11
Tools, Languages, and Frameworks for Ethereum Developers

This chapter is an introduction to the tools, languages, and frameworks used for Ethereum smart contract development. We will examine different methods of developing smart contracts for the Ethereum blockchain. We will discuss various constructs of the **Solidity** language in detail, which is currently the most popular development language for smart contract development on Ethereum.

In this chapter, we will cover the following topics:

- Languages
- Compilers
- Tools and libraries
- Frameworks
- Contract development and deployment
- The Solidity language

There are a number of tools available for Ethereum development, including clients, IDEs, and development frameworks.

The content of this chapter does not include all frameworks and tools that are out there for development on Ethereum. It shows the most commonly used tools and frameworks, including some that we will use in our examples in the following chapter.

There are a number of resources available related to development tools for Ethereum at the following address: http://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#developer-tools

---

Since we have discussed some of the main tools available in Ethereum in previous chapters, such as Remix IDE and MetaMask, this chapter will focus mainly on Solidity, Ganache, `solc`, and Truffle. Some other elements, such as prerequisites (Node.js), will also be introduced briefly.

We'll start by exploring some of the programming languages that can be used in the Ethereum blockchain.

Languages

Smart contracts can be programmed in a variety of languages for the Ethereum blockchain. There are three main languages that can be used to write contracts, and three that are of historical interest:

- **Solidity**: This language has now become a standard for contract writing for Ethereum. This language is the focus of this chapter.
- **Vyper**: This language is a Python-like experimental language that is being developed to bring security, simplicity, and auditability to smart contract development.

- **Yul**: This is an intermediate language that has the ability to compile to different backends such as EVM and `ewasm`. The design goals of Yul mainly include readability, easy control flow, optimization, formal verification, and simplicity.
- **Mutan**: This is a Go-style language, which was deprecated in early 2015 and is no longer used.
- **LLL**: This is a **Low-level Lisp-like Language**, hence the name LLL. This is also no longer used.
- **Serpent**: This is a simple and clean Python-like language. It is not used for contract development anymore and is not supported by the community.

As Solidity code needs to be compiled into bytecode, we need a compiler to do so. In the next section, we will introduce the Solidity compiler.

The Solidity compiler

Compilers are used to convert high-level contract source code into the format that the Ethereum execution environment understands. The Solidity compiler, `solc`, is the most common one in use.

`solc` converts from a high-level Solidity language into **Ethereum Virtual Machine (EVM)** bytecode so that it can be executed on the blockchain by the EVM.

Installing solc

`solc` can be installed on a Linux Ubuntu operating system using the following command:

```
$ sudo apt-get install solc
```

If **Personal Package Archives (PPAs)** are not already installed, those can be installed by running the following commands:

```
$ sudo add-apt-repository ppa:ethereum/ethereum
$ sudo apt-get update
```

To install `solc` on macOS, execute the following commands:

```
$ brew tap ethereum/ethereum
```

This command will add the Ethereum repository to the list of `brew` formulas:

```
$ brew install solidity
```

This command will produce a long output and may take a few minutes to complete. If there are no errors produced, then eventually it will install Solidity.

In order to verify that the Solidity compiler is installed and to validate the version of the compiler, the following command can be used:

```
$ solc --version
```

This command will produce the output shown as follows, displaying the version of the Solidity compiler:

```
solc, the solidity compiler commandline interface
Version: 0.8.11+commit.d7f03943.Darwin.appleclang
```

This output confirms that the Solidity compiler is installed successfully.

Experimenting with solc

`solc` supports a variety of functions. A few examples are shown as follows. As an example, we'll use a simple contract, `Addition.sol`:

```
pragma solidity ^0.8.0;
contract Addition
{
  uint8 x;
  function addx(uint8 y, uint8 z ) public
  {
    x = y + z;
  }
  function retrievex() view public returns (uint8)
  {
    return x;
  }
}
```

In order to see the smart contract in compiled binary format, we can use the following command:

```
$ solc --bin Addition.sol
```

This command will produce an output similar to the following:

```
======= Addition.sol:Addition =======
Binary:
608060405234801561001057600080fd5b506101008061002060000396000f3fe608060 4052348015600f57600080
```

This output shows the binary translation of the `Addition.sol` contract code represented in hex.

As a gas fee is charged for every operation that the EVM executes, it's a good practice to estimate gas before deploying a contract on a live network. Estimating gas gives a good approximation of how much gas will be consumed by the operations specified in the contract code, which gives an indication of how much ether is required to be spent in order to run a contract. We can use the `--gas` flag for this purpose, as shown in the following example:

```
$ solc --gas Addition.sol
```

This will give the following output:

```
======= Addition.sol:Addition =======
Gas estimation:
construction:
   147 + 100800 = 100947
external:
   addx(uint8,uint8):   infinite
   retrievex(): 2479
```

This output shows how much gas usage is expected for these operations in the `Addition.sol` smart contract. The gas estimation is shown next to each function. For example, the `retrieve()` function is expected to use `2479` gas.

We can generate the **Application Binary Interface** (**ABI**) using `solc`, which is a standard way to interact with the contracts:

```
$ solc --abi Addition.sol
```

This command will produce a file named `Addition.abi` as output. The following are the contents of the output file `Addition.abi`:

```
======= Addition.sol:Addition =======
Contract JSON ABI
[{"inputs":[{"internalType":"uint8","name":"y","type":"uint8"},{"internalType":"uint8","name"
{"inputs":[],"name":"retrievex","outputs":[{"internalType":"uint8","name":"","type":"uint8"}]
```

The preceding output displays the contents of the `Addition.abi` file, which are formatted in JSON style. It consists of inputs and outputs along with their types. We will generate and use ABIs later in this chapter to interact with the deployed smart contracts.

Another useful command to compile and produce a binary compiled file along with an ABI is shown here:

```
$ solc --bin --abi -o bin Addition.sol
```

The message displays if the compiler run is successful; otherwise, errors are reported.

```
Compiler run successful. Artifact(s) can be found in directory bin.
```

This command will produce a message and two files in the output directory `bin`:

- `Addition.abi`: This contains the ABI of the smart contract in JSON format.
- `Addition.bin`: This contains the hex representation of the binary of the smart contract code.

> The ABI encodes information about smart contracts' functions and events. It acts as an interface between EVM-level bytecode and high-level smart contract program code. To interact with a smart contract deployed on the

> Ethereum blockchain, external programs require an ABI and the address of the smart contract.

`solc` is a very powerful command and further options can be explored using the `--help` flag, which will display detailed options. However, the preceding commands used for compilation, ABI generation, and gas estimation should be sufficient for most development and deployment requirements.

Tools, libraries, and frameworks

There are various tools and libraries available for Ethereum. The most common ones are discussed here. In this section, we will first install the prerequisites that are required for developing applications for Ethereum.

Node.js

Node.js is a popular development platform for executing JavaScript code primarily on the backend server side; however, it can also be used for frontends.

As Node.js is required for most of the tools and libraries, we recommend installing it first. Node.js can be installed for your operating system by following the instructions on the official website: **https://nodejs.org/en/**

Ganache

At times, it is not possible to test on the testnet, and the mainnet is obviously not a place to test contracts. A private network can be time-consuming to set up at times. Ganache is a simulated personal blockchain with a command-line interface or a user-friendly GUI to view transactions, blocks, and relevant details. This is a fully working personal blockchain that supports the simulation of Ethereum and its different hard forks, such as **Homestead**, **Byzantium**, **Istanbul**, **Petersburg, or London**. It's available as a CLI and also a GUI.

ganache-cli

Ganache comes in handy when quick testing is required and no testnet is available. It simulates the Ethereum `geth` client behavior and allows faster development and testing. The Ganache command line is available via `npm` as a Node.js package. As such, Node.js should already have been installed and the `npm` package manager should be available. `ganache` can be installed using this command:

```
$ npm install -g ganache
```

In order to start the `ganache` command-line interface, simply issue this command, keep it running in the background, and open another terminal to work on developing contracts:

```
$ ganache
```

When Ganache runs, it will automatically generate 10 accounts and private keys, along with an HD wallet. It will start to listen for incoming con-

nections on TCP port `8545`.

`ganache-cli` has a number of flags to customize the simulated chain according to your requirements. For example, flag `-a` allows you to specify the number of accounts to generate at startup. Similarly `-b` allows the user to configure block time for mining.

Detailed help is available using the following command:

```
$ ganache --help
```

Ganache is a command-line tool. However, at times, it is desirable to have a fully featured tool with a rich **graphical user interface** (**GUI**).

Ganache UI

Ganache UI is based on a JavaScript implementation of the Ethereum blockchain, with a built-in block explorer and mining, making testing locally on the system very easy. You can view transactions, blocks, and addresses in detail on the frontend GUI. It can be downloaded from https://www.trufflesuite.com/ganache.

When you start Ganache for the first time, it will ask whether you want to create a quick blockchain or create a new workspace that can be saved, and it also has advanced setup options:

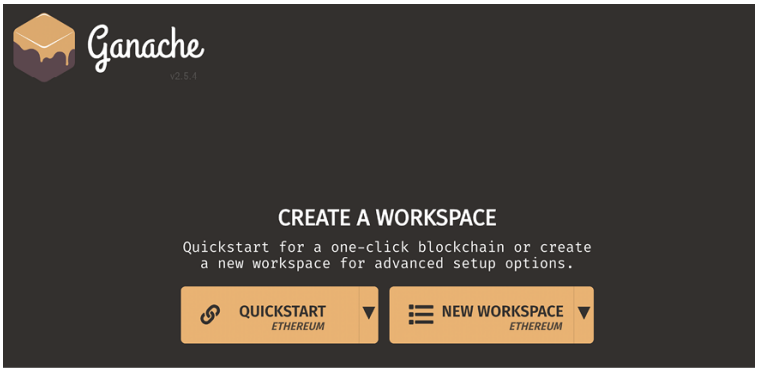

Figure 11.1: Creating a workspace

Select the **QUICKSTART** or **NEW WORKSPACE** option as required. For a quicker temporary setup with default options, which could be useful for simple testing, you can choose **QUICKSTART**. We will choose **NEW WORKSPACE** as we want to explore more advanced features.

If **NEW WORKSPACE** is selected, there are a number of options available to configure the blockchain. One of the configuration options is **WORKSPACE NAME**, where you can specify a name for your project. Additionally, Truffle projects can also be added here—we will cover Truffle in more detail later in the chapter.

Other options include **SERVER**, **ACCOUNTS & KEYS**, **CHAIN**, and **ADVANCED**. The **SERVER** tab is used to configure RPC connectivity by specifying the hostname, port number, and network ID:

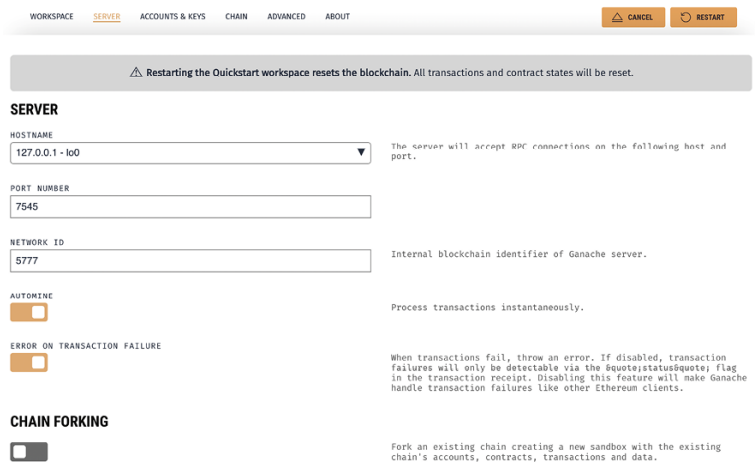

Figure 11.2: Server configuration

ACCOUNTS & KEYS provides options to configure the balance and the number of accounts to generate. The **CHAIN** option provides a configuration interface for specifying the gas limit, gas price, and hard fork, which is required to be simulated, such as Byzantine or Petersburg.

The **ADVANCED** option is available to configure logging and analytics-related settings. Once you have all the configuration options set, save the workspace by selecting **SAVE WORKSPACE**, and the main transaction view of the Ganache personal blockchain will show:

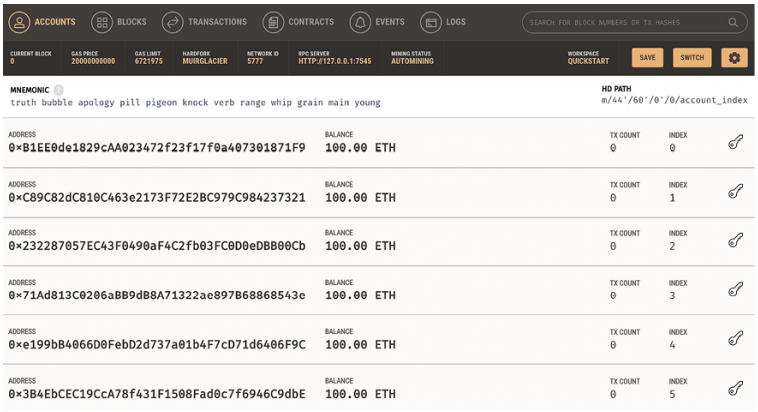

Figure 11.3: Ganache main view

With this, we conclude our introduction to Ganache, a mainstream tool used in blockchain development. Now we will move on to different development frameworks that are available for Ethereum.

There are a number of other notable frameworks available for Ethereum development. It is almost impossible to cover all of them, but an introduction to some of the mainstream frameworks and tools is given as follows.

Truffle

Truffle (available at https://www.trufflesuite.com) is a development environment that makes it easier and simpler to test and deploy Ethereum contracts. Truffle provides contract compilation and linking along with an automated testing framework using Mocha and Chai. It also makes it easier to deploy the contracts to any private, public, or testnet Ethereum blockchain. Also, an asset pipeline is provided, which makes it

easier for all JavaScript files to be processed, making them ready for use by a browser.

Before installation, it is assumed that `node` is available, which can be queried as shown here:

```
$ node -v
v16.15.0
```

If `node` is not available already, then the installation of `node` is required first in order to install truffle. The installation of truffle is very simple and can be done using the following command via **Node Package Manager** (**npm**):

```
$ npm install truffle –g
```

This will take a few minutes; once it is installed, the `truffle` command can be used to display help information and verify that it is installed correctly.

Type `truffle` in the terminal to display usage help:

```
$ truffle
```

This will display all the options that Truffle supports. Alternatively, the repository is available at **https://github.com/trufflesuite/truffle**, which can be cloned locally to install `truffle`. **Git** can be used to clone the repository using the following command:

```
$ git clone https://github.com/trufflesuite/truffle.git
```

We will use Truffle later in *Chapter 12*, *Web3 Development Using Ethereum*, to test and deploy smart contracts on the Ethereum blockchain. For now, we'll continue to explore some of the frameworks used for development on the Ethereum blockchain.

Drizzle

Drizzle is a collection of frontend libraries that allows the easy development of web UIs for decentralized applications. It is based on the Redux store and allows seamless synchronization of contract and transaction data.

Drizzle is installed using the following command:

```
$ npm install --save @drizzle/store
```

Web **User Interface** (**UI**) development is an important part of dApp development. As such, many web techniques and tools, ranging from simple HTML and JavaScript to advanced frameworks such as Redux and React, are used to develop web UIs for dApps.

Other tools

There are many other tools and frameworks available. Information about these tools is available here: `https://ethereum.org/en/developers/local-environment/`. However, we will discuss a few here:

- **Embark** is a complete and powerful developer platform for building and deploying decentralization applications. It is used for smart contract development, configuration, testing, and deployment. It also integrates with **Swarm**, **IPFS**, and **Whisper**. There is also a web interface called **Cockpit** available with Embark, which provides an integrated development environment for easy development and debugging of decentralized applications.
- **Brownie** is a Python-based framework for Ethereum smart contract development and testing. It has the full support of Solidity and Vyper with relevant testing and debugging tools. More information is available at `https://eth-brownie.readthedocs.io/en/stable/`.
- **Waffle** is a framework for smart contract development and testing. It claims to be faster than Truffle. This framework allows dApp development, debugging, and testing in Solidity and Vyper. It is based on `Ethers.js`. More details are available on the official website at `https://getwaffle.io`.
- The `OpenZeppelin` toolkit has a rich set of tools that allow easy smart contract development. It supports compiling, deploying, upgrading, and interacting with smart contracts. Further information is available here: `https://openzeppelin.com/sdk/`.

In this section, we have covered some of the mainstream frameworks that are used in the Ethereum ecosystem for development. In the next section, we will explore which tools are available for writing and deploying smart contracts.

Contract development and deployment

There are various steps that need to be taken in order to develop and deploy contracts. Broadly, these can be divided into three steps: writing, testing, and deploying. After deployment, the next optional step is to create the UI and present it to the end users via a web server. We'll cover that in the following chapter. A web interface is sometimes not needed in contracts where no human input or monitoring is required, but usually there is a requirement to create a web interface so that end users can interact with the contract using familiar web-based interfaces.

Writing smart contracts

The writing step is concerned with writing the contract source code in Solidity. This can be done in any text editor. There are various plugins and add-ons available for Vim in Linux, Atom, and other editors that provide syntax highlighting and formatting for Solidity source code.

Visual Studio Code has become quite popular and is used commonly for Solidity development. There are Solidity plugins available that allow syntax highlighting, formatting, and **IntelliSense**. Also, **Truffle** is available as a plugin, which improves the developer experience considerably.

Both can be installed via the **Extensions** option in Visual Studio Code.

The Solidity plugin for Visual Studio is available in the Visual Studio Marketplace at https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity

Truffle for VS Code is available here: https://marketplace.visualstudio.com/items?itemName=trufflesuite-csi.truffle-vscode

Testing smart contracts

Testing is usually performed by automated means. Earlier in the chapter, you were introduced to Truffle, which uses the Mocha framework to test contracts. However, manual functional testing can be performed as well by using Remix IDE, which was discussed in *Chapter 10*, *Ethereum in Practice*, and running functions manually and validating results. We will cover this in *Chapter 12*, *Web3 Development Using Ethereum*.

Deploying smart contracts

Once the contract is verified, working, and tested on a simulated environment (for example, Ganache) or on a private net, it can be deployed to a public testnet such as **Ropsten** and eventually to the live blockchain (mainnet). We will cover all these steps, including verification, development, and creating a web interface, in the next chapter.

Now that we have covered which tools can be used to write Solidity smart contracts, we will introduce the Solidity language. This will be a brief introduction to Solidity, which should provide the base knowledge required to write smart contracts. The syntax of the language is very similar to C and JavaScript, and it is quite easy to program. We'll start by exploring what a smart contract written in the Solidity language looks like.

The Solidity language

Solidity is the domain-specific language of choice for programming contracts in Ethereum. Its syntax is close to both JavaScript and C. Solidity has evolved into a mature language over the last few years and is quite easy to use, but it still has a long way to go before it can become advanced, standardized, and feature-rich, like other well-established languages such as Java, C, and C#. Nevertheless, it is the most widely used language currently available for programming contracts.

It is a statically typed language, which means that variable type checking in Solidity is carried out at compile time. Each variable, either state or local, must be specified with a type at compile time. This is beneficial in the sense that any validation and checking are completed at compile time and certain types of bugs, such as the interpretation of data types, can be caught earlier in the development cycle instead of at runtime, which could be costly, especially in the case of the blockchain/smart contract paradigm.

Other features of the language include inheritance, libraries, and the ability to define composite data types. Solidity is also called a contract-oriented language. In Solidity, contracts are equivalent to the concept of classes in other object-oriented programming languages.

In the following subsections, we will look at the components of a Solidity source code file, which is important to cover before we move on to writing smart contracts in the next section.

In order to address compatibility issues that may arise from future versions of `solc`, `pragma` can be used to specify the version of the compatible compiler, as in the following example:

```
pragma solidity ^0.8.0
```

This will ensure that the source file does not compile with versions lower than `0.8.0`.

`import` in Solidity allows the importing of symbols from existing Solidity files into the current global scope. This is similar to the `import` statements available in JavaScript, as in the following, for example:

```
import "module-name";
```

Comments can be added to the Solidity source code file in a manner similar to the C language. Multiple-line comments are enclosed in `/*` and `*/`, whereas single-line comments start with `//`.

An example `solidity` program is as follows, showing the use of `pragma`, `import`, and comments:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.0;
contract valueChecker
{
    uint8 price=10;
    //price variable declared and initialized with a value of 10
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool y)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            y= true;
        }
    }
}
```

> Note that an SPDX license identifier needs to be added as well as the first line, otherwise the Solidity compiler will generate a warning. If all is in order, the program will compile successfully.

In this section, we examined what the Solidity code of a smart contract looks like. Now it's time to learn about the Solidity language.

Functions

Functions are pieces of code within a smart contract. For example, look at the following code block:

```
pragma solidity >8.0.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

In the preceding code example, with `contract Test1`, we have defined a function called `addition1()`, which returns an unsigned integer after adding `2` to the value supplied via the variable `x`, initialized just before the function.

In this case, `2` is supplied via variable `x`, and the function will return `4` by adding `2` to the value of `x`. It is a simple function, but demonstrates how functions work and what their different elements are.

There are two function types – i*nternal* and *external* functions:

- **Internal functions** can be used only within the context of the current contract.
- **External functions** can be called via external function calls.
  A **function** in Solidity can be marked as a constant. Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas. This is the practical implementation of the concept of *call* as discussed in the previous chapter. Functions in Solidity are modules of code that are associated with a contract. Functions are declared with a name, optional parameters, access modifiers, state mutability modifiers, and an optional return type. The syntax of defining a function is shown as follows:

  ```
  function <name of the function>(<parameters>) <visibility specifier> <state mutability mod
  {
      <function body>
  }
  ```

  It contains the following elements:

- **Function signature**: Functions in Solidity are identified by their **signature**, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in Remix IDE. For example, the signature for the `Matcher` function we saw earlier is the first four bytes of the 32-byte Keccak-256 hash of the function:

  ```
  {
      "f9d55e21": "Matcher(uint8)"
  }
  ```

  In this example function, `Matcher` has the signature hash of `f9d55e21`. This information is useful in order to build interfaces.

- **Input parameters of a function**: Input parameters of a function are declared in the form of `<data type> <parameter name>`. This exam-

ple clarifies the concept where `uint x` and `uint y` are input parameters of the `checkValues` function:

```
contract myContract
{
    function checkValues(uint x, uint y)
    {
    }
}
```

- **Output parameters of a function**: Output parameters of a function are declared in the form of `<data type> <parameter name>`. This example shows a simple function returning a `uint` value:

```
contract myContract
{
    function getValue() returns (uint z)
    {
        z=x+y;
    }
}
```

A function can return multiple values, as well as taking multiple inputs. In the preceding example function, `getValue` only returns one value, but a function can return up to 14 values of different data types. The names of the unused return parameters can optionally be omitted. An example of such a function could be:

```
pragma solidity ^0.8.0;
contract Test1
{

 function addition1(uint x, uint y) public pure returns (uint z, uint a)
    {
        z= x+y ;
        a=x+y;
        return (z,a);

    }
}
```

Here, when the code runs, it will take two parameters as input, `x` and `y`, add both, then assign them to `z` and `a`, and finally return `z` and `a`. For example, if we provide `1` and `1` for `x` and `y`, respectively, then when the variables `z` and `a` are returned by the function, both will contain `2` as a result.

- **Internal function calls**: Functions within the context of the current contract can be called internally in a direct manner. These calls are made to call the functions that exist within the same contract. These calls result in simple `JUMP` calls at the EVM bytecode level.
- **External function calls**: External function calls are made via message calls from a contract to another contract. In this case, all function parameters are copied to the memory. If a call to an internal function is made using the `this` keyword, it is also considered an external call. The `this` variable is a pointer that refers to the current contract. It is explicitly convertible to an address and all members of a contract are inherited from the address.

- **Fallback functions**: This is an unnamed function in a contract with no arguments and return data. This function executes every time ether is received. It is required to be implemented within a contract if the contract is intended to receive ether; otherwise, an exception will be thrown and ether will be returned. This function also executes if no other function signatures match in the contract. If the contract is expected to receive ether, then the fallback function should be declared with the payable **modifier**.
- The payable is required; otherwise, this function will not be able to receive any ether. This function can be called using the `address.call()` method as, for example, in the following:

```
function ()
{
    throw;
}
```

  In this case, if the fallback function is called according to the conditions described earlier; it will call `throw`, which will roll back the state to what it was before making the call. It can also be some other construct than `throw`; for example, it can log an event that can be used as an alert to feed back the outcome of the call to the calling application.

- **Modifier functions**: These functions are used to change the behavior of a function and can be called before other functions. Usually, they are used to check some conditions or verification before executing the function. `_` (underscore) is used in the modifier functions that will be replaced with the actual body of the function when the modifier is called. Basically, it symbolizes the function that needs to be *guarded*. This concept is similar to *guard* functions in other languages.
- **Constructor function**: This is an optional function that has the same name as the contract and is executed once a contract is created. Constructor functions cannot be called later on by users, and there is only one constructor allowed in a contract. This implies that no overloading functionality is available.
- **Function visibility specifiers** (access modifiers/access levels): Functions can be defined with four access specifiers as follows:
  - **External**: These functions are accessible from other contracts and transactions. They cannot be called internally unless the `this` keyword is used.
  - **Public**: By default, functions are public. They can be called either internally or by using messages.
  - **Internal**: Internal functions are visible to other derived contracts from the parent contract.
  - **Private**: Private functions are only visible to the same contract they are declared in.
- **Function modifiers**:
  - **pure**: This modifier prohibits access or modification to state.
  - **view**: This modifier disables any modification to state.
  - **payable**: This modifier allows payment of ether with a call.
  - **virtual**: This allows the function's or modifier's behavior to be changed in the derived contracts.
  - **override**: This states that this function, modifier, or public state variable changes the behavior of a function or modifier in a base contract.

We can see these elements in an example shown below:

```
function orderMatcher (uint x)
private view returns (bool return value)
```

In the preceding code example, `function` is the keyword used to declare the function. `orderMatcher` is the function name, `uint x` is an optional parameter, `private` is the **access modifier** or **specifier** that controls access to the function from external contracts, `view` is an optional keyword used to specify that this function does not change anything in the contract but is used only to retrieve values from the contract, and `returns (bool return value)` is the optional return type of the function.

Variables

Just like any programming language, variables in Solidity are the named memory locations that hold values in a program. There are three types of variables in Solidity: **local variables**, **global variables**, and **state variables**.

Local variables

These variables have a scope limited to only within the function they are declared in. In other words, their values are present only during the execution of the function in which they are declared.

Global variables

These variables are available globally as they exist in the global namespace. They are used to perform various functions such as ABI encoding, cryptographic functions, and querying blockchain and transaction information.

Solidity provides a number of **global variables** that are always available in the global namespace. These variables provide information about blocks and transactions. Additionally, cryptographic functions, ABI encoding/decoding, and address-related variables are available.

A subset of available global variables is shown as follows.

- This returns the current block number:

```
block.number
```

- This returns the gas price of the transaction:

```
tx.gasprice (uint)
```

- This variable returns the address of the miner of the current block:

```
block.coinbase (address payable)
```

- This returns the current block timestamp:

```
now (uint)
```

- This returns the current block difficulty:

```
block.difficulty (uint)
```

A subset of some functions is shown as follows:

- The function below is used to compute the Keccak-256 hash of the argument provided to the function:

```
keccak256(...) returns (bytes32)
```

- This function returns the associated address of the public key from the elliptic curve signature:

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
```

There are several other global variables available. A comprehensive list and details can be found in Solidity's official documentation: **https://solidity.readthedocs.io/en/latest/units-and-global-variables.html**.

State variables

State variables have their values permanently stored in smart contract storage. State variables are declared outside the body of a function, and they remain available throughout the contract depending on the accessibility assigned to them and as long as the contract persists:

```
pragma solidity >=0.8.0;
contract Addition {
    uint x; // State variable
}
```

Here, `x` is a state variable whose value will be stored in contract storage.

There are three types of state variables, based on their visibility scope:

- **Private**: These variables are only accessible internally from within the contract that they are originally defined in. They are also not accessible from any derived contract from the original contract.
- **Public**: These variables are part of the contract interface. In simple words, anyone is able to get the value of these variables. They are accessible within the contract internally by using the `this` keyword. They can also be called from other contracts and transactions. A `getter` function is automatically created for all public variables.
- **Internal**: These variables are only accessible internally within the contract that they are defined in. In contrast to private state variables, they are also accessible from any derived contract from the original (parent) contract.

There are two modifiers for state variables:

- `constant` : This disallows assignment, except initialization.
- `immutable` : This allows exactly one assignment at construction time and is constant afterward.

In the next section, we will introduce the data types supported in Solidity.

Data types

Solidity has two categories of data types – v**alue types** and **reference types**:

- Value types are variables that are always passed by a value. This means that value types hold their value or data directly, allowing a variable's value held in memory to be directly accessible by accessing the variable.
- Reference types store the address of the memory location where the value is stored. This is in contrast with value types, which store the actual value of a variable directly with the variable itself.

Recall from *Chapter 9*, *Ethereum Architecture*, that EVM can read and write data in different locations. The specific location used for storing values of a variable depends on the data type of the variable and where the variable has been declared. For example, function parameter variables are stored in memory, whereas state variables are stored in storage.

Now we'll describe value types in detail.

Value types

Value types mainly include **Booleans**, **integers**, **addresses**, and **literals**, which are explained in detail here.

Boolean

This data type has two possible values, `true` or `false` , for example:

```
bool v = true;
bool v = false;
```

This statement assigns the value `true` or `false` to `v` depending on the assignment.

Integers

This data type represents integers. Various keywords are used to declare integer data types:

- `int` : Signed integer. `int8` to `int256` , which means that keywords are available from `int8` up to `int256` in increments of 8, for example, `int8` , `int16` , and `int24` .
- `uint` : Unsigned integer. `uint8` , `uint16` , up to `uint256` . Usage is dependent on how many bits are required to be stored in the variable.

For example, in this code, note that `uint` is an alias for `uint256` :

```
uint256 x;
uint y;
```

```
uint256 z;
```

These types can also be declared with the `constant` keyword, which means that no storage slot will be reserved by the compiler for these variables. In this case, each occurrence will be replaced with the actual value:

```
uint constant z=10+10;
```

Address

This data type holds a 160-bit long (20-byte) value. This type has several members that can be used to interact with and query the contracts. These members are described here:

- **Balance**: The `balance` member returns the balance of the address in Wei.
- **Send**: This member is used to send an amount of ether to an address (Ethereum's 160-bit address) and returns `true` or `false` depending on the result of the transaction, for example, the following:

```
address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;
address from = this;
if (to.balance < 10 && from.balance > 50) to.send(20);
```

- **Call functions**: The `call`, `callcode`, and `delegatecall` functions are provided in order to interact with functions that do not have an ABI. These functions should be used with caution as they are not safe to use due to the impact on the type safety and security of the contracts.
- **Array value types (fixed-size and dynamically sized byte arrays)**: Solidity has fixed-size and dynamically sized byte arrays. Fixed-size keywords range from `bytes1` to `bytes32`, whereas dynamically sized keywords include `bytes` and `string`. The `bytes` keyword is used for raw byte data, and `string` is used for strings encoded in UTF-8. As these arrays are returned by the value, calling them will incur a gas cost.
  An example of a static (fixed-size) array is as follows:

```
bytes32[10] bankAccounts;
```

  An example of a dynamically sized array is as follows:

```
bytes32[] trades;
```

  `length` is a member of array value types and returns the length of the byte array:

```
trades.length;
```

Literals

These are used to represent a fixed value. There are different types of literals that are described as follows:

- **Integer literals**: These are a sequence of decimal numbers in the range of 0–9. An example is shown as follows:

```
uint8 x = 2;
```

- **String literals**: This type specifies a set of characters written with double or single quotes. An example is shown as follows:

```
'packt'
"packt"
```

- **Hexadecimal literals**: These are prefixed with the keyword `hex` and specified within double or single quotation marks. An example is shown as follows:

```
(hex'AABBCC');
```

- **Enums**: This allows the creation of user-defined types. An example is shown as follows:

```
enum Order {Filled, Placed, Expired };
Order private ord;
ord=Order.Filled;
```

Explicit conversion to and from all integer types is allowed with enums.

Reference types

As the name suggests, these types are passed by reference and are discussed in the following section. These are also known as **complex types**. Reference types include **arrays, structs**, and **mappings**.

When using reference types, it is essential to explicitly specify the storage area where the type is stored, for example, *memory*, *storage*, or *calldata*.

Arrays

Arrays represent a contiguous set of elements of the same size and type laid out at a memory location. The concept is the same as any other programming language. Arrays have two members, named `length` and `push`.

Structs

These constructs can be used to group a set of dissimilar data types under a logical group. These can be used to define new custom types, as shown in the following example:

```
pragma solidity ^0.8.0;
contract TestStruct {
  struct Trade
  {
    uint tradeid;
    uint quantity;
    uint price;
    string trader;
```

```
    }
    //This struct can be initialized and used as below
    Trade tStruct = Trade({tradeid:123, quantity:1, price:1, trader:"equinox"});
}
```

In the preceding code, we declared a `struct` named `Trade` that has four fields. `tradeid`, `quantity`, and `price` are of the `uint` type, whereas `trader` is of the `string` type. Once the `struct` is declared, we can initialize and use it. We initialize it by using `Trade tStruct` and assigning `123` to `tradeid`, `1` to `quantity`, and `"equinox"` to `trader`.

Sometimes it is desirable to choose the location of the variable data storage. This choice allows for better gas expense management. We can use the data location name to specify where a particular complex data type will be stored. Depending on the default or specified annotation, the location can be `storage`, `memory`, or `calldata`. This is applicable to arrays and structs and can be specified using the `storage` or `memory` keyword. `calldata` behaves almost like memory. It is an unmodifiable and temporary area that can be used to store function arguments.

For example, in the preceding structs example, if we want to use only memory (temporarily) we can do that by using the `memory` keyword when using the structure and assigning values to fields in the `struct`, as shown here:

```
Trade memory tStruct;
tStruct.tradeid = 123;
```

As copying between memory and storage can be quite expensive, specifying a location can be helpful to control the gas expenditure at times.

Parameters of external functions use **calldata** memory. By default, parameters of functions are stored in **memory**, whereas all other local variables make use of **storage**. State variables, on the other hand, are required to use storage.

Mappings

Mappings are used for key-to-value mapping. This is a way to associate a value with a key. All values in this map are already initialized with all zeroes, as in the following example:

```
mapping (address => uint) offers;
```

This example shows that `offers` is declared as a mapping. Another example makes this clearer:

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

This is basically a dictionary or a hash table, where string values are mapped to integer values. The mapping named `bids` has the string `packt` mapped to value `10`.

Control structures

The control structures available in the Solidity language are `if...else`, `do`, `while`, `for`, `break`, `continue`, and `return`. They work exactly the same as other languages, such as the C language or JavaScript.

Some examples are shown here:

- `if`: If `x` is equal to `0`, then assign value `0` to `y`, else assign `1` to `z`:

```
if (x == 0)
    y = 0;
else
    z = 1;
```

- `do`: Increment `x` while `z` is greater than `1`:

```
do{
    x++;
} (while z>1);
```

- `while`: Increment `z` while `x` is greater than `0`:

```
while(x > 0){
    z++;
}
```

- `for`, `break`, and `continue`: Perform some work until `x` is less than or equal to `10`. This `for` loop will run `10` times; if `z` is `5`, then break the `for` loop:

```
for(uint8 x=0; x<=10; x++)
{
    //perform some work
    z++
    if(z == 5) break;
}
```

- `continue` can be used in situations where we want to start the next iteration of the loop immediately without executing the rest of the code. For example, take the code shown next:

```
        for (uint i = 0; i < 5; i++) {
             if (i == 2) {
                 // Skip to next
                 continue;

             }

             if (i == 5) {
                 // Exit loop
                 break;

             }
         }
```

It will continue the work in a similar vein, but when the condition is met, the loop will start again.

- `return` : `return` is used to stop the execution of a function and returns an optional value. For example:

```
return 0;
```

It will stop the execution and return a value of `0`.

Events

Events in Solidity can be used to log certain events in EVM logs. These are quite useful when external interfaces are required to be notified of any change or event in the contract. These logs are stored on the blockchain in transaction logs. Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract.

In a simple example here, the `valueEvent` event will return `true` if the `x` parameter passed to the function `Matcher` is equal to or greater than `10`:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.0;
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool y)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            y = true;
        }
    }
}
```

Inheritance

Inheritance is supported in Solidity. The `is` keyword is used to derive a contract from another contract. In the following example, `valueChecker2` is derived from the `valueChecker` contract. The derived contract has access to all non-private members of the parent contract:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.0;
contract valueChecker
{
    uint8 price = 20;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool y)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            y = true;
        }
    }
}
```

```
contract valueChecker2 is valueChecker
{
    function Matcher2() public view returns (uint)
    {
        return price+10;
    }
}
```

In the preceding example, if the `uint8 price = 20` is changed to `uint8 private price = 20`, then it will not be accessible by the `valueChecker2` contract. This is because now the member is declared as `private`, and thus it is not allowed to be accessed by any other contract. The error message that you will see when attempting to compile this contract is as follows:

```
browser/valuechecker.sol:20:8: DeclarationError: Undeclared identifier.
return price+10;
       ^---^
```

Libraries

Libraries are deployed only once at a specific address and their code is called via the `CALLCODE` or `DELEGATECALL` opcode of the EVM. The key idea behind libraries is code reusability. They are similar to contracts and act as base contracts to the calling contracts.

A library can be declared as shown in the following example:

```
library Addition
{
    function Add(uint x,uint y) returns (uint z)
    {
        return x + y;
    }
}
```

This library can then be called in the contract, as shown here. First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive ether either; this is in contrast to contracts, which can receive ether.

Now let's consider how Solidity approaches handling errors.

Error handling

Solidity provides various functions for error handling. By default, in Solidity, whenever an error occurs, the state does not change and reverts back to the original state.

Some constructs and convenience functions that are available for error handling in Solidity are introduced as follows:

- `assert` : This is used to check for conditions and throw an exception if the condition is not met. Assert is intended to be used for internal errors and invariant checking. When called, this method results in an invalid opcode and any changes in the state are reverted.
- `require` : Similar to `assert` , this is used to check conditions and throws an exception if the condition is not met. The difference is that `require` is used to validate inputs, return values, or calls to external contracts. The method also results in reverting to the original state. It can also take an optional parameter to provide a custom error message.
- `revert` : This method aborts the execution and reverts the current call. It can also optionally take a parameter to return a custom error message to the caller.
- `try/catch` : This construct is used to handle a failure in an external call.
- `throw` : `throw` is used to stop execution. As a result, all state changes are reverted. In this case, no gas is returned to the transaction originator because all the remaining gas is consumed.

This completes a brief introduction to the Solidity language. The language is very rich and under constant improvement. Detailed documentation and coding guidelines are available online at `http://solidity.readthedocs.io/en/latest/`.

Summary

This chapter started with the introduction of development tools for Ethereum, such as Ganache CLI. The installation of Node.js was also introduced, as most of the tools are JavaScript- and Node.js-based. Then we discussed some frameworks such as Truffle, along with local blockchain solutions for development and testing, such as Ganache and Drizzle. We also introduced Solidity in this chapter and explored different concepts, such as value types, reference types, functions, and error handling concepts. We also learned how to write contracts using Solidity.

In the next chapter, we will explore the topic of Web3, a JavaScript API that is used to communicate with the Ethereum blockchain.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://packt.link/ips2H