Smart Contracts and Solidity

As we discussed in [intro_chapter], there are two different types of accounts in Ethereum:externally owned accounts (EOAs) and contract accounts. EOAs are controlled by users, often via software such as a wallet application that is external to the Ethereum platform.In contrast, contract accounts are controlled by program code (also commonly referred to as “smart contracts”) that is executed by the Ethereum Virtual Machine. In short, EOAs are simple accounts without any associated code or data storage, whereas contract accounts have both associated code and data storage. EOAs are controlled by transactions created and cryptographically signed with a private key in the "real world" external to and independent of the protocol, whereas contract accounts do not have private keys and so "control themselves" in the predetermined way prescribed by their smart contract code. Both types of accounts are identified by an Ethereum address. In this chapter, we will discuss contract accounts and the program code that controls them.

What Is a Smart Contract?

The term *smart contract* has been used over the years to describe a wide variety of different things. In the 1990s, cryptographer Nick Szabo coined the term and defined it as “a set of promises, specified in digital form, including protocols within which the parties perform on the other promises.” Since then, the concept of smart contracts has evolved, especially after the introduction of decentralized blockchain platforms with the invention of Bitcoin in 2009. In the context of Ethereum, the term is actually a bit of a misnomer, given that Ethereum smart contracts are neither smart nor legal contracts, but the term has stuck. In this book, we use the term “smart contracts” to refer to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol—i.e., on the decentralized Ethereum world computer.

Let’s unpack that definition:

Computer programs

Smart contracts are simply computer programs. The word “contract” has no legal meaning in this context.

Immutable

Once deployed, the code of a smart contract cannot change. Unlike with traditional software, the only way to modify a smart contract is to deploy a new instance.

Deterministic

The outcome of the execution of a smart contract is the same for everyone who runs it, given the context of the transaction that initiated its execution and the state of the Ethereum blockchain at the moment of execution.

EVM context

Smart contracts operate with a very limited execution context. They can access their own state, the context of the transaction that called them, and some information about the most recent blocks.

Decentralized world computer

The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system as a whole operates as a single "world computer."

Life Cycle of a Smart Contract

Smart contracts are typically written in a high-level language, such as Solidity. But in order to run, they must be compiled to the low-level bytecode that runs in the EVM. Once compiled, they are deployed on the Ethereum platform using a special *contract creation* transaction, which is identified as such by being sent to the special contract creation address, namely 0x0 (see [contract_reg]). Each contract is identified by an Ethereum address, which is derived from the contract creation transaction as a function of the originating account and nonce. The Ethereum address of a contract can be used in a transaction as the recipient, sending funds to the contract or calling one of the contract's functions. Note that, unlike with EOAs, there are no keys associated with an account created for a new smart contract. As the contract creator, you don't get any special privileges at the protocol level (although you can explicitly code them into the smart contract). You certainly don't receive the private key for the contract account, which in fact does not exist—we can say that smart contract accounts own themselves.

Importantly, contracts *only run if they are called by a transaction*. All smart contracts in Ethereum are executed, ultimately, because of a transaction initiated from an EOA. A contract can call another contract that can call another contract, and so on, but the first contract in such a chain of execution will always have been called by a transaction from an EOA. Contracts never run "on their own" or "in the background." Contracts effectively lie dormant until a transaction triggers execution, either directly or indirectly as part of a chain of contract calls. It is also worth noting that smart contracts are not executed "in parallel" in any sense—the Ethereum world computer can be considered to be a single-threaded machine.

Transactions are *atomic*, they are either successfully terminated or reverted. A successful termination of a transaction means different things under different scenarios: (1) if a transaction is sent from an EOA to another EOA then any changes to the global state (e.g. account balances) made by the transaction are recorded; (2) if a transaction is sent from an EOA to a contract that does not invoke any other contracts, then any changes to the global state are recorded (e.g. account balances, state variables of the contracts) (3) if a transaction is sent from an EOA to a contract that only invokes other contracts in a manner that propagates errors, then any changes to the global state are recorded (e.g. account balances, state variables of the contracts); and (4) if a transaction is sent from an EOA to a contract that invokes other contracts in a manner that does not propagates errors, then there may only be some changes to the global state recorded (e.g. account balances, state variables of the non erroring contracts), whereas other changes to the global state are not recorded (e.g. state variables of the erroring contracts). Otherwise, if a transaction is reverted, all of its effects (changes in state) are "rolled back" as if the transaction never ran. A failed transaction is still recorded as having been attempted, and the ether spent on gas for the execution is deducted from the originating account, but it otherwise has no other effects on contract or account state.

As mentioned previously, it is important to remember that a contract's code cannot be changed. However, a contract can be "deleted," removing the code and its internal state (storage) from its address, leaving a blank account. Any transactions sent to that account address after the contract has been deleted do not result in any code execution, because there is no longer any code there to execute.To delete a contract, you execute an EVM opcode called SELFDESTRUCT (previously called SUICIDE).That operation costs “negative gas,” a gas refund, thereby incentivizing the release of network client resources from the deletion of stored state. Deleting a contract in this way does not remove the transaction history (past) of the contract, since the blockchain itself is immutable. It is also important to note that the SELFDESTRUCT capability will only be available if the contract author programmed the smart contract to have that functionality. If the contract’s code does not have a SELFDESTRUCT opcode, or it is inaccessible, the smart contract cannot be deleted.

Introduction to Ethereum High-Level Languages

The EVM is a virtual machine that runs aspecial form of code called *EVM bytecode*, analogous to your computer’s CPU, which runs machine code such as x86_64. We will examine the operation and language of the EVM in much more detail in [evm_chapter]. In this section we will look at how smart contracts are written to run on the EVM.

While it is possible to program smart contracts directly in bytecode, EVM bytecode is rather unwieldy and very difficult for programmers to read and understand. Instead, most Ethereum developers use a high-level language to write programs, and a compiler to convert them into bytecode.

While any high-level language could be adapted to write smart contracts, adapting an arbitrary language to be compilable to EVM bytecode is quite a cumbersome exercise and would in general lead to some amount of confusion. Smart contracts operate in a highly constrained and minimalistic execution environment (the EVM). In addition, a special set of EVM-specific system variables and functions needs to be available. As such, it is easier to build a smart contract language from scratch than it is to make a general-purpose language suitable for writing smart contracts. As a result, a number of special-purpose languages have emerged for programming smart contracts. Ethereum has several such languages, together with the compilers needed to produce EVM-executable bytecode.

In general, programming languages can be classified into two broad programming paradigms: *declarative* and *imperative*, also known as *functional* and *procedural*, respectively. In declarative programming, we write functions that express the *logic* of a program, but not its *flow*.Declarative programming is used to create programs where there are no *side effects*, meaning that there are no changes to state outside of a function. Declarative programming languages include Haskell and SQL.Imperative programming, by contrast, is where a programmer writes a set of procedures that combine the logic and flow of a program. Imperative programming languages include C++ and Java. Some languages are “hybrid,” meaning that they encourage declarative programming but can also be used to express an imperative programming paradigm. Such hybrids include Lisp, JavaScript, and Python. In general, any imperative language can be used to write in a declarative paradigm, but it often results in inelegant code. By comparison, pure declarative languages cannot be used to write in an imperative paradigm. In purely declarative languages, *there are no “variables.”*

While imperative programming is more commonly used by programmers, it can be very difficult to write programs that execute *exactly as expected*. The ability of any part of the program to change the state of any other makes it difficult to reason about a program’s execution and introduces many opportunities for bugs. Declarative programming, by comparison, makes it easier to understand how a program will behave: since it has no side effects, any part of a program can be understood in isolation.

In smart contracts, bugs literally cost money. As a result, it is critically important to write smart contracts without unintended effects. To do that, you must be able to clearly reason about the expected behavior of the program. So, declarative languages play a much bigger role in smart contracts than they do in general-purpose software. Nevertheless, as you will see, the most widely used language for smart contracts (Solidity) is imperative. Programmers, like most humans, resist change!

Currently supported high-level programming languages for smart contracts include (ordered by approximate age):

LLL

A functional (declarative) programming language, with Lisp-like syntax. It was the first high-level language for Ethereum smart contracts but is rarely used today.

Serpent

A procedural (imperative) programming language with a syntax similar to Python. Can also be used to write functional (declarative) code, though it is not entirely free of side effects.

Solidity

A procedural (imperative) programming language with a syntax similar to JavaScript, C++, or Java. The most popular and frequently used language for Ethereum smart contracts.

Vyper

A more recently developed language, similar to Serpent and again with Python-like syntax. Intended to get closer to a pure-functional Python-like language than Serpent, but not to replace Serpent.

Bamboo

A newly developed language, influenced by Erlang, with explicit state transitions and without iterative flows (loops). Intended to reduce side effects and increase auditability. Very new and yet to be widely adopted.

As you can see, there are many languages to choose from. However, of all of these Solidity is by far the most popular, to the point of being the *de facto* high-level language of Ethereum and even other EVM-like blockchains. We will spend most of our time using Solidity, but will also explore some of the examples in other high-level languages to gain an understanding of their different philosophies.

Building a Smart Contract with Solidity

Soliditywas created byDr. Gavin Wood (coauthor of this book) as a language explicitly for writing smart contracts with features to directly support execution in the decentralized environment of the Ethereum world computer. The resulting attributes are quite general, and so it has ended up being used for coding smart contracts on several other blockchain platforms. It was developed by Christian Reitiwessner and then also by Alex Beregszaszi, Liana Husikyan, Yoichi Hirai, and several former Ethereum core contributors. Solidity is now developed and maintained as an independent project on GitHub.

The main "product" of the Solidity project is the Solidity compiler, solc, which converts programs written in the Solidity language to EVM bytecode. The project also manages the important application binary interface (ABI) standard for Ethereum smart contracts, which we will explore in detail in this chapter. Each version of the Solidity compiler corresponds to and compiles a specific version of the Solidity language.

To get started, we will download a binary executable of the Solidity compiler. Then we will develop and compile a simple contract, following on from the example we started with in [intro_chapter].

Selecting a Version of Solidity

Solidity follows a versioning model called *semantic versioning*, which specifies version numbers structured as three numbers separated by dots: *MAJOR.MINOR.PATCH*. The "major" number is incremented for major and *backward-incompatible* changes, the "minor" number is incremented as backward-compatible features are added in between major releases, and the "patch" number is incremented for backward-compatible bug fixes.

At the time of writing, Solidity is at version 0.6.4. The rules for major version 0, which is for initial development of a project, are different: anything may change at any time. In practice, Solidity treats the "minor" number as if it were the major version and the "patch" number as if it were the minor version. Therefore, in 0.6.4, 6 is considered to be the major version and 4 the minor version.

The 0.5 major version release of Solidity is anticipated imminently.

As you saw in [intro_chapter], your Solidity programs can contain a pragma directive that specifies the minimum and maximum versions of Solidity that it is compatible with, and can be used to compile your contract.

Since Solidity is rapidly evolving, it is often better to install the latest release.

Download and Install

There are a number of methods you can use to download and install Solidity, either as a binary release or by compiling from source code. You can find detailed instructions in the Solidity documentation.

Here’s how to install the latest binary release of Solidity on an Ubuntu/Debian operating system, using the apt package manager:

```
<pre data-type="programlisting">
$ <strong>sudo add-apt-repository ppa:ethereum/ethereum</strong>
$ <strong>sudo apt update</strong>
$ <strong>sudo apt install solc</strong>
</pre>
```

Once you have solc installed, check the version by running:

```
<pre data-type="programlisting">
$ <strong>solc --version</strong>
```

```
solc, the solidity compiler commandline interface
Version: 0.6.4+commit.1dca32f3.Linux.g++
</pre>
```

There are a number of other ways to install Solidity, depending on your operating system and requirements, including compiling from the source code directly. For more information see https://github.com/ethereum/solidity.

Development Environment

To develop in Solidity, you can use any text editor and solc on the command line. However, you might find that some text editors designed for development, such as Emacs, Vim, and Atom, offer additional features such as syntax highlighting and macros that make Solidity development easier.

There are also web-based development environments, such as Remix IDE and EthFiddle.

Use the tools that make you productive. In the end, Solidity programs are just plain text files. While fancy editors and development environments can make things easier, you don't need anything more than a simple text editor, such as nano (Linux/Unix), TextEdit (macOS), or even NotePad (Windows). Simply save your program source code with a *.sol* extension and it will be recognized by the Solidity compiler as a Solidity program.

Writing a Simple Solidity Program

In [intro_chapter], we wrote our first Solidity program. When we first built the Faucet contract, we used the Remix IDE to compile and deploy the contract. In this section, we will revisit, improve, and embellish Faucet.

Our first attempt looked like Faucet.sol: A Solidity contract implementing a faucet.

Example 1. Faucet.sol: A Solidity contract implementing a faucet

```
// SPDX-License-Identifier: CC-BY-SA-4.0

// Version of Solidity compiler this program was written for
pragma solidity 0.6.4;

// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive() external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

Compiling with the Solidity Compiler (solc)

Now, we will use the Solidity compiler on the command line to compile our contract directly. The Solidity compiler solc offers a variety of options, which you can see by passing the --help argument.

We use the --bin and --optimize arguments of solc to produce an optimized binary of our example contract:

```
<pre data-type="programlisting">
$ <strong>solc --optimize --bin Faucet.sol</strong>
======= Faucet.sol:Faucet =======
Binary:
6080604052348015610010576000080fd5b5060cc8061001f6000396000f3fe6080604052600436106
01f5760003560e01c80632e1a7d4d14602a576025565b36602557005b600080fd5b34801560355760
0080fd5b506050600480360360208110156604a57600080fd5b50356052565b005b67016345785d8a0
0008111156066576000080fd5b604051339082156108fc029083906000818181858888f19350505050
158015609257573d6000803e3d6000fd5b505056fea26469706673582212205cf23994b22f7ba19eee5
6c77b5fb127bceec1276b6f76ca71b5f95330ce598564736f6c63430006040033
</pre>
```

The result that solc produces is a hex-serialized binary that can be submitted to the Ethereum blockchain.

The Ethereum Contract ABI

In computer software, an *application binary interface* is an interface between two program modules; often, between the operating system and user programs. An ABI defines how data structures and functions are accessed in *machine code*; this is not to be confused with an API, which defines this access in high-level, often human-readable formats as *source code*. The ABI is thus the primary way of encoding and decoding data into and out of machine code.

In Ethereum, the ABI is used to encode contract calls for the EVM and to read data out of transactions. The purpose of an ABI is to define the functions in the contract that can be invoked and describe how each function will accept arguments and return its result.

A contract's ABI is specified as a JSON array of function descriptions (see Functions) and events (see Events). A function description is a JSON object with fields `type`, `name`, `inputs`, `outputs`, `constant`, and `payable`. An event description object has fields `type`, `name`, `inputs`, and `anonymous`.

We use the solc command-line Solidity compiler to produce the ABI for our *Faucet.sol* example contract:

```
<pre data-type="programlisting">
$ <strong>solc --abi Faucet.sol</strong>
======= Faucet.sol:Faucet =======
Contract JSON ABI
[{"inputs":[{"internalType":"uint256","name":"withdraw_amount","type":"uint256"}], \
"name":"withdraw","outputs":[],"stateMutability":"nonpayable","type":"function"}, \
{"stateMutability":"payable","type":"receive"}]
</pre>
```

As you can see, the compiler produces a JSON array describing the two functions that are defined by *Faucet.sol*. This JSON can be used by any application that wants to access the Faucet contract once it is deployed. Using the ABI, an application such as a wallet or DApp browser can construct transactions that call the functions in Faucet with the correct arguments and argument types. For example, a wallet would know that to call the function withdraw it would have to provide a uint256 argument named withdraw_amount. The wallet could prompt the user to provide that value, then create a transaction that encodes it and executes the withdraw function.

All that is needed for an application to interact with a contract is an ABI and the address where the contract has been deployed.

Selecting a Solidity Compiler and Language Version

As we saw in the previous code, our Faucet contract compiles successfully with Solidity version 0.6.4. But what if we had used a different version of the Solidity compiler? The language is still in constant flux and things may change in unexpected ways. Our contract is fairly simple, but what if our program used a feature that was only added in Solidity version 0.6.1 and we tried to compile it with 0.6.0?

To resolve such issues, Solidity offers a *compiler directive* known as a *version pragma* that instructs the compiler that the program expects a specific compiler (and language) version. Let's look at an example:

```
pragma solidity ^0.6.0;
```

The Solidity compiler reads the version pragma and will produce an error if the compiler version is incompatible with the version pragma. In this case, our version pragma says that this program can be compiled by a Solidity compiler with a minimum version of 0.6.0. The symbol ^ states, however, that we allow compilation with any *minor revision* above 0.6.0; e.g., 0.6.1, but not 0.7.0 (which is a major revision, not a minor revision). Pragma directives are not compiled into EVM bytecode. They are only used by the compiler to check compatibility.

Let's add a pragma directive to our Faucet contract. We will name the new file *Faucet2.sol*, to keep track of our changes as we proceed through these examples starting in Faucet2.sol: Adding the version pragma to Faucet.

Example 2. Faucet2.sol: Adding the version pragma to Faucet

```
// SPDX-License-Identifier: CC-BY-SA-4.0

// Version of Solidity compiler this program was written for
pragma solidity ^0.6.4;

// Our first contract is a faucet!
```

```
contract Faucet {
    // Accept any incoming amount
    receive() external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 100000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

Adding a version pragma is a best practice, as it avoids problems with mismatched compiler and language versions. We will explore other best practices and continue to improve the Faucet contract throughout this chapter.

Programming with Solidity

In this section, we will look at some of the capabilities of the Solidity language. As we mentioned in [intro_chapter], our first contract example was very simple and also flawed in various ways. We'll gradually improve it here, while exploring how to use Solidity. This won't be a comprehensive Solidity tutorial, however, as Solidity is quite complex and rapidly evolving. We'll cover the basics and give you enough of a foundation to be able to explore the rest on your own. The documentation for Solidity can be found on the project website.

Data Types

First, let's look at some of the basic data types offered in Solidity:

Boolean (bool)

Boolean value, true or false, with logical operators ! (not), && (and), || (or), == (equal), and != (not equal).

Integer (int, uint)

Signed (int) and unsigned (uint) integers, declared in increments of 8 bits from int8 to uint256. Without a size suffix, 256-bit quantities are used, to match the word size of the EVM.

Fixed point (fixed, ufixed)

Fixed-point numbers, declared with (`u`)`fixedMxN` where *M* is the size in bits (increments of 8 up to 256) and *N* is the number of decimals after the point (up to 18); e.g., ufixed32x2.

Address

A 20-byte Ethereum address. The address object has many helpful member functions, the main ones being balance (returns the account balance) and `transfer` (transfers ether to the account).

Byte array (fixed)

Fixed-size arrays of bytes, declared with bytes1 up to bytes32.

Byte array (dynamic)

Variable-sized arrays of bytes, declared with bytes or string.

Enum

User-defined type for enumerating discrete values: enum NAME {LABEL1, LABEL 2, ...}.

Arrays

An array of any type, either fixed or dynamic: uint32[][5] is a fixed-size array of five dynamic arrays of unsigned integers.

Struct

User-defined data containers for grouping variables: `struct NAME {TYPE1 VARIABLE1; TYPE2 VARIABLE2; ...}`.

Mapping

Hash lookup tables for *key* => *value* pairs: mapping(KEY_TYPE => VALUE_TYPE) NAME.

In addition to these data types, Solidity also offers a variety of value literals that can be used to calculate different units:

Time units

The units seconds, minutes, hours, and days can be used as suffixes, converting to multiples of the base unit seconds.

Ether units

The units wei, finney, szabo, and ether can be used as suffixes, converting to multiples of the base unit wei.

In our Faucet contract example, we used a uint (which is an alias for uint256) for the withdraw_amount variable. We also indirectly used an address variable, which we set with msg.sender. We will use more of these data types in our examples in the rest of this chapter.

Let’s use one of the unit multipliers to improve the readability of our example contract. In the withdraw function we limit the maximum withdrawal, expressing the limit in wei, the base unit of ether:

```
require(withdraw_amount <= 100000000000000000);
```

That’s not very easy to read. We can improve our code by using the unit multiplier ether, to express the value in ether instead of wei:

```
require(withdraw_amount <= 0.1 ether);
```

Predefined Global Variables and Functions

When a contract is executed in the EVM, it has access to a small set of global objects. These include the block, msg, and tx objects. In addition, Solidity exposes a number of EVM opcodes as predefined functions. In this section we will examine the variables and functions you can access from within a smart contract in Solidity.

Transaction/message call context

The msg object is the transaction call (EOA originated) or message call (contract originated) that launched this contract execution. It contains a number of useful attributes:

msg.sender

We've already used this one. It represents the address that initiated this contract call, not necessarily the originating EOA that sent the transaction. If our contract was called directly by an EOA transaction, then this is the address that signed the transaction, but otherwise it will be a contract address.

msg.value

The value of ether sent with this call (in wei).

msg.gas

The amount of gas left in the gas supply of this execution environment. This was deprecated in Solidity v0.4.21 and replaced by the gasleft function.

msg.data

The data payload of this call into our contract.

msg.sig

The first four bytes of the data payload, which is the function selector.

NOTE Whenever a contract calls another contract, the values of all the attributes of msg change to reflect the new caller's information. The only exception to this is the delegatecall function, which runs the code of another contract/library within the original msg context.

Transaction context

The tx object provides a means of accessing transaction-related information:

tx.gasprice

The gas price in the calling transaction.

tx.origin

The address of the originating EOA for this transaction. WARNING: unsafe!

Block context

The block object contains information about the current block:

block.blockhash(__blockNumber__)

The block hash of the specified block number, up to 256 blocks in the past. Deprecated and replaced with the blockhash function in Solidity v0.4.22.

block.coinbase

The address of the recipient of the current block's fees and block reward.

block.difficulty

The difficulty (proof of work) of the current block.

block.gaslimit

The maximum amount of gas that can be spent across all transactions included in the current block.

block.number

The current block number (blockchain height).

block.timestamp

The timestamp placed in the current block by the miner (number of seconds since the Unix epoch).

address object

Any address, either passed as an input or cast from a contract object, has a number of attributes and methods:

address.balance

The balance of the address, in wei. For example, the current contract balance is address(this).balance.

address.transfer(__amount__)

Transfers the amount (in wei) to this address, throwing an exception on any error. We used this function in our Faucet example as a method on the msg.sender address, as msg.sender.transfer.

address.send(__amount__)

Similar to transfer, only instead of throwing an exception, it returns false on error. WARNING: always check the return value of send.

address.call(__payload__)

Low-level CALL function—can construct an arbitrary message call with a data payload. Returns false on error. WARNING: unsafe—recipient can (accidentally or maliciously) use up all your gas, causing your contract to halt with an OOG exception; always check the return value of call.

address.delegatecall(__payload__)

Low-level DELEGATECALL function, like callcode(...) but with the full msg context seen by the current contract. Returns false on error. WARNING: advanced use only!

Built-in functions

Other functions worth noting are:

addmod, mulmod

For modulo addition and multiplication. For example, addmod(x,y,k) calculates (x + y) % k.

keccak256, sha256, sha3, ripemd160

Functions to calculate hashes with various standard hash algorithms.

ecrecover

Recovers the address used to sign a message from the signature.

selfdestruct(__recipient_address__)

Deletes the current contract, sending any remaining ether in the account to the recipient address.

this

The address of the currently executing contract account.

Contract Definition

Solidity's principal data type is contract; our Faucet example simply defines a `contract` object. Similar to any object in an object-oriented language, the contract is a container that includes data and methods.

Solidity offers two other object types that are similar to a contract:

interface

An interface definition is structured exactly like a contract, except none of the functions are defined, they are only declared. This type of declaration is often called a *stub*; it tells you the functions' arguments and return types without any implementation. An interface specifies the "shape" of a contract; when inherited, each of the functions declared by the interface must be defined by the child.

library

A library contract is one that is meant to be deployed only once and used by other contracts, using the delegatecall method (see address object).

Functions

Within a contract, we define functions that can be called by an EOA transaction or another contract. In our Faucet example, we have two functions: withdraw and the (unnamed) *fallback* function.

The syntax we use to declare a function in Solidity is as follows:

```
<pre data-type="programlisting">
function FunctionName([<em>parameters</em>]) {public|private|internal|external}
[pure|view|payable] [<em>modifiers</em>] [returns (<em>return types</em>)]
</pre>
```

Let’s look at each of these components:

FunctionName

The name of the function, which is used to call the function in a transaction (from an EOA), from another contract, or even from within the same contract.One function in each contract may be defined as a fallback function using the "fallback" keyword or a receive ether function defined using the "receive" keyword. If present, the receive ether function is called whenever the call data is empty (whether or not ether is received). Otherwise, the fallback function is called when no other function is named. The fallback function cannot have any arguments or return anything.

parameters

Following the name, we specify the arguments that must be passed to the function, with their names and types. In our Faucet example we defined uint withdraw_amount as the only argument to the `withdraw` function.

The next set of keywords (public, private, internal, external) specify the function’s *visibility*:

public

Public is the default; such functions can be called by other contracts or EOA transactions, or from within the contract. In our Faucet example, both functions are defined as public.

external

External functions are like public functions, except they cannot be called from within the contract unless explicitly prefixed with the keyword this.

internal

Internal functions are only accessible from within the contract—they cannot be called by another contract or EOA transaction. They can be called by derived contracts (those that inherit this one).

private

Private functions are like internal functions but cannot be called by derived contracts.

Keep in mind that the terms *internal* and *private* are somewhat misleading. Any function or data inside a contract is always *visible* on the public blockchain, meaning that anyone can see the code or data. The keywords described here only affect how and when a function can be *called*.

The second set of keywords (pure, constant, view, payable) affect the behavior of the function:

constant or view

A function marked as a *view* promises not to modify any state.The term *constant* is an alias for view that will be deprecated in a future release. At this time, the compiler does not enforce the view modifier, only producing a warning, but this is expected to become an enforced keyword in v0.5 of Solidity.

pure

A pure function is one that neither reads nor writes any variables in storage. It can only operate on arguments and return data, without reference to any stored data. Pure functions are intended to encourage declarative-style programming without side effects or state.

payable

A payable function is one that can accept incoming payments. Functions not declared as payable will reject incoming payments. There are two exceptions, due to design decisions in the EVM: coinbase payments and SELFDESTRUCT inheritance will be paid even if the fallback function is not declared as payable, but this makes sense because code execution is not part of those payments anyway.

As you can see in our Faucet example, we have one payable function (the fallback function), which is the only function that can receive incoming payments.

Contract Constructor and selfdestruct

There is a special function that is only used once. When a contract is created, it also runs the *constructor function* if one exists, to initialize the state of the contract. The constructor is run in the same transaction as the contract creation. The constructor function is optional; you'll notice our Faucet example doesn't have one.

Constructors can be specified in two ways. Up to and including in Solidity v0.4.21, the constructor is a function whose name matches the name of the contract, as you can see here:

```
contract MEContract {
    function MEContract() {
        // This is the constructor
    }
}
```

The difficulty with this format is that if the contract name is changed and the constructor function name is not changed, it is no longer a constructor. Likewise, if there is an accidental typo in the naming of the contract and/or constructor, the function is again no longer a constructor. This can cause some pretty nasty, unexpected, and difficult-to-find bugs. Imagine for example if the constructor is setting the owner of the contract for purposes of control. If the function is not actually the constructor because of a naming error, not only will the owner be left unset at the time of contract creation, but the function may also be deployed as a permanent and "callable" part of the contract, like a normal function, allowing any third party to hijack the contract and become the "owner" after contract creation.

To address the potential problems with constructor functions being based on having an identical name as the contract, Solidity v0.4.22 introduces a constructor keyword that operates like a constructor function but does not have a name. Renaming the contract does not affect the constructor at all. Also, it is easier to identify which function is the constructor. It looks like this:

```
pragma ^0.4.22
contract MEContract {
    constructor () {
        // This is the constructor
    }
}
```

To summarize, a contract’s life cycle starts with a creation transaction from an EOA or contract account. If there is a constructor, it is executed as part of contract creation, to initialize the state of the contract as it is being created, and is then discarded.

The other end of the contract’s life cycle is *contract destruction*.Contracts are destroyed by a special EVM opcode called SELFDESTRUCT. It used to be called `SUICIDE`, but that name was deprecated due to the negative associations of the word. In Solidity, this opcode is exposed as a high-level built-in function called selfdestruct, which takes one argument: the address to receive any ether balance remaining in the contract account. It looks like this:

```
selfdestruct(address recipient);
```

Note that you must explicitly add this command to your contract if you want it to be deletable—this is the only way a contract can be deleted, and it is not present by default. In this way, users of a contract who might rely on a contract being there forever can be certain that a contract can’t be deleted if it doesn’t contain a `SELFDESTRUCT` opcode.

Adding a Constructor and selfdestruct to Our Faucet Example

The Faucet example contract we introduced in [intro_chapter] does not have any constructor or selfdestruct functions. It is an eternal contract that cannot be deleted. Let’s change that, by adding a constructor and selfdestruct function. We probably want selfdestruct to be callable *only* by the EOA that originally created the contract. By convention, this is usually stored in an address variable called owner. Our constructor sets the owner variable, and the selfdestruct function will first check that the owner called it directly.

First, our constructor:

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.6.0;

// Our first contract is a faucet!
contract Faucet {

    address owner;

    // Initialize Faucet contract: set owner
    constructor() {
        owner = msg.sender;
    }

    [...]
}
```

Our contract now has an address type variable named owner. The name "owner" is not special in any way. We could call this address variable "potato" and still use it the same way. The name owner simply makes its purpose clear.

Next, our constructor, which runs as part of the contract creation transaction, assigns the address from msg.sender to the owner variable. We used msg.sender in the `withdraw` function to identify the initiator of the withdrawal request. In the constructor, however, the msg.sender is the EOA or contract address that initiated contract creation. We know this is the case *because* this is a constructor function: it only runs once, during contract creation.

Now we can add a function to destroy the contract. We need to make sure that only the owner can run this function, so we will use a require statement to control access. Here's how it will look:

```
// Contract destructor
function destroy() public {
    require(msg.sender == owner);
    selfdestruct(owner);
}
```

If anyone calls this destroy function from an address other than owner, it will fail. But if the same address stored in owner by the constructor calls it, the contract will self-destruct and send any remaining balance to the owner address. Note that we did not use the unsafe tx.origin to determine whether the owner wished to destroy the contract—using tx.origin would allow malign contracts to destroy your contract without your permission.

Function Modifiers

Solidity offers a special type of function called a *function modifier*. You apply modifiers to functions by adding the modifier name in the function declaration. Modifiers are most often used to create conditions that apply to many functions within a contract. We have an access control statement already, in our destroy function. Let's create a function modifier that expresses that condition:

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

This function modifier, named onlyOwner, sets a condition on any function that it modifies, requiring that the address stored as the owner of the contract is the same as the address of the transaction's msg.sender. This is the basic design pattern for access control, allowing only the owner of a contract to execute any function that has the onlyOwner modifier.

You may have noticed that our function modifier has a peculiar syntactic "placeholder" in it, an underscore followed by a semicolon (&#95;;). This placeholder is replaced by the code of the function that is being modified. Essentially, the modifier is "wrapped around" the modified function, placing its code in the location identified by the underscore character.

To apply a modifier, you add its name to the function declaration. More than one modifier can be applied to a function; they are applied in the sequence they are declared, as a space-separated list.

Let’s rewrite our destroy function to use the onlyOwner modifier:

```
function destroy() public onlyOwner {
    selfdestruct(owner);
}
```

The function modifier’s name (onlyOwner) is after the keyword public and tells us that the destroy function is modified by the onlyOwner modifier. Essentially, you can read this as "Only the owner can destroy this contract." In practice, the resulting code is equivalent to "wrapping" the code from onlyOwner around destroy.

Function modifiers are an extremely useful tool because they allow us to write preconditions for functions and apply them consistently, making the code easier to read and, as a result, easier to audit for security. They are most often used for access control, but they are quite versatile and can be used for a variety of other purposes.

Inside a modifier, you can access all the values (variables and arguments) visible to the modified function. In this case, we can access the owner variable, which is declared within the contract. However, the inverse is not true: you cannot access any of the modifier’s variables inside the modified function.

Contract Inheritance

Solidity’s contract object supports *inheritance*, which is a mechanism for extending a base contract with additional functionality. To use inheritance, specify a parent contract with the keyword is:

```
contract Child is Parent {
    ...
}
```

With this construct, the Child contract inherits all the methods, functionality, and variables of Parent. Solidity also supports multiple inheritance, which can be specified by comma-separated contract names after the keyword is:

```
contract Child is Parent1, Parent2 {
    ...
}
```

Contract inheritance allows us to write our contracts in such a way as to achieve modularity, extensibility, and reuse. We start with contracts that are simple and implement the most generic capabilities, then extend them by inheriting those capabilities in more specialized contracts.

In our Faucet contract, we introduced the constructor and destructor, together with access control for an owner, assigned on construction. Those capabilities are quite generic: many contracts will have them. We can define them as generic contracts, then use inheritance to extend them to the Faucet contract.

We start by defining a base contract Owned, which has an owner variable, setting it in the contract's constructor:

```
contract Owned {
    address owner;

    // Contract constructor: set owner
    constructor() {
        owner = msg.sender;
    }

    // Access control modifier
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}
```

Next, we define a base contract Mortal, which inherits Owned:

```
contract Mortal is Owned {
    // Contract destructor
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}
```

As you can see, the Mortal contract can use the onlyOwner function modifier, defined in Owned. It indirectly also uses the owner address variable and the constructor defined in Owned. Inheritance makes each contract simpler and focused on its specific functionality, allowing us to manage the details in a modular way.

Now we can further extend the Owned contract, inheriting its capabilities in Faucet:

```
contract Faucet is Mortal {
    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 0.1 ether);
        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
    // Accept any incoming amount
    receive () external payable {}
}
```

By inheriting Mortal, which in turn inherits Owned, the Faucet contract now has the constructor and destroy functions, and a defined owner. The functionality is the same as when those functions were within Faucet, but now we can reuse those functions in other contracts without writing them again. Code reuse and modularity make our code cleaner, easier to read, and easier to audit.

Error Handling (assert, require, revert)

A contract call can terminate and return an error. Error handling in Solidity is handled by four functions: assert, require, revert, and throw (now deprecated).

When a contract terminates with an error, all the state changes (changes to variables, balances, etc.) are reverted, all the way up the chain of contract calls if more than one contract was called. This ensures that transactions are *atomic*, meaning they either complete successfully or have no effect on state and are reverted entirely.

The assert and require functions operate in the same way, evaluating a condition and stopping execution with an error if the condition is false. By convention, assert is used when the outcome is expected to be true, meaning that we use assert to test internal conditions. By comparison, require is used when testing inputs (such as function arguments or transaction fields), setting our expectations for those conditions.

We’ve used require in our function modifier onlyOwner, to test that the message sender is the owner of the contract:

```
require(msg.sender == owner);
```

The require function acts as a *gate condition*, preventing execution of the rest of the function and producing an error if it is not satisfied.

As of Solidity v0.6.0, require can also include a helpful text message that can be used to show the reason for the error. The error message is recorded in the transaction log. So, we can improve our code by adding an error message in our require function:

```
require(msg.sender == owner, "Only the contract owner can call this function");
```

The revert and throw functions halt the execution of the contract and revert any state changes. The throw function is obsolete and will be removed in future versions of Solidity; you should use revert instead. The revert function can also take an error message as the only argument, which is recorded in the transaction log.

Certain conditions in a contract will generate errors regardless of whether we explicitly check for them. For example, in our Faucet contract, we don’t check whether there is enough ether to satisfy a withdrawal request. That’s because the transfer function will fail with an error and revert the transaction if there is insufficient balance to make the transfer:

```
msg.sender.transfer(withdraw_amount);
```

However, it might be better to check explicitly and provide a clear error message on failure. We can do that by adding a require statement before the transfer:

```
require(this.balance >= withdraw_amount,
        "Insufficient balance in faucet for withdrawal request");
msg.sender.transfer(withdraw_amount);
```

Additional error-checking code like this will increase gas consumption slightly, but it offers better error reporting than if omitted. You will need to find the right balance between gas consumption and verbose error checking based on the expected use of your contract. In the case of a Faucet contract intended for a testnet, we'd probably err on the side of extra reporting even if it costs more gas. Perhaps for a mainnet contract we'd choose to be frugal with our gas usage instead.

Events

Whena transaction completes (successfully or not), it produces a *transaction receipt*, as we will see in [evm_chapter]. The transaction receipt contains *log* entries that provide information about the actions that occurred during the execution of the transaction. *Events* are the Solidity high-level objects that are used to construct these logs.

Events are especially useful for light clients and DApp services, which can "watch" for specific events and report them to the user interface, or make a change in the state of the application to reflect an event in an underlying contract.

Event objects take arguments that are serialized and recorded in the transaction logs, in the blockchain. You can supply the keyword indexed before an argument, to make the value part of an indexed table (hash table) that can be searched or filtered by an application.

We have not added any events in our Faucet example so far, so let's do that. We will add two events, one to log any withdrawals and one to log any deposits. We will call these events Withdrawal and Deposit, respectively. First, we define the events in the Faucet contract:

```
contract Faucet is Mortal {
    event Withdrawal(address indexed to, uint amount);
    event Deposit(address indexed from, uint amount);

    [...]
}
```

We've chosen to make the addresses indexed, to allow searching and filtering in any user interface built to access our Faucet.

Next, we use the emit keyword to incorporate the event data in the transaction logs:

```
// Give out ether to anyone who asks
function withdraw(uint withdraw_amount) public {
    [...]
    msg.sender.transfer(withdraw_amount);
    emit Withdrawal(msg.sender, withdraw_amount);
}
// Accept any incoming amount
receive () external payable {
    emit Deposit(msg.sender, msg.value);
}
```

The resulting *Faucet.sol* contract looks like Faucet8.sol: Revised Faucet contract, with events.

Example 3. Faucet8.sol: Revised Faucet contract, with events

```
// SPDX-License-Identifier: CC-BY-SA-4.0

// Version of Solidity compiler this program was written for
pragma solidity ^0.6.4;

contract Owned {
    address payable owner;

    // Contract constructor: set owner
    constructor() public {
        owner = msg.sender;
    }

    // Access control modifier
    modifier onlyOwner {
        require(msg.sender == owner, "Only the contract owner can call this
function");
        _;
    }
}

contract Mortal is Owned {
    // Contract destructor
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract Faucet is Mortal {
    event Withdrawal(address indexed to, uint amount);
    event Deposit(address indexed from, uint amount);

    // Accept any incoming amount
    receive() external payable {
        emit Deposit(msg.sender, msg.value);
    }

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 0.1 ether);

        require(
            address(this).balance >= withdraw_amount,
            "Insufficient balance in faucet for withdrawal request"
        );

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);

        emit Withdrawal(msg.sender, withdraw_amount);
    }
}
```

Catching events

OK, so we've set up our contract to emit events. How do we see the results of a transaction and "catch" the events? The web3.js library provides a data structure that contains a transaction's logs. Within those we can see the events generated by the transaction.

Let's use truffle to run a test transaction on the revised Faucet contract. Follow the instructions in [truffle] to set up a project directory and compile the `Faucet` code. The source code can be found in the book's GitHub repository under *code/truffle/FaucetEvents*.

```
<pre data-type="programlisting">
$ <strong>truffle develop</strong>
truffle(develop)> <strong>compile</strong>
truffle(develop)> <strong>migrate</strong>
Using network 'develop'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb77ceae7c3f5afb7fbe3a6c5974d352aa844f53f955ee7d707ef6f3f8e6b4e61
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Faucet...
  ... 0xfa850d754314c3fb83f43ca1fa6ee20bc9652d891c00a2f63fd43ab5bfb0d781
  Faucet: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

```
truffle(develop)> <strong>Faucet.deployed().then(i => {FaucetDeployed = i})</strong>
truffle(develop)> <strong>FaucetDeployed.send(web3.utils.toWei(1, "ether")).then(res => \
                  { console.log(res.logs[0].event, res.logs[0].args) })</strong>
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
truffle(develop)> <strong>FaucetDeployed.withdraw(web3.utils.toWei(0.1, "ether")).then(res
=> \
                  { console.log(res.logs[0].event, res.logs[0].args) })</strong>
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
</pre>
```

After deploying the contract using the deployed function, we execute two transactions. The first transaction is a deposit (using send), which emits a Deposit event in the transaction logs:

```
Deposit { from: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 18, c: [ 10000 ] } }
```

Next, we use the withdraw function to make a withdrawal. This emits a Withdrawal event:

```
Withdrawal { to: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  amount: BigNumber { s: 1, e: 17, c: [ 1000 ] } }
```

To get these events, we looked at the logs array returned as a result (res) of the transactions. The first log entry (logs[0]) contains an event name in logs[0].event and the event arguments in logs[0].args. By showing these on the console, we can see the emitted event name and the event arguments.

Events are a very useful mechanism, not only for intra-contract communication, but also for debugging during development.

Calling Other Contracts (send, call, callcode, delegatecall)

Calling other contracts from within your contract is a very useful but potentially dangerous operation. We'll examine the various ways you can achieve this and evaluate the risks of each method. In short, the risks arise from the fact that you may not know much about a contract you are calling into or that is calling into your contract. When writing smart contracts, you must keep in mind that while you may mostly expect to be dealing with EOAs, there is nothing to stop arbitrarily complex and perhaps malign contracts from calling into and being called by your code.

Creating a new instance

The safest way to call another contract is if you create that other contract yourself. That way, you are certain of its interfaces and behavior. To do this, you can simply instantiate it, using the keyword new, as in other object-oriented languages. In Solidity, the keyword new will create the contract on the blockchain and return an object that you can use to reference it. Let's say you want to create and call a Faucet contract from within another contract called Token:

```
contract Token is Mortal {
    Faucet _faucet;

    constructor() {
        _faucet = new Faucet();
    }
}
```

This mechanism for contract construction ensures that you know the exact type of the contract and its interface. The contract Faucet must be defined within the scope of Token, which you can do with an import statement if the definition is in another file:

```
import "Faucet.sol";

contract Token is Mortal {
    Faucet _faucet;

    constructor() {
        _faucet = new Faucet();
    }
}
```

You can optionally specify the value of ether transfer on creation, and pass arguments to the new contract's constructor:

```
import "Faucet.sol";

contract Token is Mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }
}
```

You can also then call the Faucet functions. In this example, we call the destroy function of Faucet from within the destroy function of Token:

```
import "Faucet.sol";

contract Token is Mortal {
    Faucet _faucet;

    constructor() {
        _faucet = (new Faucet).value(0.5 ether)();
    }
    function destroy() ownerOnly {
        _faucet.destroy();
    }
}
```

Note that while you are the owner of the Token contract, the Token contract itself owns the new Faucet contract, so only the Token contract can destroy it.

Addressing an existing instance

Another way you can call a contract is by casting the address of an existing instance of the contract. With this method, you apply a known interface to an existing instance. It is therefore critically important that you know, for sure, that the instance you are addressing is in fact of the type you assume. Let’s look at an example:

```
import "Faucet.sol";

contract Token is Mortal {

    Faucet _faucet;

    constructor(address _f) {
        _faucet = Faucet(_f);
        _faucet.withdraw(0.1 ether);
    }
}
```

Here, we take an address provided as an argument to the constructor, _f, and we cast it to a Faucet object. This is much riskier than the previous mechanism, because we don’t know for sure whether that address actually is a Faucet object. When we call withdraw, we are assuming that it accepts the same arguments and executes the same code as our Faucet declaration, but we can’t be sure. For all we know, the withdraw function at this address could execute something completely different from what we expect, even if it is named the same. Using addresses passed as input and casting them into specific objects is therefore much more dangerous than creating the contract yourself.

Raw call, delegatecall

Solidity offers some even more "low-level" functions for calling other contracts. These correspond directly to EVM opcodes of the same name and allow us to construct a contract-to-contract call manually. As such, they represent the most flexible *and* the most dangerous mechanisms for calling other contracts.

Here’s the same example, using a call method:

```
contract Token is Mortal {
    constructor(address _faucet) {
        _faucet.call("withdraw", 0.1 ether);
    }
}
```

As you can see, this type of call is a *blind* call into a function, very much like constructing a raw transaction, only from within a contract's context.It can expose your contract to a number of security risks, most importantly *reentrancy*, which we will discuss in more detail in [reentrancy_security]. The call function will return false if there is a problem, so you can evaluate the return value for error handling:

```
contract Token is Mortal {
    constructor(address _faucet) {
        if !(_faucet.call("withdraw", 0.1 ether)) {
            revert("Withdrawal from faucet failed");
        }
    }
}
```

Another variant of call is delegatecall, which replaced the more dangerous callcode. The `callcode` method will be deprecated soon, so it should not be used.

As mentioned in address object, a delegatecall is different from a call in that the msg context does not change. For example, whereas a call changes the value of msg.sender to be the calling contract, a delegatecall keeps the same msg.sender as in the calling contract. Essentially, delegatecall runs the code of another contract inside the context of the execution of the current contract. It is most often used to invoke code from a library. It also allows you to draw on the pattern of using library functions stored elsewhere, but have that code work with the storage data of your contract.

The delegate call should be used with great caution. It can have some unexpected effects, especially if the contract you call was not designed as a library.

Let's use an example contract to demonstrate the various call semantics used by call and delegatecall for calling libraries and contracts. In CallExamples.sol: An example of different call semantics, we use an event to log the details of each call and see how the calling context changes depending on the call type.

Example 4. CallExamples.sol: An example of different call semantics

```
pragma solidity ^0.4.22;

contract calledContract {
    event callEvent(address sender, address origin, address from);
    function calledFunction() public {
        emit callEvent(msg.sender, tx.origin, this);
    }
}

library calledLibrary {
    event callEvent(address sender, address origin,  address from);
    function calledFunction() public {
        emit callEvent(msg.sender, tx.origin, this);
    }
}

contract caller {

    function make_calls(calledContract _calledContract) public {

    // Calling calledContract and calledLibrary directly
    _calledContract.calledFunction();
    calledLibrary.calledFunction();

    // Low-level calls using the address object for calledContract
    require(address(_calledContract).
            call(bytes4(keccak256("calledFunction()"))));
    require(address(_calledContract).
            delegatecall(bytes4(keccak256("calledFunction()"))));



    }
}
```

As you can see in this example, our main contract is caller, which calls a library calledLibrary and a contract calledContract. Both the called library and the contract have identical calledFunction functions, which emit an event callEvent. The event callEvent logs three pieces of data: msg.sender, tx.origin, and this. Each time calledFunction is called it may have a different execution context (with different values for potentially all the context variables), depending on whether it is called directly or through delegatecall.

In caller, we first call the contract and library directly, by invoking calledFunction in each. Then, we explicitly use the low-level functions call and delegatecall to call calledContract.calledFunction. This way we can see how the various calling mechanisms behave.

Let’s run this in a Truffle development environment and capture the events, to see how it looks:

```
<pre data-type="programlisting">
truffle(develop)> <strong>migrate</strong>
Using network 'develop'.
[...]
Saving artifacts...
truffle(develop)> <strong>(await web3.eth.getAccounts())[0]</strong>
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
truffle(develop)> <strong>caller.address</strong>
'0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
truffle(develop)> <strong>calledContract.address</strong>
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
truffle(develop)> <strong>calledLibrary.address</strong>
```

```
'0xf25186b5081ff5ce73482ad761db0eb0d25abfbf'
truffle(develop)> <strong>caller.deployed().then( i => { callerDeployed = i })</strong>

truffle(develop)> <strong>callerDeployed.make_calls(calledContract.address).then(res => \
                  { res.logs.forEach( log => { console.log(log.args) })})</strong>
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
{ sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10' }
{ sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
  from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f' }
</pre>
```

Let’s see what happened here. We called the make_calls function and passed the address of calledContract, then caught the four events emitted by each of the different calls. Let’s look at the make_calls function and walk through each step.

The first call is:

```
_calledContract.calledFunction();
```

Here, we’re calling calledContract.calledFunction directly, using the high-level ABI for calledFunction. The event emitted is:

```
sender: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

As you can see, msg.sender is the address of the caller contract. The tx.origin is the address of our account, web3.eth.accounts[0], that sent the transaction to caller. The event was emitted by calledContract, as we can see from the last argument in the event.

The next call in make_calls is to the library:

```
calledLibrary.calledFunction();
```

It looks identical to how we called the contract, but behaves very differently. Let’s look at the second event emitted:

```
sender: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
origin: '0x627306090abab3a6e1400e9345bc60c78a8bef57',
from: '0x8f0483125fcb9aaaefa9209d8e9d7b9c8b9fb90f'
```

This time, the msg.sender is not the address of caller. Instead, it is the address of our account, and is the same as the transaction origin. That's because when you call a library, the call is always delegatecall and runs within the context of the caller. So, when the calledLibrary code was running, it inherited the execution context of caller, as if its code was running inside caller. The variable this (shown as from in the event emitted) is the address of caller, even though it is accessed from within `calledLibrary`.

The next two calls, using the low-level call and delegatecall, verify our expectations, emitting events that mirror what we just saw.

Gas Considerations

Gas, described in more detail in [gas], is an incredibly important consideration in smart contract programming. Gas is a resource constraining the maximum amount of computation that Ethereum will allow a transaction to consume. If the gas limit is exceeded during computation, the following series of events occurs:

- An "out of gas" exception is thrown.
- The state of the contract prior to execution is restored (reverted).
- All ether used to pay for the gas is taken as a transaction fee; it is *not* refunded.

Because gas is paid by the user who initiates the transaction, users are discouraged from calling functions that have a high gas cost. It is thus in the programmer's best interest to minimize the gas cost of a contract's functions. To this end, there are certain practices that are recommended when constructing smart contracts, so as to minimize the gas cost of a function call.

Avoid Dynamically Sized Arrays

Any loop through a dynamically sized array where a function performs operations on each element or searches for a particular element introduces the risk of using too much gas. Indeed, the contract may run out of gas before finding the desired result, or before acting on every element, thus wasting time and ether without giving any result at all.

Avoid Calls to Other Contracts

Calling other contracts, especially when the gas cost of their functions is not known, introduces the risk of running out of gas. Avoid using libraries that are not well tested and broadly used. The less scrutiny a library has received from other programmers, the greater the risk of using it.

Estimating Gas Cost

If you need to estimate the gas necessary to execute a certain method of a contract considering its arguments, you could use the following procedure:

```
var contract = web3.eth.contract(abi).at(address);
var gasEstimate = contract.myAweSomeMethod.estimateGas(arg1, arg2,
    {from: account});
```

gasEstimate will tell you the number of gas units needed for its execution. It is an estimate because of the Turing completeness of the EVM—it is relatively trivial to create a function that will take vastly different amounts of gas to execute different calls. Even production code can change execution paths in subtle ways, resulting in hugely different gas costs from one call to the next. However, most functions are sensible and estimateGas will give a good estimate most of the time.

To obtain the gas price from the network you can use:

```
var gasPrice = web3.eth.getGasPrice();
```

And from there you can estimate the gas cost:

```
var gasCostInEther = web3.utils.fromWei((gasEstimate * gasPrice), 'ether');
```

Let's apply our gas estimation functions to estimating the gas cost of our Faucet example, using the code from the book's repository.

Start Truffle in development mode and execute the JavaScript file in gas_estimates.js: Using the estimateGas function, *gas_estimates.js*.

Example 5. gas_estimates.js: Using the estimateGas function

```
var FaucetContract = artifacts.require("./Faucet.sol");

FaucetContract.web3.eth.getGasPrice(function(error, result) {
    var gasPrice = Number(result);
    console.log("Gas Price is " + gasPrice + " wei"); // "10000000000000"

    // Get the contract instance
    FaucetContract.deployed().then(function(FaucetContractInstance) {

        // Use the keyword 'estimateGas' after the function name to get the gas
        // estimation for this particular function (aprove)
        FaucetContractInstance.send(web3.utils.toWei(1, "ether"));
        return FaucetContractInstance.withdraw.estimateGas(web3.utils.toWei(0.1,
"ether"));

    }).then(function(result) {
        var gas = Number(result);

        console.log("gas estimation = " + gas + " units");
        console.log("gas cost estimation = " + (gas * gasPrice) + " wei");
        console.log("gas cost estimation = " +
                FaucetContract.web3.utils.fromWei((gas * gasPrice), 'ether') + "
ether");
    });
});
```

Here’s how that looks in the Truffle development console:

```
<pre data-type="programlisting">
$ <strong>truffle develop</strong>

truffle(develop)> <strong>exec gas_estimates.js</strong>
Using network 'develop'.

Gas Price is 20000000000 wei
gas estimation = 31397 units
gas cost estimation = 627940000000000 wei
gas cost estimation = 0.00062794 ether
</pre>
```

It is recommended that you evaluate the gas cost of functions as part of your development workflow, to avoid any surprises when deploying contracts to the mainnet.

Conclusions

In this chapter we started working with smart contracts in detail and explored the Solidity contract programming language. We took a simple example contract, *Faucet.sol*, and gradually improved it and made it more complex, using it to explore various aspects of the Solidity language. In [vyper_chap] we will work with Vyper, another contract-oriented programming language. We will compare Vyper to Solidity, showing some of the differences in the design of these two languages and deepening our understanding of smart contract programming.