx.contractCall(nftAssetContract, "mint", [
      types.principal(recipient.address),
    ], deployer.address),
  ]);
  block.receipts[0].result.expectOk();
  const nftMintEvent = block.receipts[0].events[0].nft_mint_event;
  const [nftAssetContractPrincipal, nftAssetId] = nftMintEvent.asset_identifier
    .split("::");
  return {
    nftAssetContract: nftAssetContractPrincipal,
    nftAssetId,
    tokenId: nftMintEvent.value.substr(1),
    block,
  };
}
```

```
function mintFt(
  {
    chain,
    deployer,
    amount,
    recipient,
    paymentAssetContract = defaultPaymentAssetContract,
  }: {
    chain: Chain;
    deployer: Account;
    amount: number;
    recipient: Account;
    paymentAssetContract?: string;
  },
) {
  const block = chain.mineBlock([
    Tx.contractCall(paymentAssetContract, "mint", [
      types.uint(amount),
      types.principal(recipient.address),
    ], deployer.address),
  ]);
  block.receipts[0].result.expectOk();
  const ftMintEvent = block.receipts[0].events[0].ft_mint_event;
  const [paymentAssetContractPrincipal, paymentAssetId] = ftMintEvent
    .asset_identifier.split("::");
  return {
    paymentAssetContract: paymentAssetContractPrincipal,
    paymentAssetId,
    block,
  };
}
```

Asserting NFT transfers

NFT transfer events are emitted by Clarinet but no function exists to assert their existence. We therefore make our own and define a basic interface that describes the transfer event. It will check that an event with the expected properties exists: the right NFT asset contract, token ID, and principal.

```
interface Sip009NftTransferEvent {
  type: string;
  nft_transfer_event: {
    asset_identifier: string;
    sender: string;
    recipient: string;
    value: string;
  };
}

function assertNftTransfer(
  event: Sip009NftTransferEvent,
  nftAssetContract: string,
```

```
  tokenId: number,
  sender: string,
  recipient: string,
) {
  assertEquals(typeof event, "object");
  assertEquals(event.type, "nft_transfer_event");
  assertEquals(
    event.nft_transfer_event.asset_identifier.substr(
      0,
      nftAssetContract.length,
    ),
    nftAssetContract,
  );
  event.nft_transfer_event.sender.expectPrincipal(sender);
  event.nft_transfer_event.recipient.expectPrincipal(recipient);
  event.nft_transfer_event.value.expectUint(tokenId);
}
```

Order tuple helper

Since listing assets by calling into `list-asset` is something that we will do quite often, we will also create a helper function that constructs the `net-asset` order tuple. The function takes an object with properties equal to that of the tuple, just camel-cased instead of with dashes.

```
interface Order {
  taker?: string;
  tokenId: number;
  expiry: number;
  price: number;
  paymentAssetContract?: string;
}

const makeOrder = (order: Order) =>
  types.tuple({
    "taker": order.taker
      ? types.some(types.principal(order.taker))
      : types.none(),
    "token-id": types.uint(order.tokenId),
    "expiry": types.uint(order.expiry),
    "price": types.uint(order.price),
    "payment-asset-contract": order.paymentAssetContract
      ? types.some(types.principal(order.paymentAssetContract))
      : types.none(),
  });
```

Whitelisting transaction

The helper to create a whitelisting transaction is a one-liner. It takes the asset contract to whitelist and whether it should be whitelisted, finally the the contract owner that should send the transaction, as it is a guarded function.

```
const whitelistAssetTx = (
  assetContract: string,
  whitelisted: boolean,
  contractOwner: Account,
) =>
  Tx.contractCall(contractName, "set-whitelisted", [
    types.principal(assetContract),
    types.bool(whitelisted),
  ], contractOwner.address);
```

Listing an NFT

Listing a new order is likewise a one-liner. We will also make it so that you can pass in both an `Order` object or an order tuple (which is represented as a string by Clarinet). If an `Order` is passed, all we have to do is call the `makeOrder` helper.

```
const listOrderTx = (
  nftAssetContract: string,
  maker: Account,
  order: Order | string,
) =>
  Tx.contractCall(contractName, "list-asset", [
    types.principal(nftAssetContract),
    typeof order === "string" ? order : makeOrder(order),
  ], maker.address);
```

Listing tests

We can then use the helpers to construct our first tests: listing an NFT for sale for STX and for SIP010 fungible tokens.

```
Clarinet.test({
  name: "Can list an NFT for sale for STX",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    block.receipts[1].result.expectOk().expectUint(0);
    assertNftTransfer(
      block.receipts[1].events[0],
```

```
      nftAssetContract,
      tokenId,
      maker.address,
      contractPrincipal(deployer),
    );
  },
});

Clarinet.test({
  name: "Can list an NFT for sale for any SIP010 fungible token",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const { paymentAssetContract } = mintFt({
      chain,
      deployer,
      recipient: maker,
      amount: 1,
    });
    const order: Order = {
      tokenId,
      expiry: 10,
      price: 10,
      paymentAssetContract,
    };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      whitelistAssetTx(paymentAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    block.receipts[2].result.expectOk().expectUint(0);
    assertNftTransfer(
      block.receipts[2].events[0],
      nftAssetContract,
      tokenId,
      maker.address,
      contractPrincipal(deployer),
    );
  },
});
```

Invalid listings

A listing call should fail under the following circumstances:

- The expiry block height of the order is in the past.

- The NFT is being listed for nothing. (A price of zero.)
- Someone is trying to list an NFT for sale that the sender does not own.

```
Clarinet.test({
  name: "Cannot list an NFT for sale if the expiry is in the past",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const expiry = 10;
    const order: Order = { tokenId, expiry, price: 10 };
    chain.mineEmptyBlockUntil(expiry + 1);
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    block.receipts[1].result.expectErr().expectUint(1000);
    assertEquals(block.receipts[1].events.length, 0);
  },
});

Clarinet.test({
  name: "Cannot list an NFT for sale for nothing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 0 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    block.receipts[1].result.expectErr().expectUint(1001);
    assertEquals(block.receipts[1].events.length, 0);
  },
});

Clarinet.test({
  name: "Cannot list an NFT for sale that the sender does not own",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
```

```
      (name) => accounts.get(name)!,
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: taker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    block.receipts[1].result.expectErr().expectUint(1);
    assertEquals(block.receipts[1].events.length, 0);
  },
});
```

Cancelling listings

Only the maker can cancel an active listing, we will cover this with two tests.

```
Clarinet.test({
  name: "Maker can cancel a listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "cancel-listing", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], maker.address),
    ]);
    block.receipts[2].result.expectOk().expectBool(true);
    assertNftTransfer(
      block.receipts[2].events[0],
      nftAssetContract,
      tokenId,
      contractPrincipal(deployer),
      maker.address,
    );
  },
});
```

```
Clarinet.test({
  name: "Non-maker cannot cancel listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, otherAccount] = ["deployer", "wallet_1", "wallet_2"]
      .map((name) => accounts.get(name)!);
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "cancel-listing", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], otherAccount.address),
    ]);
    block.receipts[2].result.expectErr().expectUint(2001);
    assertEquals(block.receipts[2].events.length, 0);
  },
});
```

Retrieving listings

Listings can be retrieved until they are cancelled. We will add a test that retrieves an active listing and make sure that the returned tuple contains the information we expect. The next test verifies that retrieving a listing that is cancelled or does not exist returns `none`.

```
Clarinet.test({
  name: "Can get listings that have not been cancelled",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    const listingIdUint = block.receipts[1].result.expectOk();
    const receipt = chain.callReadOnlyFn(contractName, "get-listing", [
      listingIdUint,
    ], deployer.address);
```

```
    const listing: { [key: string]: string } = receipt.result.expectSome()
      .expectTuple() as any;

    listing["expiry"].expectUint(order.expiry);
    listing["maker"].expectPrincipal(maker.address);
    listing["payment-asset-contract"].expectNone();
    listing["price"].expectUint(order.price);
    listing["taker"].expectNone();
    listing["nft-asset-contract"].expectPrincipal(nftAssetContract);
    listing["token-id"].expectUint(tokenId);
  },
});

Clarinet.test({
  name: "Cannot get listings that have been cancelled or do not exist",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    chain.mineBlock([
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "cancel-listing", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], maker.address),
    ]);
    const receipts = [types.uint(0), types.uint(999)].map((listingId) =>
      chain.callReadOnlyFn(
        contractName,
        "get-listing",
        [listingId],
        deployer.address,
      )
    );
    receipts.map((receipt) => receipt.result.expectNone());
  },
});
```

Fulfilling listings

And here are the ones we have been waiting for: the tests for order fulfilment. Since a seller can list an NFT for sale for either STX or SIP010 tokens, we write a separate test for both.

```
Clarinet.test({
  name: "Can fulfil an active listing with STX",
```

```
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], taker.address),
    ]);
    block.receipts[2].result.expectOk().expectUint(0);
    assertNftTransfer(
      block.receipts[2].events[0],
      nftAssetContract,
      tokenId,
      contractPrincipal(deployer),
      taker.address,
    );
    block.receipts[2].events.expectSTXTransferEvent(
      order.price,
      taker.address,
      maker.address,
    );
  },
});

Clarinet.test({
  name: "Can fulfil an active listing with SIP010 fungible tokens",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const price = 50;
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const { paymentAssetContract, paymentAssetId } = mintFt({
      chain,
      deployer,
      recipient: taker,
      amount: price,
    });
```

```
    const order: Order = { tokenId, expiry: 10, price, paymentAssetContract };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      whitelistAssetTx(paymentAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-ft", [
        types.uint(0),
        types.principal(nftAssetContract),
        types.principal(paymentAssetContract),
      ], taker.address),
    ]);
    block.receipts[3].result.expectOk().expectUint(0);
    assertNftTransfer(
      block.receipts[3].events[0],
      nftAssetContract,
      tokenId,
      contractPrincipal(deployer),
      taker.address,
    );
    block.receipts[3].events.expectFungibleTokenTransferEvent(
      price,
      taker.address,
      maker.address,
      paymentAssetId,
    );
  },
});
```

Basic fulfilment errors

There are some basic situations in which fulfilment fails, these are:

- The seller is trying to buy its own NFT,
- A buyer is trying to fulfil a listing that does not exist; and,
- A buyer is trying to fulfil a listing that has expired.

```
Clarinet.test({
  name: "Cannot fulfil own listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker] = ["deployer", "wallet_1"].map((name) =>
      accounts.get(name)!
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
```

```
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], maker.address),
    ]);
    block.receipts[2].result.expectErr().expectUint(2005);
    assertEquals(block.receipts[2].events.length, 0);
  },
});

Clarinet.test({
  name: "Cannot fulfil an unknown listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const { nftAssetContract } = mintNft({ chain, deployer, recipient: maker });
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], taker.address),
    ]);
    block.receipts[1].result.expectErr().expectUint(2000);
    assertEquals(block.receipts[1].events.length, 0);
  },
});

Clarinet.test({
  name: "Cannot fulfil an expired listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const expiry = 10;
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry, price: 10 };
    chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
    ]);
    chain.mineEmptyBlockUntil(expiry + 1);
    const block = chain.mineBlock([
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], taker.address),
```

```
    ]);
    block.receipts[0].result.expectErr().expectUint(2002);
    assertEquals(block.receipts[0].events.length, 0);
  },
});
```

Wrong payment asset or trait reference

We now add tests that confirm that a listing cannot be fulfilled with the wrong payment asset. We test both STX and SIP010 tokens, as well as the situation where the transaction sender passes in the wrong asset trait reference.

Since we have to test our listings against different assets we need to instantiate a bogus NFT and payment asset contract. We could copy and paste our test asset contracts but that seems tedious. Clarinet actually allows you to instantiate the same contract file under a different name. Open `Clarinet.toml` and add an entry for our `bogus-nft` by coping the entry for `sip009-nf`.

```
[contracts.sip009-nft]
path = "contracts/sip009-nft.clar"

[contracts.bogus-nft]
path = "contracts/sip009-nft.clar"
```

Then we do the same for the payment asset. We will call the new entry `bogus-ft`.

```
[contracts.sip010-token]
path = "contracts/sip010-token.clar"

[contracts.bogus-ft]
path = "contracts/sip010-token.clar"
```

From this point on, `sip009-nft.clar` and `sip010-token.clar` will be instantiated twice under different names. Quite useful for our tests.

```
Clarinet.test({
  name: "Cannot fulfil a listing with a different NFT contract reference",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const expiry = 10;
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: 10 };
    const bogusNftAssetContract = `${deployer.address}.bogus-nft`;
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
```

```
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(bogusNftAssetContract),
      ], taker.address),
    ]);
    block.receipts[2].result.expectErr().expectUint(2003);
    assertEquals(block.receipts[2].events.length, 0);
  },
});

Clarinet.test({
  name: "Cannot fulfil an active STX listing with SIP010 fungible tokens",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const price = 50;
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const { paymentAssetContract } = mintFt({
      chain,
      deployer,
      recipient: taker,
      amount: price,
    });
    const order: Order = { tokenId, expiry: 10, price };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      whitelistAssetTx(paymentAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-ft", [
        types.uint(0),
        types.principal(nftAssetContract),
        types.principal(paymentAssetContract),
      ], taker.address),
    ]);
    block.receipts[3].result.expectErr().expectUint(2004);
    assertEquals(block.receipts[3].events.length, 0);
  },
});

Clarinet.test({
  name: "Cannot fulfil an active SIP010 fungible token listing with STX",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const price = 50;
```

```
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const { paymentAssetContract } = mintFt({
      chain,
      deployer,
      recipient: taker,
      amount: price,
    });
    const order: Order = { tokenId, expiry: 10, price, paymentAssetContract };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      whitelistAssetTx(paymentAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], taker.address),
    ]);
    block.receipts[3].result.expectErr().expectUint(2004);
    assertEquals(block.receipts[3].events.length, 0);
  },
});

Clarinet.test({
  name:
    "Cannot fulfil an active SIP010 fungible token listing with a different SIP010
fungible token contract reference",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const price = 50;
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const { paymentAssetContract } = mintFt({
      chain,
      deployer,
      recipient: taker,
      amount: price,
    });
    const bogusPaymentAssetContract = `${deployer.address}.bogus-ft`;
    const order: Order = { tokenId, expiry: 10, price, paymentAssetContract };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      whitelistAssetTx(paymentAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
```

```
      Tx.contractCall(contractName, "fulfil-listing-ft", [
        types.uint(0),
        types.principal(nftAssetContract),
        types.principal(bogusPaymentAssetContract),
      ], taker.address),
    ]);
    block.receipts[3].result.expectErr().expectUint(2004);
    assertEquals(block.receipts[3].events.length, 0);
  },
});
```

Insufficient balance

It should naturally be impossible to purchase an NFT if the buyer does not have sufficient payment asset balance. There should be no token events in such cases.

```
Clarinet.test({
  name: "Cannot fulfil an active STX listing with insufficient balance",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = { tokenId, expiry: 10, price: taker.balance + 10 };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], taker.address),
    ]);
    block.receipts[2].result.expectErr().expectUint(1);
    assertEquals(block.receipts[2].events.length, 0);
  },
});

Clarinet.test({
  name:
    "Cannot fulfil an active SIP010 fungible token listing with insufficient
balance",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const price = 50;
    const { nftAssetContract, tokenId } = mintNft({
```

```
      chain,
      deployer,
      recipient: maker,
    });
    const { paymentAssetContract } = mintFt({
      chain,
      deployer,
      recipient: taker,
      amount: price,
    });
    const order: Order = {
      tokenId,
      expiry: 10,
      price: taker.balance + 10,
      paymentAssetContract,
    };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      whitelistAssetTx(paymentAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-ft", [
        types.uint(0),
        types.principal(nftAssetContract),
        types.principal(paymentAssetContract),
      ], taker.address),
    ]);
    block.receipts[3].result.expectErr().expectUint(1);
    assertEquals(block.receipts[3].events.length, 0);
  },
});
```

Intended taker

Sellers have the ability to list an NFT for sale that only one specific principal can purchase, by setting the "intended taker" field. Another two tests are in order: one where an intended taker is indeed able to fulfil a listing, and another where an unintended take is not able to fulfil the listing.

```
Clarinet.test({
  name: "Intended taker can fulfil active listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker] = ["deployer", "wallet_1", "wallet_2"].map(
      (name) => accounts.get(name)!,
    );
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = {
      tokenId,
      expiry: 10,
```

```
      price: 10,
      taker: taker.address,
    };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
        types.principal(nftAssetContract),
      ], taker.address),
    ]);
    block.receipts[2].result.expectOk().expectUint(0);
    assertNftTransfer(
      block.receipts[2].events[0],
      nftAssetContract,
      tokenId,
      contractPrincipal(deployer),
      taker.address,
    );
    block.receipts[2].events.expectSTXTransferEvent(
      order.price,
      taker.address,
      maker.address,
    );
  },
});

Clarinet.test({
  name: "Unintended taker cannot fulfil active listing",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const [deployer, maker, taker, unintendedTaker] = [
      "deployer",
      "wallet_1",
      "wallet_2",
      "wallet_3",
    ].map((name) => accounts.get(name)!);
    const { nftAssetContract, tokenId } = mintNft({
      chain,
      deployer,
      recipient: maker,
    });
    const order: Order = {
      tokenId,
      expiry: 10,
      price: 10,
      taker: taker.address,
    };
    const block = chain.mineBlock([
      whitelistAssetTx(nftAssetContract, true, deployer),
      listOrderTx(nftAssetContract, maker, order),
      Tx.contractCall(contractName, "fulfil-listing-stx", [
        types.uint(0),
```

```
        types.principal(nftAssetContract),
      ], unintendedTaker.address),
    ]);
    block.receipts[2].result.expectErr().expectUint(2006);
    assertEquals(block.receipts[2].events.length, 0);
  },
});
```

Multiple orders

We add one final bonus test that fulfils a few random listings in a random order. It can be useful to catch bugs that might not arise in a controller state with one listing and one purchase.

```
Clarinet.test({
  name: "Can fulfil multiple active listings in any order",
  async fn(chain: Chain, accounts: Map<string, Account>) {
    const deployer = accounts.get("deployer")!;
    const expiry = 100;

    const randomSorter = () => Math.random() - .5;

    // Take some makers and takers in random order.
    const makers = ["wallet_1", "wallet_2", "wallet_3", "wallet_4"].sort(
      randomSorter,
    ).map((name) => accounts.get(name)!);
    const takers = ["wallet_5", "wallet_6", "wallet_7", "wallet_8"].sort(
      randomSorter,
    ).map((name) => accounts.get(name)!);

    // Mint some NFTs so the IDs do not always start at zero.
    const mints = [...Array(1 + ~~(Math.random() * 10))].map(() =>
      mintNft({ chain, deployer, recipient: deployer })
    );

    // Mint an NFT for all makers and generate orders.
    const nfts = makers.map((recipient) =>
      mintNft({ chain, deployer, recipient })
    );
    const orders: Order[] = makers.map((maker, i) => ({
      tokenId: nfts[i].tokenId,
      expiry,
      price: 1 + ~~(Math.random() * 10),
    }));

    // Whitelist asset contract
    chain.mineBlock([
      whitelistAssetTx(mints[0].nftAssetContract, true, deployer),
    ]);

    // List all NFTs.
    const block = chain.mineBlock(
```

```
      makers.map((maker, i) =>
        listOrderTx(nfts[i].nftAssetContract, maker, makeOrder(orders[i]))
      ),
    );
    const orderIdUints = block.receipts.map((receipt) =>
      receipt.result.expectOk().toString()
    );

    // Attempt to fulfil all listings.
    const block2 = chain.mineBlock(
      takers.map((taker, i) =>
        Tx.contractCall(contractName, "fulfil-listing-stx", [
          orderIdUints[i],
          types.principal(nfts[i].nftAssetContract),
        ], taker.address)
      ),
    );

    const contractAddress = contractPrincipal(deployer);

    // Assert that all orders were fulfilled and that the NFTs and STX have been
tranferred to the appropriate principals.
    block2.receipts.map((receipt, i) => {
      assertEquals(receipt.result.expectOk(), orderIdUints[i]);
      assertNftTransfer(
        receipt.events[0],
        nfts[i].nftAssetContract,
        nfts[i].tokenId,
        contractAddress,
        takers[i].address,
      );
      receipt.events.expectSTXTransferEvent(
        orders[i].price,
        takers[i].address,
        makers[i].address,
      );
    });
  },
});
```

That was quite a lot, but there we are!

Runtime cost analysis

Every transaction executed on-chain comes at a cost. Miners run and verify transactions, levying *transaction fees* for their work. These costs should be very familiar to anyone who has ever interacted with a blockchain.

But there are other costs that not many are aware of. They are called *execution costs* and each block is limited by them. In addition to block size, these are what limit the number of transactions that can fit into a single block. They are also used to reduce resource consumption when calling read-only functions.

Execution costs

In **Clarity**, execution costs are broken up into five different categories, each with its own limit.

```
+----------------------+--------------+-----------------+
|                      | Block Limit  | Read-Only Limit |
+----------------------+--------------+-----------------+
| Runtime              | 5000000000   | 1000000000      |
+----------------------+--------------+-----------------+
| Read count           | 15000         | 30             |
+----------------------+--------------+-----------------+
| Read length (bytes)  | 100000000    | 100000          |
+----------------------+--------------+-----------------+
| Write count          | 15000         | 0              |
+----------------------+--------------+-----------------+
| Write length (bytes) | 15000000     | 0               |
+----------------------+--------------+-----------------+
```

Runtime costs limits overall complexity of the code that can be executed. For example, negating a boolean value is less complex than calculating SHA512 hash, therefore `(not false)` will consume less runtime costs than `(sha512 "hello world")` . This category is also affected by contract size.

Read count limits how many times we can read from memory or chain state to a extract piece of information. It is affected by reading constants, variables, intermediate variables created with `let` , maps, but also by some functions that needs to save intermediate results during execution.

Read length (bytes) limits how much data we can read from memory or the chain. It is also affected by contract size. Calling into a contract using `contract-call?` Increases the read length by an amount equal to the contract size in bytes, *every time*. If you call into a contract with a length of 2, 000 bytes twice, the read length is increased twice.

Write count limits how many times we can write data into chain. It increments when writing to variables and maps.

Write length (bytes) limits how much data we can write to the chain.

Read and write limits are self explanatory and practically static. The more data we read and write, the closer we get to the limits. Runtime costs are more dynamic, and more difficult to grasp as each Clarity function has its own runtime cost. For some it is static, and for the others it changes based on how much data they have to process. For example, the `not` function always takes exactly one boolean and does the same amount of work, while `filter` takes an iterator function and a list that can contain a number of elements, making the amount of work required dynamic.

Developers have to be careful with how they structure their code, which functions they use, and they use them, as it is quite easy to write code that is entirely correct, yet so expensive to execute that it will eat significant portion of execution costs set for all transactions in a block. As a result, the function may only be able to be called a few times per block—not to mention having to compete with others for such a large chunk of the block.

Analysis using Clarinet

Analysis costs can be challenging, so it is important to have good tooling. Clarinet has cost analysis features built-in which make the development lifecycle a lot easier. We will take a look at manual as well as automated cost analysis tests.

Manual analysis

You can get the costs of any expression manually by dropping into a console session. Open the counter project from chapter 7.2 and start a session with `clarinet console` . Remember that you can type `::help` to see REPL commands. In there, you will find a command to analyse the costs of an expression:

```
::get_costs <expr>                     Display the cost analysis
```

You can put the command in front of any Clarity expression to tell the REPL that you also want to see the execution costs. Let us try it with the `not` function to negate a `false` value:

```
>> ::get_costs (not false)
+----------------------+----------+------------+
|                      | Consumed | Limit      |
+----------------------+----------+------------+
| Runtime              | 186      | 5000000000 |
+----------------------+----------+------------+
| Read count           | 0        | 15000      |
+----------------------+----------+------------+
| Read length (bytes)  | 0        | 100000000  |
+----------------------+----------+------------+
| Write count          | 0        | 15000      |
+----------------------+----------+------------+
| Write length (bytes) | 0        | 15000000   |
+----------------------+----------+------------+

true
```

As you can see, the `not` function consumed 186 runtime units. All the others are zero because the not function does not read or write anything.

Next, let us try calling the `sha512` function with an input uint of `u1234` .

```
>> ::get_costs (sha512 u1234)
+----------------------+----------+------------+
|                      | Consumed | Limit      |
+----------------------+----------+------------+
| Runtime              | 193      | 5000000000 |
```

```
+----------------------+----------+------------+
| Read count           | 0        | 15000      |
+----------------------+----------+------------+
| Read length (bytes)  | 0        | 100000000  |
+----------------------+----------+------------+
| Write count          | 0        | 15000      |
+----------------------+----------+------------+
| Write length (bytes) | 0        | 15000000   |
+----------------------+----------+------------+

0x523be47185d0ffe54cb649d4e6303db92f54d2949becd8b6c0c91830006523b155fd9eaad1095c0f20801
```

It thus appears that the `sha512` function is more expensive to execute than `not`, which makes sense.

But we are not limited to just inline expressions. We can also get the runtime cost of a contract call. Remember how to call `count-up`? Let us see what costs are involved with updating the `counters` map.

```
>> ::get_costs (contract-call? .counter count-up)
+----------------------+----------+------------+
|                      | Consumed | Limit      |
+----------------------+----------+------------+
| Runtime              | 6692     | 5000000000 |
+----------------------+----------+------------+
| Read count           | 5        | 15000      |
+----------------------+----------+------------+
| Read length (bytes)  | 418      | 100000000  |
+----------------------+----------+------------+
| Write count          | 1        | 15000      |
+----------------------+----------+------------+
| Write length (bytes) | 165      | 15000000   |
+----------------------+----------+------------+

(ok true)
```

Quite the difference from the above. Explaining all the exact numbers is a bit tough but we can make a few deductions:

- Loading the contract into memory affects the the runtime and read dimensions.
- Reading the map via `map-get?`, as found in the `get-count` function, adds to the read dimensions.
- The `count-up` function calls `map-set` and writes a new value, which is evident from the write count of one.

Automated analysis

Clarinet can automatically run cost analysis on test. As long as your unit tests are comprehensive, then you do not need to do anything special to make use of this feature. The unit tests we wrote in chapter 7.4 can be executed while also analysing costs by adding the `--costs` option. The full command is thus as follows: `clarinet test --costs`. You will see that a costs analysis table will be printed after the result of the unit tests.

```
Running counter/tests/counter_test.ts
* get-count returns u0 for principals that never called count-up before ... ok (5ms)
* count-up counts up for the tx-sender ... ok (6ms)
* counters are specific to the tx-sender ... ok (13ms)

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out (382ms)

Contract calls cost synthesis
+-----------------------------------+-----------------+------------+----------------
-----+-------------+----------------------+--------------+
|                                   | Runtime (units) | Read Count | Read Length
(bytes) | Write Count | Write Length (bytes) | Tx per Block |
+-----------------------------------+-----------------+------------+----------------
-----+-------------+----------------------+--------------+
| counter::count-up                 |    6692 (0.00%) |  5 (0.03%) |         418
(0.00%) |   1 (0.01%) |          165 (0.00%) |         1550 |
+-----------------------------------+-----------------+------------+----------------
-----+-------------+----------------------+--------------+
| counter::get-count                |    3213 (0.00%) |  4 (0.03%) |         418
(0.00%) |           0 |                    0 |         1937 |
+-----------------------------------+-----------------+------------+----------------
-----+-------------+----------------------+--------------+
|
|
+-----------------------------------+-----------------+------------+----------------
-----+-------------+----------------------+--------------+
| Mainnet Block Limits (Stacks 2.0) |      5000000000 |      15000 |
100000000 |       15000 |             15000000 |            / |
+-----------------------------------+-----------------+------------+----------------
-----+-------------+----------------------+--------------+
```

What makes the test method even more useful is that it also shows you the percentage of the block budget each function consumes. That number is very important as it shows you how many of these contract calls can fit in a single block. And remember, every user of the blockchain competes for the same block budget! If your function calls takes a good chunk of the block budget then miners might elect to ignore your transaction so that it can fit a larger number of transactions in the block. You should therefore strive to make the percentages as low as possible.

Optimising runtime cost

Setting up a cost baseline

As was seen in the previous section, costs are influenced by various factors. It can therefore be hard to trace exactly which parts of your smart contracts are the largest contributors to the overall runtime cost. It can be a good idea to add a cost baseline by temporarily including a simple function to the contract you want to analyse. The cost dimensions of calling the function will then serve as the minimum amount required to interact with your contract. Such a function could look like this:

```
(define-read-only (a) true)
```

Clarity is an interpreted language, which means that the length of symbols also has an effect on the runtime cost. That means that the length of function, argument, and variable names must be taken into account. The baseline function above has the shortest possible name `a` and returns the simplest type of Clarity value; namely, a boolean.

Having a reference point is very important, especially when working with big contracts, because in big contracts functions that should be cheap can be quite expensive only due to contract size. You might thus erroneously assume there is something wrong with your function when it is actually due to the sheer size of the contract itself.

Contracts that require a lot of upfront setup

More complicated smart contract systems might require a decent amount of upfront work. Examples include adding initial values to data maps or by calling into different contracts. These actions normally only need to happen once. If your application demands such a setup then consider moving the setup logic into a separate contract. The setup contract can then be deployed by itself and used to initialise your project. Once the setup is completed, the application will run more efficiently as the core contract is not bogged down by many lines of initialising code. Not to mention that decoupling your setup logic can make maintenance easier as well.

Common optimisations

Remove code repetition

A general rule is to prevent code repetition. If you catch yourself copying and pasting parts of a function then consider turning that logic into a separate function.

Inline expressions

Any time a variable is defined using `let`, check if the variable is actually used more than once. If not, inline the variable expression where it is used. It might even be the case that the `let` expression itself can be replaced with a cheaper one like `begin`.

Remove fake variables

You should only define contract variables with `define-data-var` if you actually plan on changing the variables over the contract lifetime. Any variables that are defined once and then only read can usually be replaced by constants defined with `define-constant`.

Call into contracts as few times as possible

A pattern commonly observed is to have one contract that contains business logic and another that is meant for storage. If a function you are writing calls into a contract multiple times and you control the destination contract, then consider rewriting both contracts such that only a single call is necessary. A contrived example to highlight the change is as follows:

```
(define-public (example-1)
    (let
        (
            ;; Here the storage contract is called twice, which means
            ;; the contract code is loaded twice, which in turn means
            ;; the read dimension is incremented twice.
            (value-a (contract-call? .storage-contract get-value-a))
            (value-b (contract-call? .storage-contract get-value-b))
```

```
        )
        ;; Business logic here...
        (ok true)
    )
)

(define-public (example-2)
    (let
        (
            ;; Here the storage is called only once. Cost savings!
            (value-a-b (contract-call? .storage-contract get-value-a-and-b))
            ;; The value returned is a tuple containing both values.
            (value-a (get a value-a-b))
            (value-b (get b value-a-b))
        )
        ;; Business logic here...
        (ok true)
    )
)
```

Further optimisation techniques

Reducing runtime cost can be as challenging as writing the smart contract in the first place. There is no silver bullet for when a contract turns out to be rather expensive to execute. Next to the more common optimisations, here are some more techniques and tips to consider:

1. Reduce obvious code complexity. First write correct code, then work on the costs.
2. Reduce the amount of data to be read and written. Analyse what information actually needs to be stored and retrieved.
3. Reduce number of times the contract reaches for the same data. Do not read the same variables or map data multiple times.
4. Look at how data stored in data variables and maps is used in the contract. Split or combine them where it makes sense. If a data variable or map stores a tuple, see if you always need all fields.
5. Combine multiple contract calls to the same contract into single call if possible.
6. Inline or extract logic if you notice repetition.
7. Reducing the amount of data passed between functions. For example, if you only need one field of a tuple, pass just that value instead of the entire tuple.
8. See if reversing logic is cheaper. For example, instead of counting positive values in a list, it might be more cost effective to count zeros if it is expected that the list should consists of only positive values.
9. Unroll loops made with `map` and `fold` and see what difference it makes.
10. Reduce contract size by removing comments, using shorter names for functions, variables, and keys in tuples.

Long term recommendations

Start with working code and focus on functionality first. Once that is done, write tests and write a lot of them. Code refactoring is difficult and tests help you make sure that your code behaves the same way before and after the refactor. Bugs become exceedingly difficult to catch the more complicated the project becomes. A complex function may break accidentally when it is refactored. Furthermore, aim for low hanging fruits first by applying common optimisation techniques. Avoid optimising complex

functions as the smaller ones may give you what you need. Small gains can accumulate very quickly and cause a snowball effect. Reducing costs of one function by mere fraction may result in a large cost reduction in another.

Code that is efficient in byte size—that is to say, takes up the least amount of code—may often not be the most efficient in terms of runtime cost. For example, if you execute a `fold` over a small list, but each element of that list is a tuple, test if unwinding the code is cheaper to execute. You will find it to be the case rather often. Unwinding a loop means to get rid of the iteration in favour of sequential statements. The contract becomes larger in terms of actual code but may be reduced in complexity. If you are unfamiliar with unwinding, here is a very basic example of the process:

```
(define-private (sum-values-iter (current uint) (previous uint))
    (+ previous current)
)

(define-read-only (sum-values (values (list 10 uint)))
    (fold sum-values-iter values u0)
)

;; Unwinding removes the iterative function `fold` .
(define-read-only (sum-values-unwind (values (list 10 uint)))
    (+
        (default-to u0 (element-at? values u0))
        (default-to u0 (element-at? values u1))
        (default-to u0 (element-at? values u2))
        (default-to u0 (element-at? values u3))
        (default-to u0 (element-at? values u4))
        (default-to u0 (element-at? values u5))
        (default-to u0 (element-at? values u6))
        (default-to u0 (element-at? values u7))
        (default-to u0 (element-at? values u8))
        (default-to u0 (element-at? values u9))
    )
)
```

Always remember that you should do as little on-chain as possible. You may find that a read-only function does not fit into read-only limits. If the output of the function is meant to be displayed in the frontend of your application, then consider exposing the required contract data used by the function directly and handle the actual calculations off-chain.

Best practices

Clarity is limited by design but still very flexibility in execution. There are many different ways to implement something that is functionally equivalent. We all know though, that functional equivalence does not mean runtime cost equivalence. No programming language book would be complete without discussing some best practices. All programming languages have their own patterns, common traps, and other caveats developers ought to consider.

Coding style

Here are some suggestions on Clarity coding style. The chapter is intentionally not called *coding guidelines:* the community leads the way. If coding style SIPs are ever ratified then they shall be covered here. Until then we can only make a best effort based on the experiences of long-time developers.

Pointless begins

Most functions take an invariant amount of inputs. Some developers therefore develop a tendency to overuse `begin`. If your `begin` only contains one expression, then you can remove it!

```
(define-public (get-listing (id uint))
    (begin
        (ok (map-get? listings {id: id}))
    )
)
```

And without:

```
(define-public (get-listing (id uint))
    (ok (map-get? listings {id: id}))
)
```

There actually is a runtime cost attached to `begin`, so leaving it out makes your contract call cheaper. The same goes for `begin` inside other invariant function expressions.

```
>> ::get_costs (+ 1 2)
+----------------------+----------+------------+
|                      | Consumed | Limit      |
+----------------------+----------+------------+
| Runtime              | 4000     | 5000000000 |
+----------------------+----------+------------+
3

>> ::get_costs (begin (+ 1 2))
+----------------------+----------+------------+
|                      | Consumed | Limit      |
+----------------------+----------+------------+
| Runtime              | 6000     | 5000000000 |
+----------------------+----------+------------+
3
```

Nested lets

The `let` function allows us to define local variables. It is useful if you would otherwise have to read data or redo a calculation multiple times. Variable expressions are actually evaluated in sequence which means that a later variable expression can refer to prior expressions. There is therefore no need to nest multiple `let` expressions if all you want to do is calculate a value based on some prior variables.

```
(let
    (
    (value-a u10)
    (value-b u20)
    (result (* value-a value-b))
    )
    (ok result)
)
```

Avoid *-panic functions

There are multiple ways to unwrap values, but `unwrap-panic` and `unwrap-err-panic` should generally be avoided. They abort the call with a runtime error if they fail to unwrap the supplied value. A runtime error does not give any meaningful information to the application calling the contract and makes error handling more difficult. Whenever possible, use `unwrap!` and `unwrap-err!` with a meaningful error code.

Compare the functions `update-name` and `update-name-panic` in the example below.

```
(define-public (update-name (id uint) (new-name (string-ascii 50)))
    (let
        (
            ;; Emits an error value when the unwrap fails.
            (listing (unwrap! (get-listing id) err-unknown-listing))
        )
        (asserts! (is-eq tx-sender (get maker listing)) err-not-the-maker)
        (map-set listings {id: id} (merge listing {name: new-name}))
        (ok true)
    )
)

(define-public (update-name-panic (id uint) (new-name (string-ascii 50)))
    (let
        (
            ;; No meaningful error code is emitted if the unwrap fails.
            (listing (unwrap-panic (get-listing id)))
        )
        (asserts! (is-eq tx-sender (get maker listing)) err-not-the-maker)
        (map-set listings {id: id} (merge listing {name: new-name}))
        (ok true)
    )
)
```

It is best to restrict the use of `unwrap-panic` and `unwrap-err-panic` to instances where you already know beforehand that the unwrapping should not fail. (Because of a prior guard, for example.)

Avoid the if function

To be honest, you should not actually avoid using the `if` function. But anytime you use it, ask yourself if you do really need it. Quite often, you can refactor the code and replace it with `asserts!` or `try!`. New developers often end up creating nested `if` structures because they need to check

multiple conditions in sequence. Those structures become extremely hard to follow and are prone to error.

For example:

```
(define-public (update-name (new-name (string-ascii 50)))
    (if (is-eq tx-sender contract-owner)
        (ok (var-set contract-name new-name))
        err-not-contract-owner
    )
)
```

Can be rewritten to:

```
(define-public (update-name (new-name (string-ascii 50)))
    (begin
        (asserts! (is-eq tx-sender contract-owner) err-not-contract-owner)
        (ok (var-set contract-name new-name))
    )
)
```

Multiply nested `if` expressions can usually be rewritten this way. Just compare this:

```
(define-public (some-function)
    (if bool-expr-A
        (if bool-expr-B
            (if bool-expr-C
                (ok (process-something))
                if-C-false
            )
            if-B-false
        )
        if-A-false
    )
)
```

To this:

```
(define-public (some-function)
    (begin
        (asserts! bool-expr-A if-A-false)
        (asserts! bool-expr-B if-B-false)
        (asserts! bool-expr-C if-C-false)
        (ok (process-something))
    )
)
```

To match or not to match

`match` is a really powerful function, but a `try!` is sufficient in many cases. A commonly observed pattern is as follows:

```
(match (some-expression)
    success (ok success)
    error (err error)
)
```

For which the functionally equivalent simplification is nothing more than the function call itself:

```
(some-expression)
```

`match` unwraps the result of a `response` and enters either the success or failure branch with the unwrapped `ok` or `err` value. Immediately returning those values is therefore pointless.

Here is a real transfer function found in a mainnet contract:

```
(define-public (transfer (token-id uint) (sender principal) (recipient principal))
    (if (and (is-eq tx-sender sender))
        (match (nft-transfer? my-nft token-id sender recipient)
            success (ok success)
            error (err error))
        (err u500)
    )
)
```

Refactoring the `if` and `match`, we are left with just this:

```
(define-public (transfer (token-id uint) (sender principal) (recipient principal))
    (begin
        (asserts! (is-eq tx-sender sender) (err u500))
        (nft-transfer? my-nft token-id sender recipient)
    )
)
```

What to store on-chain

Smart contracts are a special breed of resource-constraint systems. Developers that just enter the blockchain space usually bring with them certain assumptions that may not be valid when writing smart contracts. It is paramount to remember that a blockchain is distributed data storage *over time*. Every state change is *paid for* in terms of the miner fee.

Data storage

The user pays for every byte of storage that is written to the chain, which means that you naturally want to keep the amount of data therefore to a minimum. However, before you try to optimise the *amount* of data, you should actually think about the *type* of data you want to store. Ask yourself if the data should exist on-chain in the first place. It is a common trap to think that in order for all data of your app to be immutable and trustless, it has to be stored in a smart contract.

Imagine you are making a job board application that does job matching on-chain. A job posting has a title, description, information on who posted it and so on. Should you really start defining complicated data maps like this?

```
(define-map jobs
    uint
    {
        poster: principal,
        title: (string-utf8 200),
        description: (string-utf8 10000),
        posting-date: (string-utf8 12),
        expiry-date: (string-utf8 12),
        vacancies: uint,
        salary-range-lower: uint,
        salary-range-upper: uint
        ;; And so on...
    }
)
```

Definitely not! If the `(string-utf8 10000)` does not already tick you off, then consider the following: what if your app has been running for a couple of months and there is a need to introduce some new fields? You cannot simply update the contract and migrating the data already present in the map to a new contract is going to be a serious operation.

What should you do instead? If you care about the integrity of your job posting, then submitting a *hash of the data* to the contract is sufficient. The job data itself can live in off-chain storage and your app will compare a hash of the data with the hash that is stored on-chain. If they differ then there was an unauthorised change. The principal that posted the job can still update the job information by submitting the updated hash to the contract.

Here is what that part of the contract could look like:

```
(define-map jobs
    uint
    {
        poster: principal,
        data-hash: (buff 32)
    }
```

```
)

(define-read-only (get-job-hash (job-id uint))
    (get data-hash (map-get? jobs job-id))
)

(define-public (update-job-posting (job-id uint) (new-data-hash (buff 32)))
    (begin
        (asserts! (is-eq (get poster (map-get? jobs job-id)) (some tx-sender)) (err
u100)) ;; not the poster
        (ok (map-set jobs job-id {poster: tx-sender, data-hash: new-data-hash}))
    )
)
```

And some pseudo-code of the client-side:

```
function validate_job(job_id, data) {
  let hash = sha256(data);
  return hash === contract_read({
    contract_address,
    function_name: "get-job-hash",
    function_args: [uintCV(job_id)],
  });
}

function update_job(job_id, data) {
  let hash = sha256(data);
  return broadcast_contract_call({
    contract_address,
    function_name: "update-job-posting",
    function_args: [uintCV(job_id), buffCV(hash)],
  });
}
```

Historical data

The blockchain by virtue of its fundamental principles already stores the full history. Smart contracts therefore do not usually have to track a history of something themselves. If you are building an on-chain auction, for example, you may want to track the history of the highest bids. Maybe you store the highest bid in a variable and you add the previous highest bids in a list (bad) or a map (less bad). No need! The built-in function `at-block` allows you to go back in time. It changes the context to what the chain state was at the specified block, allowing you to see what the highest bid variable contained.

```
(define-constant err-bid-too-low (err u100))
(define-constant err-invalid-block (err u101))

(define-data-var highest-bid
    {bidder: principal, amount: uint}
    {bidder: tx-sender, amount: u0}
)
```

```
(define-public (bid (amount uint))
    (begin
        (asserts! (> amount (get amount (var-get highest-bid))) err-bid-too-low)
        (ok (var-set highest-bid {bidder: tx-sender, amount: amount}))
    )
)

(define-read-only (get-highest-bid)
    (var-get highest-bid)
)

(define-read-only (get-highest-bid-at (historical-block-height uint))
    (at-block
        (unwrap! (get-block-info? id-header-hash historical-block-height) err-
invalid-block)
        (ok (get-highest-bid))
    )
)
```

Contract upgradability

Once a smart contract is deployed to a blockchain it can no longer be changed. It is yet another property that puts smart contract development at odds with conventional application development: there is no way to *update* contract code in the future. One cannot simply issue a fix if a bug is later discovered. But like any application, usually updates are necessary to move it forward. In this chapter we will explore some ways in which developers can build decentralised applications that are still upgradable in the future.

There is no catch-all approach to creating a maximally flexible smart contract solution. However, keeping the following principles in mind will go a long way:

- Keep logic separate, do not make one monolithic contract.
- Make contracts stateless whenever possible.
- Do not hardcode principals unless you are absolutely certain you will not have to replace them.
- Do not rely on the contract deployer for future upgrades.

Monolithic versus modular designs

There are certain benefits to fitting all business logic in a single smart contract. However, it can be very detrimental when it comes to upgradability. To illustrate the problem, imagine a DAO-like smart contract that allows members to submit proposals and vote on them. All members can vote but only those that are whitelisted can submit proposals. A vote can only be cast once and cannot be changed.

Such a contract might look like the example given below. Logic to manage members and whitelisted members has been left out for brevity.

```
(define-constant err-not-whitelisted (err u100))
(define-constant err-unknown-proposal (err u101))
(define-constant err-not-member (err u102))
(define-constant err-already-voted (err u103))
(define-constant err-voting-ended (err u104))

(define-constant proposal-duration u1440)

(define-data-var proposal-nonce uint u0)
(define-map proposals uint
    {
    proposer: principal,
    title: (string-ascii 100),
    end-height: uint,
    yes-votes: uint,
    no-votes: uint
    }
)
(define-map proposal-votes {voter: principal, proposal-id: uint} {vote-height: uint,
for: bool})

(define-map members principal bool)
(define-map whitelisted-members principal bool)

(define-read-only (get-proposal (proposal-id uint))
    (ok (map-get? proposals proposal-id))
```

```
)

(define-public (submit-proposal (title (string-ascii 100)))
    (let
        (
            (proposal-id (+ (var-get proposal-nonce) u1))
        )
        (asserts! (default-to false (map-get? whitelisted-members tx-sender)) err-
not-whitelisted)
        (map-set proposals proposal-id
            {
            proposer: tx-sender,
            title: title,
            end-height: (+ block-height proposal-duration),
            yes-votes: u0,
            no-votes: u0
            }
        )
        (var-set proposal-nonce proposal-id)
        (ok proposal-id)
    )
)

(define-read-only (get-vote (member principal) (proposal-id uint))
    (map-get? proposal-votes {voter: member, proposal-id: proposal-id})
)

(define-public (vote (for bool) (proposal-id uint))
    (let
        (
            (proposal (unwrap! (map-get? proposals proposal-id) err-unknown-
proposal))
        )
        (asserts! (default-to false (map-get? members tx-sender)) err-not-member)
        (asserts! (< block-height (get end-height proposal)) err-voting-ended)
        (asserts! (is-none (get-vote tx-sender proposal-id)) err-already-voted)
        (map-set proposal-votes {voter: tx-sender, proposal-id: proposal-id} {vote-
height: block-height, for: for})
        (if for
            (map-set proposals proposal-id (merge proposal {yes-votes: (+ (get yes-
votes proposal) u1)}))
            (map-set proposals proposal-id (merge proposal {no-votes: (+ (get no-
votes proposal) u1)}))
        )
        (ok true)
    )
)
```

The code is rather succinct. But which immediate problems can you spot with this implementation? The proposal and voting logic are closely coupled. Furthermore, the state—that is to say, the stored data—is contained within the same contract. The developers will have an extremely hard time if they want to change the rules that govern proposal submission. We can reimagine the project as two

separate contracts, one that stores the proposals and votes, and another contract that is allowed to manipulate the data.

Data storage contract

The contract that stores the data will only allow one privileged principal to change its internal state. The principal itself can also be updated the same way.

```
(define-constant err-not-owner (err u100))
(define-constant err-unknown-proposal (err u101))
(define-constant err-voting-ended (err u102))

(define-data-var contract-owner principal tx-sender)
(define-data-var proposal-nonce uint u0)
(define-map proposals uint
    {
    proposer: principal,
    title: (string-ascii 100),
    end-height: uint,
    yes-votes: uint,
    no-votes: uint
    }
)
(define-map proposal-votes {voter: principal, proposal-id: uint} {vote-height: uint,
for: bool})

(define-read-only (is-contract-owner)
    (ok (asserts! (is-eq contract-caller (var-get contract-owner)) err-not-owner))
)

(define-read-only (get-contract-owner)
    (ok (var-get contract-owner))
)

(define-public (set-contract-owner (new-owner principal))
    (begin
        (try! (is-contract-owner))
        (ok (var-set contract-owner new-owner))
    )
)

(define-read-only (get-proposal (proposal-id uint))
    (map-get? proposals proposal-id)
)

(define-public (insert-proposal (title (string-ascii 100)) (end-height uint)
(proposer principal))
    (let
        (
            (proposal-id (+ (var-get proposal-nonce) u1))
        )
        (try! (is-contract-owner))
```

```
        (map-set proposals proposal-id
            {
            proposer: proposer,
            title: title,
            end-height: end-height,
            yes-votes: u0,
            no-votes: u0
            }
        )
        (var-set proposal-nonce proposal-id)
        (ok proposal-id)
    )
)

(define-read-only (get-vote (member principal) (proposal-id uint))
    (ok (map-get? proposal-votes {voter: member, proposal-id: proposal-id}))
)

(define-public (set-vote (for bool) (voter principal) (proposal-id uint))
    (let
        (
            (proposal (unwrap! (get-proposal proposal-id) err-unknown-proposal))
            (previous-vote (get-vote voter proposal-id))
        )
        (try! (is-contract-owner))
        (asserts! (< block-height (get end-height proposal)) err-voting-ended)
        (map-set proposal-votes {voter: voter, proposal-id: proposal-id} {vote-
height: block-height, for: for})

        ;; If the new vote is the same as the previous vote, return (ok true) early.
        (asserts! (not (is-eq (some for) (get for previous-vote))) (ok true))

        ;; Update vote count. If there was a previous vote, then subtract it.
        ;; If the code enters into this expression, then the previous vote must
        ;; be opposite of the new vote. (Because of the prior assertion.)
        (if for
            (map-set proposals proposal-id (merge proposal
                {
                yes-votes: (+ (get yes-votes proposal) u1),
                no-votes: (- (get no-votes proposal) (if (is-some previous-vote) u1
u0))
                })
            )
            (map-set proposals proposal-id (merge proposal
                {
                yes-votes: (- (get yes-votes proposal) (if (is-some previous-vote)
u1 u0)),
                no-votes: (+ (get no-votes proposal) u1)
                })
            )
        )
        (ok true)
```

```
    )
)
```

Privileged contract

The privileged contract is meant to become the `contract-owner` of the storage contract above. It will then be the contract that the members interact with. It checks if the required conditions to submit a proposal or cast a vote are met and will call into the storage contract to update the state. To keep the code short, member handling logic has again been omitted. Additionally, in a real scenario the privileged contract would need some kind of mechanism to change the `contract-owner` of the storage contract. How this would work will depend on the project. Perhaps the members have to signal the switch, or it could even be managed by a DAO.

Assuming such `contract-owner` update logic exists, one can see how a modular approach to smart contract design greatly improves flexibility. Were the developers of this hypothetical project to introduce NFTs in the future, they could deploy a new privileged contract that requires the `tx-sender` to own an NFT in order to be able to submit a proposal.

```
(define-constant err-not-whitelisted (err u200))
(define-constant err-not-member (err u201))

(define-constant proposal-duration u1440)

(define-map members principal bool)
(define-map whitelisted-members principal bool)

(define-public (submit-proposal (title (string-ascii 100)))
    (begin
        (asserts! (default-to false (map-get? whitelisted-members tx-sender)) err-
not-whitelisted)
        (contract-call? .proposal-storage insert-proposal title (+ block-height
proposal-duration) tx-sender)
    )
)

(define-public (vote (for bool) (proposal-id uint))
    (begin
        (asserts! (default-to false (map-get? members tx-sender)) err-not-member)
        (contract-call? .proposal-storage set-vote for tx-sender proposal-id)
    )
)

;; tx-sender is the contract deployer on the top level, so we can immediately
;; make the privileged contract the contract-owner of the storage contract.
(contract-call? .proposal-storage set-contract-owner (as-contract tx-sender))
```

Statelessness

Making a project modular is important but one can detect issues even with the previous example; namely, the privileged contract still contains state concerning user membership. Migrating state from one contract to another is tedious and perhaps virtually impossible depending on the amount of data.

Statelessness is achieved by extracting just the business logic and turning that into a separate contract. The `members` and `whitelisted-members` data maps can also be moved to a new contract—or perhaps even the existing storage contract—which is then queried by the privileged contract.

```
(define-constant err-not-owner (err u100))
(define-constant err-not-member (err u102))

(define-data-var contract-owner principal tx-sender)
(define-map members principal bool)
(define-map whitelisted-members principal bool)

(define-read-only (is-contract-owner)
    (ok (asserts! (is-eq contract-caller (var-get contract-owner)) err-not-owner))
)

(define-read-only (get-contract-owner)
    (var-get contract-owner)
)

(define-read-only (is-member (who principal))
    (asserts! (default-to false (map-get? members who)) err-not-member)
)

(define-read-only (is-whitelisted (who principal))
    (default-to false (map-get? whitelisted-members who))
)

(define-public (set-member (is-member bool) (who principal))
    (begin
        (try! (is-contract-owner))
        (ok (map-set members who is-member))
    )
)

(define-public (set-whitelisted (is-whitelisted bool) (who principal))
    (begin
        (try! (is-contract-owner))
        (ok (map-set whitelisted-members who is-whitelisted))
    )
)
```

The assertions in the privileged contracts are then to be replaced by contract calls to the member storage contract:

```
(try! (contract-call? .member-storage is-member tx-sender))
```

Hardcoded principals

Hardcoding principals in your contracts is another way in which two contracts become more tightly coupled. We saw such tight coupling in the privileged contract of the previous example. It contains a hardcoded link to the proposal storage contract and later the member storage contract. Since the

privileged contract is now completely stateless, it will be more easily to replace with a new contract in the future. Still, let us imagine that we want to remove the hardcoded principals so that we can swap them out in the future. In chapter 9 we learned about dynamic dispatch. The solution is therefore to move the contract principals themselves to data variables and use trait references to call into the appropriate storage contract.

The final vote function and surrounding code would look something like this.

```
(define-data-var member-storage-principal .member-storage)
(define-data-var proposal-storage-principal .proposal-storage)
(define-constant err-invalid-member-storage-reference (err u200))
(define-constant err-invalid-proposal-storage-reference (err u201))

(define-trait member-storage-trait
    (
        (is-member (principal) (response bool uint))
        ;; And the rest of the functions...
    )
)

(define-trait proposal-storage-trait
    (
        (set-vote (bool principal uint) (response bool uint))
        ;; And the rest of the functions...
    )
)

(define-public (vote (for bool) (proposal-id uint) (member-storage-ref <member-
storage-trait>) (proposal-storage-ref <proposal-storage-trait>))
    (begin
        (asserts! (is-eq (contract-of member-storage-ref) (var-get member-storage-
principal)) err-invalid-member-storage-reference)
        (asserts! (is-eq (contract-of proposal-storage-ref) (var-get proposal-
storage-principal)) err-invalid-proposal-storage-reference)
        (try! (contract-call? member-storage-ref is-member tx-sender))
        (contract-call? proposal-storage-ref set-vote for tx-sender proposal-id)
    )
)

;; And the rest of the contract...
```

It does introduce some state back into the contract, but those variables are exclusively used internally.

Contract deployer reliance

A pattern often seen in Clarity smart contracts is to store the contract deployer in a variable, like so:

```
(define-constant contract-deployer tx-sender)
```

The constant is then later used for authorisation purposes. Such a set up is rather brittle if it is a single-signature or *n-of-n* multi-signature principal. The original deployer or deployers could potentially

lose the private key in the future rendering the authorised functions inaccessible forever. If the upgrade mechanism requires can only be invoked by the contract deployer, then it puts the project in a tough situation. While the most effective solution depends on the project, one should always consider the ramifications of key loss. A straightforward stategy is to implement a multi-principal ownership model as seen in the multi-signature vault practice project or to make grant authorised access to a contract with DAO-like capabilities.

Links and resources

- Stacks Documentation
- Hiro Developer Documentation

Foreword

Information wants to be free. But, most people believe in the concept of ownership. Despite the seeming contradiction, a blockchain synthesizes a mechanical social contract from these two premises. Everyone gets a copy of the chain, and everyone collectively ensures that only the rightful owner can create, modify, or transfer data within it. Smart contracts are programs on the blockchain that encode the rules for owning data, which are collectively and deterministically enforced by the blockchain users’ computers.

Naturally, writing smart contracts is a high-stakes business. Bugs can lose or destroy other peoples' data, and despite the high cost this may impose, there is no way to retroactively fix them. Indeed, the history of smart contract development is rife with incidents of buggy smart contracts destroying hundreds of millions of dollars of value. Despite this, the promise of a worldwide social contract that is enforced fairly and uniformly, dispassionate of the participants’ circumstances, has inspired a generation of software developers to ply their talents to build a stable, predictable world in a time of political and economic precarity.

Clarity of Mind is a timely and much-needed resource for navigating this new world. Mr. Janssen, as both an educator and software developer, expertly guides novice and experienced readers alike on the path to successful smart contract development in the Clarity programming language. In this book, the reader not only finds a formidable language reference, but also a plethora of code examples, and proven strategies and tactics for effective smart contract design and implementation. In addition, this book familiarizes the reader with contemporary design standards for common smart contracts, such as those for implementing fungible and non-fungible tokens.

The book is approachable to all audiences, and its open-source nature guarantees that new knowledge will be assimilated in real-time as the rapidly-evolving landscape of smart contracts grows. It provides the most thorough description of the Clarity programming language outside of the language’s source code, and is poised to be the preeminent language guide and cookbook for the foreseeable future.

Jude Nelson, Stacks Open Internet Foundation, 2022

Content

Clarity of Mind Foreword Introduction

- Getting started
  - Installing Clarinet
  - Clarity basics
- Types
  - Primitives
  - Sequences
  - Composites
- Keywords
- Storing data
  - Constants
  - Variables
  - Maps
- Functions
  - Public functions
  - Private functions
  - Read-only functions
- Control flow & error handling
  - asserts!
  - try!
  - Unwrap flavours
  - Response checking
- Using Clarinet
  - Creating a new project
  - Writing your first contract
  - Interacting with your contract
  - Testing your contract
- Practice projects
  - Time-locked wallet
  - Smart claimant
  - Multi-signature vault
- Traits
  - Defining traits
  - Implementing traits
  - Passing traits as arguments
- Stacks Improvement Proposals (SIPs)
  - SIP009: NFTs or Non-Fungible Tokens
  - Creating a SIP009 NFT
  - SIP010: Fungible tokens
  - Creating a SIP010 fungible token
- Building a marketplace
  - Setup
  - Listing & cancelling

    - Fulfilling listings
    - Unit tests
- Runtime cost analysis
- Best practices
    - Coding style
    - What to store on-chain
    - Contract upgradability
- Links and resources

Clarity of Mind

Writing productive smart contracts that are predictable.

By *Marvin Janssen* with contributions of *lnow*, *Mike Cohen* and *Albert Catama*.