

Solidity Programming Essentials

Second Edition

A guide to building smart contracts and tokens using
the widely used Solidity language



Ritesh Modi



Solidity Programming Essentials

Second Edition

A guide to building smart contracts and tokens using
the widely used Solidity language

Ritesh Modi



BIRMINGHAM—MUMBAI

Solidity Programming Essentials

Second Edition

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Associate Group Product Manager: Richa Tripathi

Publishing Product Manager: Gebin George

Senior Editor: Rohit Singh

Content Development Editor: Kinnari Chohan

Technical Editor: Maran Fernandes

Copy Editor: Safis Editing

Project Coordinator: Manisha Singh

Proofreader: Safis Editing

Indexer: Subalakshmi Govindan

Production Designer: Shankar Kalbhor

Marketing Coordinator: Sonakshi Bubbar

First published: April 2018

Second edition: May 2022

Production reference: 1200522

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80323-118-1

www.packtpub.com

Contributors

About the author

Ritesh Modi is a technologist with more than 18 years of experience. He holds a master's degree in science in AI/ML from LJMU. He has been recognized as a Microsoft Regional Director for his contributions to building tech communities, products, and services. He has published more than 10 tech books in the past and is a cloud architect, speaker, and leader who is popular for his contributions to data centers, Azure, Kubernetes, blockchain, cognitive services, DevOps, AI, and automation.

"I want to thank the people who have been close to me and supported me, especially my wife, Sangeeta, my daughter, Avni, and my parents."

About the reviewer

Kevin Bluer is a lead blockchain engineer at ConsenSys, focusing on open source development tooling such as Truffle and Ganache. Before Web3, he had over 15 years of experience in software development, working with companies across the industrial spectrum, including Microsoft, JPMorgan Chase, and Viacom. He is also an experienced trainer and mentor, having worked with thousands of developers around the globe. Kevin is also an active investor, photographer, and runner.

Table of Contents

Preface

Part 1: The Fundamentals of Solidity and Ethereum

1

An Introduction to Blockchain, Ethereum, and Smart Contracts

Technical requirements	4	Mining nodes	17
What is a blockchain?	5	Ethereum validators	17
The need for blockchain	6	Ethereum accounts	17
Understanding cryptography	7	Externally owned accounts	17
Hashing	7	Contract accounts	18
Digital signatures	8	Ether, gas, and transactions	18
Reviewing blockchain and Ethereum architecture	9	Blocks	23
Relationship between blocks	10	An end-to-end transaction	25
How transactions and blocks are related to each other	12	Smart contract	26
		Writing smart contracts	26
Consensus	13	The internals of smart contract deployment	32
Proof of work	13	Summary	33
Proof of stake	16	Questions	33
Ethereum nodes	16	Further reading	33
EVM	16		

2

Installing Ethereum and Solidity

Technical requirements	36	Creating a private network	42
Ethereum networks	36	Installing Ganache	46
Main network	37	Installing the Solidity compiler	50
Test network	37	Installing the web3 framework	51
Private network	38	Installing and using MetaMask	52
Consortium network	38	Summary	59
Installing and configuring Geth	38	Questions	59
Installing Geth on macOS	39	Further reading	59
Installing Geth on Windows	39		

3

Introducing Solidity

Technical requirements	62	Storage and memory data locations	79
The Ethereum Virtual Machine	62	Rule one	79
Understanding Solidity and Solidity files	63	Rule two	79
Pragma	63	Rule three	80
Comments	64	Rule four	80
Importing Solidity code	65	Rule five	80
Contracts	66	Rule six	82
The structure of a contract	67	Rule seven	83
State variables	69	Rule eight	84
Structure	70	Using literals	85
Modifiers	71	Understanding integers	86
Events	73	Understanding Boolean	87
Enumeration	74	The byte data type	88
Functions	74	Understanding arrays	90
Exploring data types in Solidity	76	Fixed arrays	91
Value types	76	Dynamic arrays	91
Reference types	77	Special arrays	93
		Array properties	95

Knowing more about the structure of an array	95	Working with mappings	100
Enumerations	97	Summary	104
Understanding the address data type	99	Questions	105
		Further reading	105

4

Global Variables and Functions

Technical requirements	108	The difference between tx.origin and msg.sender	114
Variable scoping	108	Cryptographic global variables	115
Type conversion	109	Address global variables	116
Implicit conversion	110	Contract global variables	116
Explicit conversion	110	Recovering addresses using ecrecover	117
Block and transaction global variables	112	Summary	122
Transaction- and message-related global variables	114	Questions	123
		Further reading	123

5

Expressions and Control Structures

Technical requirements	126	Understanding the do...while loop	131
Understanding Solidity expressions	126	Understanding breaks	134
Understanding the if and if... else decision control	128	Understanding continue	135
Exploring while loops	129	Understanding return	136
		Summary	137
		Questions	137
		Further reading	137

Part 2: Writing Robust Smart Contracts

6

Writing Smart Contracts

Technical requirements	142	Polymorphism	156
Smart contracts	142	Function polymorphism	156
Writing a smart contract	143	Contract polymorphism	157
Creating contracts	144	Method overriding	158
Using the new keyword	144	Abstract contracts	159
Using the address of a contract	147	Interfaces	160
Contract constructor	148	Advanced interfaces	162
Contract composition	150	Library	164
Inheritance	150	Importing a library	165
Single inheritance	150	Summary	166
Multilevel inheritance	152	Questions	167
Hierarchical inheritance	152	Further reading	167
Multiple inheritance	153		
Encapsulation	155		

7

Solidity Functions, Modifiers, and Fallbacks

Technical requirements	170	The address call method	181
Function input and output	170	The address callcode method	185
Modifiers	172	The address delegatecall method	185
Visibility scope	175	The address staticcall method	186
View, constant, and pure functions	176	The fallback and receive functions	187
Address-related functions	179	Summary	191
The address send method	179	Questions	191
The address transfer method	181	Further reading	192

8

Exceptions, Events, and Logging

Technical requirements	194	Try-catch in Solidity	209
Exception handling	194	Events and logging	214
Require	197	Summary	218
Assert	202	Questions	219
Revert	204	Further reading	219

9

Basics of Truffle and Unit Testing

Technical requirements	222	Interactively working with Truffle	232
Application development life cycle management	222	Summary	234
Introducing Truffle	223	Questions	235
Development with Truffle	225	Further reading	235
Testing with Truffle	230		

10

Debugging Contracts

Technical requirements	238	Using a block explorer	244
Overview of debugging	238	Summary	247
The Remix editor	238	Questions	247
Using events	244	Further reading	247

Part 3: Advanced Smart Contracts

11

Assembly Programming

Technical requirements	252	Working with storage slots	261
An introduction to Solidity and its advantages	252	Calling contract functions	263
Getting started with Assembly programming	253	Determining contract addresses	265
Scopes and blocks	256	Summary	266
Returning values	257	Questions	267
Working with memory slots	258	Further reading	267

12

Upgradable Smart Contracts

Technical requirements	270	Implementing simple solutions with inheritance	277
Learning what constitutes upgradability	270	Implementing simple solutions with composition	281
Understanding dependency injection	272	Implementing advanced solutions using proxy contracts	285
Providing instance addresses during contract deployment	273	Writing upgradable contracts with upgradable storage	290
Providing instance addresses following contract deployment	274	Summary	292
Reviewing problematic smart contracts	275	Questions	292
		Further reading	292

13

Writing Secure Contracts

Technical requirements	294	Reentrancy attack	296
SafeMath and under/overflow attacks	294	The EtherPot contract	297
		The Hacker contract	299

Solutions to the reentrancy problem	300	Summary	305
Security best practices	304	Questions	306
		Further reading	306

14

Writing Token Contracts

Technical requirements	308	The ERC721 implementation	323
Introducing tokens	308	EIP223	330
ERC20 Tokens	309	ERC165	334
ERC20 standard	310	Summary	337
ERC20 functionality	310	Questions	338
ERC20 events	312	Further reading	338
Non-fungible tokens	320		
ERC721	322		

15

Solidity Design Patterns

Technical requirements	340	Data cohesion	349
Introducing entity modeling	340	Out-of-bounds nested structure	349
Ethereum storage	341	Static data within a nested structure	349
Data types in Ethereum	341	The nested structure will not change in the near future	349
Understanding data modeling in Solidity	343	Containment relationship	349
Nested versus reference fields	343	Having few relationships	350
Exploring types of relationships	346	Performing data modeling using an example	350
One-to-one relationships	347	Structures	350
One-to-many relationships	347	State variables	351
Many-to-many relationships	348	Adding Employees	351
Reviewing the rules for embedding structures	348	Retrieving a single Employees record	352
		Updating Employees	352
		Retrieving all Employees	353

Ownership in smart contracts	354	Transfer of the ownership of assets within a smart contract	364
Exploring ownership in Solidity	354		
Modifier	356	Stoppable/haltable smart contract pattern	366
Establishing ownership of a smart contract	357	Summary	368
Multiownership	359	Questions	369
Transfer of ownership	361	Further reading	369
MultiSig contracts	363		

Assessments

Index

Other Books You May Enjoy

Preface

I am not sure when there was last so much of a discussion about a single technology across governments, organizations, communities, and individuals. Blockchain is a technology that is being discussed and debated at length across the world and in many organizations, and with reason. Blockchain is not a technology that has a limited effect on our lives. It has and will have widespread ramifications in our lives. The day is not far off when blockchain will touch almost every aspect of our daily activities—paying bills, making transactions with any organization, receiving a salary, verifying our identity, receiving educational results, and so on. This is just the beginning, and we have just started to understand the meaning of decentralization and its impact.

I have been working on blockchain for quite some time now and have been a crypto-investor for a while. I am a technologist and am completely fascinated by Bitcoin because of the architectural marvel it is. I have never come across such a superior thought process and architecture that actually solves not only economic and social problems but also some technically unsolved problems, such as Byzantine general problems and fault tolerance. It solves the problem of distributed computing at large.

Ethereum is built in a similar fashion, and I was in awe when I first heard about and experienced smart contracts. Smart contracts are one of the greatest innovations to deploy decentralized applications on blockchain and extend it easily with custom logic, policies, and rules.

I have thoroughly enjoyed writing this book and sincerely hope that you also enjoy reading about and implementing Solidity. I have brought in a lot of my Solidity experience and try to make the most of it. I hope this book makes you a better Solidity developer and a superior programmer.

Do let me know if there is anything I can do to make your experience better with this book. I am all ears. Happy learning!

Who this book is for

This book is primarily aimed at beginners who would like to get started with Solidity programming to develop an Ethereum smart contract. No prior knowledge of the EVM is required, but knowing the basics of any programming language will help you follow along.

What this book covers

Chapter 1, Introduction to Blockchain, Ethereum, and Smart Contracts, takes you through the fundamentals of blockchain, its terminology and jargon, its advantages, the problems it's trying to solve, and its industry relevance. It will explain the important concepts and architecture in detail. This chapter will also teach you about concepts specific to Ethereum, such as externally owned accounts, contract accounts, and its currency in terms of gas and Ether. Ethereum is heavily based on cryptography, so you'll also learn about hash, encryption, and the usage of keys for creating transactions and accounts. How transactions and accounts are created, how gas is paid for each transaction, the difference between message calls and transactions, and the storage of code and state management will be explained in detail.

Chapter 2, Installing Ethereum and Solidity, takes you through creating a private blockchain using the Ethereum platform. It will provide step-by-step guidance for creating a private chain. Another important tool in the Ethereum ecosystem is Ganache, which is mainly used for development and testing purposes. This chapter will also show the process of installing Ganache and using it for developing, testing, and deploying Solidity contracts. You will also install MetaMask, which is a wallet and can interact with any kind of Ethereum network. MetaMask is used to create new accounts, interact with contracts, and use them. The mining of transactions will also be shown in this chapter. Remix is a great tool for authoring Solidity contracts, shown toward the end of the chapter.

Chapter 3, Introducing Solidity, begins the Solidity journey. In this chapter, you'll learn the basics of Solidity by understanding its different versions and how to use a version using pragma. Another important aspect of this chapter is understanding the big picture of authoring smart contracts. Smart contract layout will be discussed in depth using important constructs such as state variables, functions, constant functions, events, modifiers, fallbacks, enums, and structs. This chapter discusses and implements the most important element of any programming language—data types and variables. Data types can be simple or complex; there can be value and reference types and storage and memory types—all these types of variables will also be shown using examples.

Chapter 4, Global Variables and Functions, provides implementation and usage details of block- and transaction-related global functions and variables and address- and contract-related global functions and variables. These come in very handy in writing any series of smart contract development. Recovering public addresses using `ecrecover` is explained as well in depth in this chapter.

Chapter 5, Expressions and Control Structures, teaches you how to write contracts and functions that have conditional logic using `if...else` and `switch` statements. Looping is an important part of any language and Solidity provides `while` and `for` loops for looping over arrays. Examples and implementation of looping will be part of this chapter. Loops must break based on certain conditions and should continue based on other conditions.

Chapter 6, Writing Smart Contracts, is the core chapter of the book. Here, you will start writing serious smart contracts. We will discuss the design aspects of writing smart contracts, defining and implementing a contract, and deploying and creating contracts using different mechanisms such as the `new` keyword and known addresses. Solidity provides rich object orientation, and this chapter will delve deep into object-oriented concepts and implementation, such as inheritance, multiple inheritance, declaring abstract classes and interfaces, and providing method implementations to abstract functions and interfaces. Advanced interface reuse when only the address is available is also covered. Reusable contracts, also known as libraries in Solidity, are explained using simple examples.

Chapter 7, Functions, Modifiers, and Fallbacks, shows how to implement basic functions that accept inputs and return outputs, functions that just output the existing state without changing the state and modifiers. Modifiers help in organizing code better in Solidity. They help with security and reusing code within contracts. Fallbacks are important constructs and are executed when a function call does not match any of the existing function signatures. Fallbacks are also important for transferring Ether to contracts. Both modifiers and fallbacks will be discussed and implemented with examples for easy understanding. Receive functions are relatively new in Solidity but sample coverage is provided on them in this chapter.

Chapter 8, Exceptions, Events, and Logging, covers exceptions, events, and logging, which are important in Solidity from a contract development perspective. Ether should be returned to the caller in the case of an error and exception. Exception handling will be explained and implemented in depth in this chapter using newer Solidity constructs, such as `assert`, `require`, and `revert`. The `try-catch` blocks introduced recently in Solidity are covered using multiple examples in this chapter. Events and logging help in understanding the execution of contracts and functions. This chapter will show and explain the implementation of both events and logs.

Chapter 9, Truffle Basics and Unit Testing, covers the basics of Truffle, including understanding its concepts, creating a project and understanding its project structure, modifying its configuration, and taking a sample contract through the entire life cycle of writing, testing, deploying, and migrating a contract. Testing is as important for contracts as writing a contract. Truffle helps in providing a framework to test; however, tests should be written. This chapter will discuss the basics of unit tests, writing unit tests using Solidity, and executing them against the smart contract. Unit tests will be executed by creating transactions and validating their results. This chapter will show implementation details for writing and executing unit tests for a sample contract. Interactively working with contracts using Truffle is also shown in this chapter.

Chapter 10, Debugging Contracts, covers troubleshooting and debugging using multiple tools, such as Remix and events. This chapter will show how to execute code line by line, checking the state after every line of code and changing contract code accordingly.

Chapter 11, Assembly Programming, goes into a slightly more complex topic and explores assembly programming in Solidity. Solidity has its own assembly programming and this chapter covers it from the ground up, from working with variables, scopes, and blocks and returning values to advanced topics including working with memory and state slots and calling contract functions. This chapter also includes assembly examples related to determining whether an address is a contract address or not.

Chapter 12, Upgradable Smart Contracts, is an important chapter from a contract maintainability perspective. This chapter will take you on a journey from a non-upgradable contract to writing upgradable contracts using different strategies. Some of these strategies include upgradability using inheritance, composition, proxy contracts, and upgradable storage.

Chapter 13, Writing Secure Contracts, brings security into perspective related to smart contracts. It starts by listing some of the security best practices and then moves on to showing some of the most prevalent vulnerabilities with smart contracts and how best to avoid them. These include the complete implementation of reentrancy attacks on smart contracts and overflow/underflow attacks.

Chapter 14, Writing Token Contracts, is about building and implementing both ERC20 and ERC721 tokens. Fungible and **Non-Fungible Tokens (NFTs)** are explained with the help of complete implementation in this chapter. Implementation of some of the ancillary interfaces, such as ERC165 and ERC223, is also explained along with their implementations.

Chapter 15, Solidity Design Patterns, describes some of the important design patterns prevalent in Solidity. It starts with the modeling of entities in smart contracts and between contracts using references, relationships, and embedment. It also shows a complete example of performing CRUD operations on these entities. Some of the design patterns shown with implementations in this chapter include ownership, multi-ownership, and multi-sig contracts. We will also discuss haltable or stoppable smart contracts.

To get the most out of this book

Software/hardware covered in the book	Operating system requirements
Solidity compiler (solc) 0.8.13	Windows, macOS, or Linux
web3.js v1.5.3	
Ganache v7.0.3	

Most of the examples shown in this book are executed using the online Remix editor available at <https://remix.ethereum.org>.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781803231181_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Apart from `pragma`, `import`, and comments, we can define contracts, libraries, and interfaces at a global level."

A block of code is set as follows:

```
// This is a single-line comment in Solidity
/*
 * This is a multiline comment
 * In Solidity. Use this when multiple consecutive lines
 * Should be commented as a whole */
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
pragma solidity >=0.7.0 <0.9.0;

contract ConversionDemo {

    function ConversionExplicitUINT8toUINT256() pure public
returns (uint){
    .....
}
```

Any command-line input or output is written as follows:

```
pragma solidity >=0.7.0 <0.9.0;
```

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Solidity Programming Essentials*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1: The Fundamentals of Solidity and Ethereum

In this section, we will learn about the basic building blocks of Solidity programming. In this part, we will first briefly understand Ethereum and its toolsets, and then move on to writing simple contracts and explaining the fundamental concepts related to Solidity data types, variables, conditional statements, `while` and `for` loops, and `return` statements. Solidity provides global variables and functions, which are also explained in this section.

This part contains the following chapters:

- *Chapter 1, Introduction to Blockchain, Ethereum, and Smart Contracts*
- *Chapter 2, Installing Ethereum and Solidity*
- *Chapter 3, Introducing Solidity*
- *Chapter 4, Global Variables and Functions*
- *Chapter 5, Expressions and Control Structure*

1

An Introduction to Blockchain, Ethereum, and Smart Contracts

The last decade has already seen an extraordinary evolution of the technology and computing ecosystem. Technological innovation and its impact have been noticeable across the spectrum, from the **Internet of Things (IoT)** and **Artificial Intelligence (AI)** to blockchains. Each of them has had a disruptive force within multiple industries, and blockchains are one of the most disruptive technologies today – so much so that blockchains have the potential to change almost every industry. Blockchains are revolutionizing almost all industries and domains while bringing forward newer business models. Blockchain is not a new technology; however, it has gained momentum over the last couple of years. It is a big leap forward in terms of thinking about decentralized and distributed applications. It is about the current architectural landscape and strategies for moving toward immutable distributed databases.

In this chapter, you will quickly learn and understand the basic and foundational concepts of blockchains and Ethereum. We will also discuss some of the important concepts that make blockchains and Ethereum work. Also, we will touch briefly on the topic of smart contracts and how to author them using Solidity.

Please note that this chapter explains important blockchain concepts briefly. It does not explain all concepts in great detail, as that would require a complete book by itself. Since Ethereum is an implementation of a blockchain, both the terms will be used interchangeably in this book.

This chapter will focus on introducing the following topics:

- What a blockchain is and why it is used
- Understanding cryptography
- Blockchain and Ethereum architecture
- Ether, gas, and transactions
- Consensus
- Nodes
- Mining
- Understanding accounts, transactions, and blocks
- Smart contracts

By the end of this chapter, you will know all about the basics of blockchain, Ethereum, and smart contracts, which will prepare the base for the next chapters.

Technical requirements

To execute the instructions given in this chapter, you can use a machine with any operating system (Mac, Windows, or Linux) with a browser installed on it. Code for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter01>.

What is a blockchain?

A blockchain is essentially a decentralized, distributed database or ledger, as follows:

- **Decentralization:** In simple terms, this means that the application or service continues to be available and usable even if a server or a group of servers on a network crashes or is not available. The service or application is deployed on a network in a way that no server has absolute control over data and execution; rather, each server has a current copy of data and execution logic.
- **Distributed:** This means that any server or node on a network is connected to every other node on the network. Rather than having one-to-one or one-to-many connectivity between servers, servers have many-to-many connections with other servers.
- **Database:** This refers to the location for storing durable data that can be accessed at any point in time. A database allows the storage and retrieval of data as functionality and also provides management functionalities to manage data efficiently, such as exporting, importing, backup, and restoration.
- **Ledger:** This is an accounting term. Think of it as specialized storage and retrieval of data. Think of ledgers that are available to banks – for example, when a transaction is executed with a bank. Let's say that Tom deposits USD 100 in his account; the bank enters this information in a ledger as credit. At some point in the future, Tom withdraws USD 25. The bank does not modify the existing entry and stored data from 100 to 75. Instead, it adds another entry in the same ledger as a debit of USD 25. This is because a ledger is a specialized database that does not allow modification of existing data. It allows you to create and append a new transaction to modify the current balance in the ledger. The blockchain is a database that has the same characteristics as a ledger. It allows newer transactions to be stored in an append-only pattern without any scope to modify past transactions. It is important here to understand that existing data can be modified by using a new transaction, but past transactions cannot be modified. A balance of USD 100 can be modified at any time by executing a new debit or credit transaction, but previous transactions cannot be modified. Take a look at the following diagram for a better understanding:

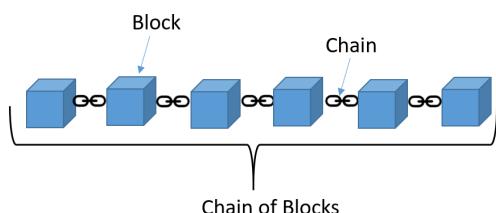


Figure 1.1 – A blockchain is essentially a chain of blocks

Blockchain means a chain of blocks – that is, having multiple blocks chained together, with each block storing transactions in such a way that it is not possible to change these transactions. We will discuss this in later sections when we talk about the storage of transactions and how immutability is achieved in a blockchain.

Because they are decentralized and distributed, blockchain solutions are stable, robust, durable, and highly available. There is no single point of failure. No single node or server is the owner of the data and solution, and everyone participates as a stakeholder.

Not being able to change and modify past transactions makes blockchain solutions highly trustworthy, transparent, and incorruptible.

The need for blockchain

The main objective of blockchain is to accept transactions from accounts, update the current state, and maintain this state till another transaction updates it again. This entire process can be divided into two phases in blockchain. There is a decoupling in between when a transaction is accepted by Ethereum and when the transaction is executed and written to the ledger. This decoupling is quite important for decentralization and distributed architecture to work as expected.

Blockchain helps primarily in the following three different ways:

- **Trust:** Blockchain helps in creating applications that are decentralized and collectively owned by multiple people. Nobody within this group has the power to change or delete previous transactions. Even if someone tries to do so, it will not be accepted by other stakeholders.
- **Autonomy:** There is no single owner for blockchain-based applications. No one controls the blockchain, but everyone participates in its activities. This helps in creating solutions that cannot be manipulated or corrupted.
- **Intermediaries:** Blockchain-based applications can help remove intermediaries from existing processes. Generally, there is a central body, such as vehicle registration and license issuing, that acts as a registrar for registering vehicles as well as issuing driver licenses. Without blockchain-based systems, there is no central body, and if a license is issued or a vehicle is registered after a blockchain mining process, that will remain a fact for an epoch (an epoch is a period of time – say, 5 seconds) without the need of any central authority vouching for it.

Blockchain is heavily dependent on cryptography technologies, as we will discuss in the following section.

Understanding cryptography

Cryptography is the science of converting plain, simple text into secret, hidden, and meaningful text, and vice versa. It also helps in transmitting and storing data that cannot be easily deciphered using owned keys.

There are two types of cryptography in computing:

- **Symmetric cryptography:** This refers to the process of using a single key for both encryption and decryption. It means the same key should be available for multiple people if they want to exchange messages using this form of cryptography.
- **Asymmetric cryptography:** This refers to the process of using two keys for encryption and decryption. Any key can be used to encrypt and decrypt. Messages encrypted with a public key can be decrypted using a private key, and messages encrypted by a private key can be decrypted using a public key. Let's understand this with the help of an example. Tom uses Alice's public key to encrypt messages and sends it to Alice. Alice can use her private key to decrypt the message and extract its content. Messages encrypted with Alice's public key can only be decrypted by Alice, as only she holds her private key and no one else. This is the general use case of asymmetric keys. There is another use that we will see in the *Digital signatures* section.

Hashing

Hashing is the process of transforming any input data into fixed-length random character data, and it is not possible to regenerate or identify the original data from the resultant hash. Hashes are also known as fingerprints of input data. It is next to impossible to derive input data based on its hash value. Hashing ensures that even a slight change in input data will completely change the output data, and no one can ascertain the change in the original data.

Another important property of hashing is that no matter the size of input string data, the length of its output is always fixed. For example, using the SHA-256 hashing algorithm and function with any length of input will always generate 256-bit output data. This can especially become useful when large amounts of data can be stored as 256-bit output data. Ethereum uses the hashing technique quite extensively. It hashes every transaction, hashes the hash of two transactions at a time, and ultimately generates a single root transaction hash for every transaction within a block.

Another important property of hashing is that it is not mathematically feasible to identify two different input strings that will output the same hash. Similarly, it is not possible to find the input computationally and mathematically from the hash itself.

Ethereum uses Keccak256 as its hashing algorithm. The following screenshot shows an example of hashing. The Ritesh Modi input generates a hash, as shown in the following screenshot:

SHA256 Hash

The screenshot shows a user interface for hashing. On the left, there is a label "Data:" followed by a text input field containing the text "Ritesh Modi". On the right, there is a label "Hash:" followed by a text input field containing the hash value "b9fda68f334232a4c832ff355aef9949bf3229cd2f9be8dccf95c8ee1d2c2dbb". The entire interface is contained within a light gray box.

Figure 1.2 – A hashing example using the SHA-256 algorithm

Even a small modification of input generates a completely different hash, as shown in the following screenshot:

SHA256 Hash

The screenshot shows a user interface for hashing. On the left, there is a label "Data:" followed by a text input field containing the text "RiteshModi". On the right, there is a label "Hash:" followed by a text input field containing the hash value "d1571b194cf62f3ffd7601af6777b370b17e9641da8878b2994c18d69317abc4". The entire interface is contained within a light gray box.

Figure 1.3 – A completely different hash output based on a small change in the original input

Digital signatures

Earlier, we discussed cryptography using asymmetric keys. One of the important uses for asymmetric keys is in the creation and verification of a digital signature. Digital signatures are very similar to a signature done by an individual on a piece of paper. Similar to a paper signature, a digital signature helps in identifying an individual. It also helps in ensuring that messages are not tampered with while in transit. Let's understand digital signatures with the help of an example.

Alice wants to send a message to Tom. How can Tom identify and ensure that the message has come from Alice only and that the message has not been changed or tampered with in transit? Instead of sending a raw message/transaction, Alice creates a hash of the entire payload and encrypts the hash with her private key. She appends the resultant digital signature to the hash and transmits it to Tom. When the transaction reaches Tom, he extracts the digital signature and decrypts it using Alice's public key to find the original hash. He also extracts the original hash from the rest of the message and compares both the hashes. If the hashes match, it means that it actually originated from Alice and that it has not been tampered with.

Digital signatures are used to sign transaction data by the owner of the asset or cryptocurrency, such as Ether. With a basic understanding of cryptography, it's time to introduce Ethereum and blockchain at a high level.

Reviewing blockchain and Ethereum architecture

Blockchain is an architecture comprising multiple components, and what makes blockchain unique is the way these components function and interact with each other. Ethereum allows you to extend its functionality with the help of smart contracts. (Smart contracts will be addressed in detail throughout this book.)

Some of the important Ethereum components are the **Ethereum Virtual Machine (EVM)**, miner, block, transaction, consensus algorithm, account, smart contract, mining, Ether, and gas. We are going to discuss each of these components in this chapter.

A blockchain network consists of multiple nodes belonging to miners and some nodes that do not mine but help in the execution of smart contracts and transactions. These are known as EVMs. Each node is connected to another node on the network. These nodes use a peer-to-peer protocol to talk to each other. They, by default, use port 30303 to talk among themselves.

Each miner maintains an instance of a ledger. A ledger contains all blocks in the chain. With multiple miners, it is quite possible that each miner's ledger instance might have different blocks to another. The miners synchronize their blocks on an ongoing basis to ensure that every miner's ledger instance is the same as the other. Details about ledgers, blocks, and transactions are discussed in detail in subsequent sections in this chapter.

The EVM executes smart contracts and helps bring about changes to the global state. Smart contracts help in extending Ethereum by writing custom business functionality into it. These smart contracts can be executed as part of a transaction, and it follows the process of mining as discussed earlier.

A person with an account on a network can send a message for the transfer of Ether from their account to another or can send a message to invoke a function within a contract. Ethereum does not distinguish them as far as transactions are considered. The transaction must be digitally signed with an account holder's private key. This is to ensure that the identity of the sender can be established while verifying the transaction and changing the balances of multiple accounts. Let's take a look at the components of Ethereum in the following diagram:

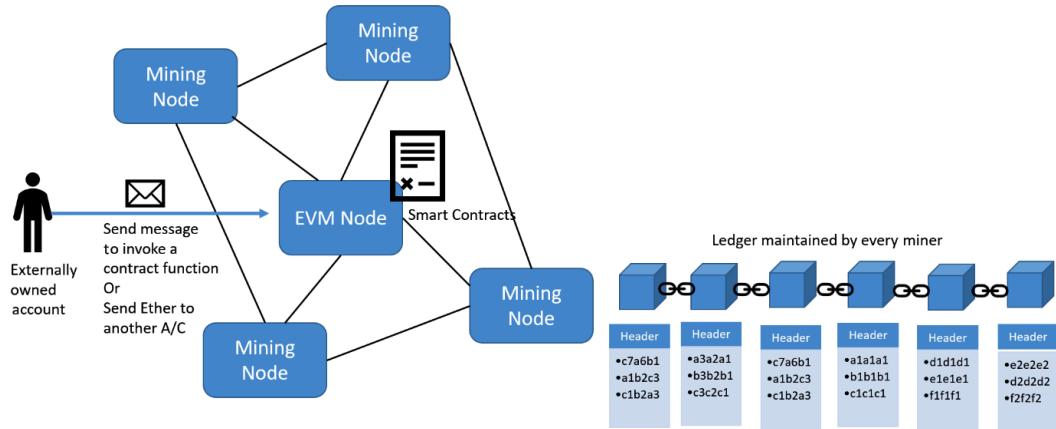


Figure 1.4 – The relationship between blockchain and mining

The previous diagram illustrates some of the important components in Ethereum. The externally owned accounts are responsible for initiating transactions on Ethereum. The transactions that are executed within the Ethereum nodes are finally written as blocks on the blockchain. These blocks have header sections that help in chaining the blocks.

Relationship between blocks

In blockchain and Ethereum, every block is related to another block. There is a parent-child relationship between two blocks. There can be only one child to a parent and a child can have a single parent. This helps in forming a chain in blockchain, as shown. Blocks will be explained in a later section in this chapter:

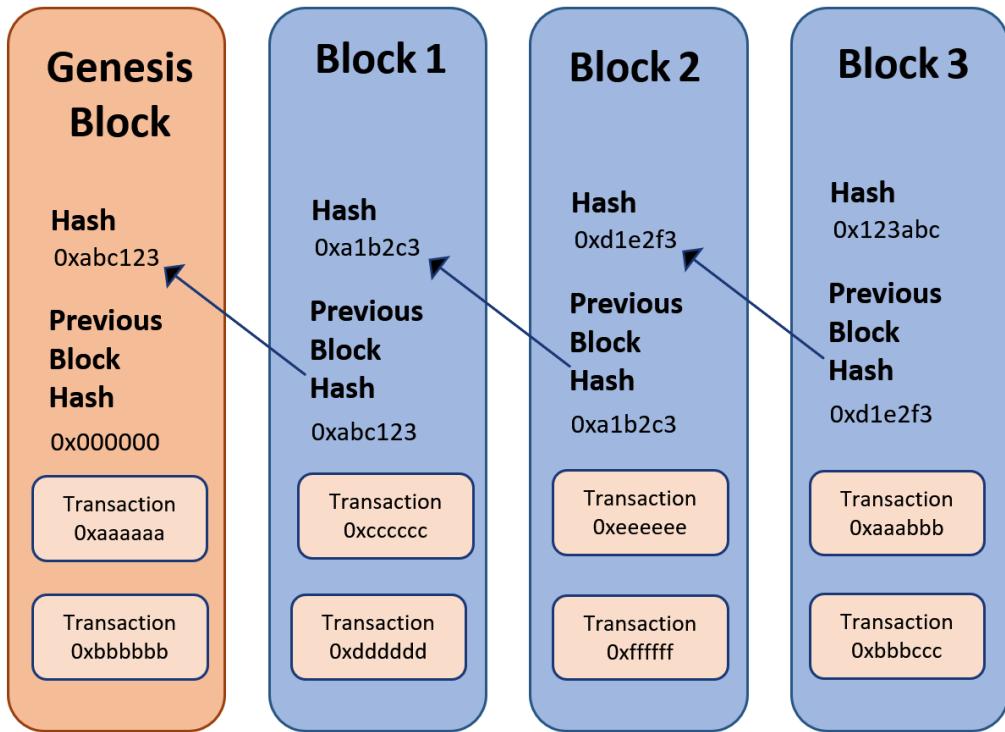


Figure 1.5 – The relationship between blocks using a block hash

In this diagram, we can see three blocks apart from the **Genesis Block** – **Block 1**, **Block 2**, and **Block 3**. **Block 1** is the parent of **Block 2**, and **Block 2** is the parent of **Block 3**. The relationship is established by storing the parent block's hash in a child's block header. **Block 2** stores the hash of **Block 1** in its header and **Block 3** stores the hash of **Block 2** in its header. So, the question arises – who is the parent of the first block? Ethereum has a concept of the genesis block, also known as the **first block**. This block is created automatically when the chain is first initiated. You can say that a chain is initiated with the first block, the genesis block, and the formation of this block is driven through the `genesis.json` file.

The next chapter will show you how to use the `genesis.json` file to create the first block while initializing the blockchain.

How transactions and blocks are related to each other

Now that we know that blocks are related to each other, you will be interested in knowing how transactions are related to blocks. Ethereum stores transactions within blocks. Each block has an upper gas limit, and each transaction needs a certain amount of gas to be consumed as part of its execution. The cumulative gas from all transactions that are not yet written in a ledger cannot surpass the block gas limit. This ensures that all transactions do not get stored within a single block. As soon as the gas limit is reached, other transactions are removed from the block and mining begins thereafter. The gas concept will be covered in a subsequent section in this chapter. This section should be revisited after reading about gas.

The transactions are hashed and stored in the block. The hashes of two transactions are taken and hashed further to generate another hash. This process eventually provides a single hash from all transactions stored within the block. This hash is known as the **transaction Merkle root hash** and is stored in a block's header.

A change in any transaction will result in a change in its hash and, eventually, a change in the root transaction hash. It will have a cumulative effect because the hash of the block will change, and the child block has to change its hash because it stores its parent hash. This helps in making transactions immutable. This is also shown in the following diagram:

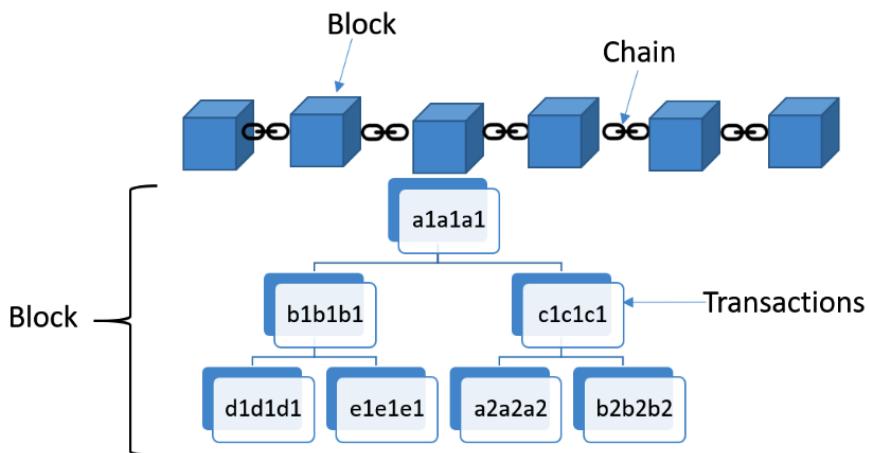


Figure 1.6 – The Merkle root representing all transaction hashes

Blocks and transactions are core to blockchain, but the question remains about the process of adding them to the chain. A generated block should be agreed upon and acceptable to all the nodes within a network. The process of coming to an agreement on a block and subsequently either adding it to the chain or rejecting it is based on the process known as consensus.

Consensus

Consensus is one of the most important concepts in the world of blockchain. As we know, a blockchain comprises multiple nodes that are running the same software and storing the same data. There should be some agreement between the nodes to accept a given state as a known agreeable state. With thousands of nodes running together, intertwined using a **Peer-to-Peer (P2P)** network, they should agree on what is stored historically on the blocks and what should be included as transactions on the new block. It is quite possible that a node goes rogue and adds transactions that are not valid. The consensus mechanism or agreements among nodes would ensure that those transactions are not accepted by other ones. A rogue actor has to minimally control a majority of the nodes to make everyone agree on the dubious transactions.

There are various ways to implement and achieve consensus among nodes. Bitcoin is implemented using the **Proof-of-Work (PoW)** consensus algorithm. Ethereum also implements the PoW consensus algorithm as part of Ethereum 1.0. The new upcoming Ethereum 2.0 is changing its consensus algorithm to the **Proof-of-Stake (PoS)** consensus algorithm.

One of the main drawbacks of the PoW consensus algorithm is that it is quite costly to generate and agree on blocks due to its mining process, which will be explained later. Mining involves heavy usage of electricity and that has negative consequences toward environmental hazards and cost. Both PoW and PoS will be explained in the next subsections.

Proof of work

A miner is responsible for writing transactions to the Ethereum chain. A miner's job is very similar to that of an accountant. An accountant is responsible for writing and maintaining a ledger; similarly, a miner is solely responsible for writing a transaction to an Ethereum ledger. A miner is interested in writing transactions to a ledger because of the reward associated with it. Miners get two types of reward – a reward for writing a block to the chain and cumulative gas fees from all transactions in the block. There are generally many miners available within a blockchain network, each trying and competing to write transactions. However, only one miner can write the block to the ledger, and the rest will not be able to write the current block.

The miner responsible for writing the block is determined by way of a puzzle. The challenge is given to every miner, and they try to solve the puzzle using their computing power. The miner who solves the puzzle first writes the block containing transactions to their own ledger and sends the block and nonce value to other miners for verification. Once verified and accepted, the new block is written to all ledgers belonging to miners. In this process, the winning miner also receives Ether as a reward. Every mining node maintains its own instance of the Ethereum ledger, and the ledger is ultimately the same across all miners. It is the miner's job to ensure that their ledger is updated with the latest blocks. The following are the three important functions performed by miners or mining nodes:

- Mine or create a new block with a transaction and write this to the Ethereum ledger.
- Advertise and send a newly mined block to other miners.
- Accept new blocks mined by other miners and keep their own ledger instance up to date.

Mining nodes refer to nodes that belong to miners. These nodes are part of the same network where the EVM is hosted. At some point in time, miners will create a new block, collect all transactions from the transaction pool, and add them to the newly created block. Finally, this block is added to the chain. There are additional concepts such as consensus and the solving of a target puzzle before writing the block, which will be explained in the following section.

Miners are always looking forward to mining new blocks and are also listening actively to receive new blocks from other miners. They are listening for new transactions stored in the transaction pool. Miners also broadcast the new transactions to other connected nodes after validation. A miner collects all transactions available within the transaction pool, subject to constraints such as block size and block gas limits, and creates a block with them. This activity is done by all miners.

The miner constructs a new block and adds all transactions to it. Before adding these transactions, it will check whether any of the transactions are not already written in a block that it might receive from other miners. If so, it will discard those transactions.

The miner will add its own coinbase transaction for getting rewards for mining the block.

The next task for a miner is to generate the block header and perform the following tasks:

1. The miner takes hashes of two transactions at a time to generate a new hash till they get a single hash from all transactions. The hash is referred to as a **root** transaction hash or Merkle root transaction hash. This hash is added to the block header.
2. The miner also identifies the hash of the previous block. The previous block will become a parent to the current block, and its hash will also be added to the block header.
3. The miner calculates the state and receipts of the transaction root hashes and adds them to the block header.
4. A nonce and timestamp are also added to the block header.
5. A block hash consisting of both a block header and body is generated.
6. The mining process starts where the miner keeps changing the nonce value and tries to find a hash that will satisfy as an answer to the given puzzle. Keep in mind that everything mentioned here is executed by every miner in the network.
7. Eventually, one of the miners will be able to solve the puzzle and advertise the result to other miners in the network. The other miners will verify the answer and, if found correct, further verify every transaction, accept the block, and append it to their ledger instance.

This entire process is also known as PoW wherein a miner provides proof that they have worked on computing the final answer that is satisfactory as a solution to the puzzle. The header block and its content are shown in the following diagram:

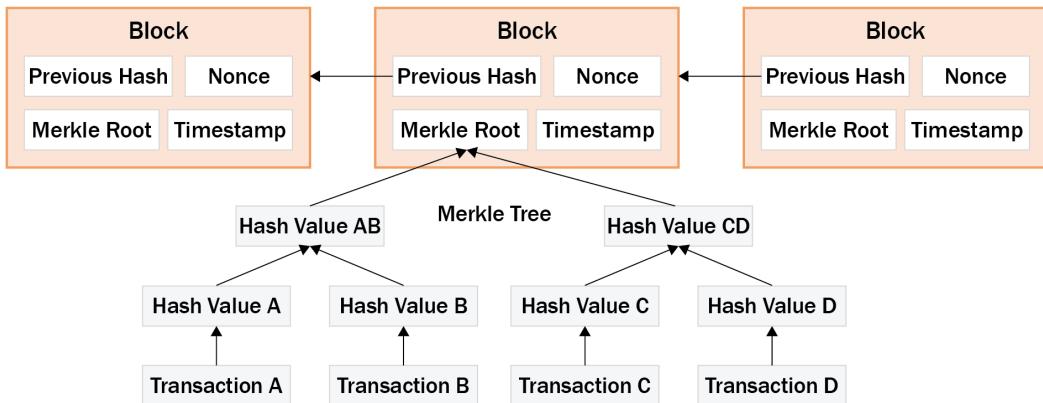


Figure 1.7 – PoW generating a Merkle root and using transactions

Let's now discuss the next mechanism.

Proof of stake

Ethereum 2.0 will launch a new consensus mechanism known as PoS. It is another algorithm to reach consensus within a distributed network architecture. While PoW is computationally heavy, PoS does not perform any compute-heavy activities. Under this mechanism, interested stakeholders can stake their Ether (a minimum of 32 ETH) with the network. Once the Ethers are staked, they are locked, and these stakeholders become validators. These validators have their nodes running, and their job is to attest to new blocks for their validity. Attestation here means that the validator is vouching for the correctness of the block and its constituent transactions.

After a minimum number of validators attest, the block is finalized and becomes a permanent part of the chain. The network will at random also choose a validator to create a new block for every epoch. The validators are not competing with each other anymore as in the case of the PoW consensus mechanism. This makes PoS more environmentally friendly by being energy efficient and also anyone with 32 Ethers can potentially become a validator. Now that we understand the different consensus mechanisms, it's time to visit the different types of nodes supported by Ethereum.

Ethereum nodes

Nodes represent the computers that are connected using a P2P protocol to form an Ethereum network. There are the following three types of nodes in Ethereum:

- The EVM
- Mining nodes (Ethereum 1.0)
- Validators (Ethereum 2.0)

Please note that this distinction is made to clarify concepts of Ethereum. In most scenarios, there is no dedicated EVM. Instead, all nodes act as miners as well as EVM nodes.

EVM

Think of an EVM as the execution runtime for smart contracts. EVMs are primarily responsible for providing a runtime that can execute code written in smart contracts. It can access accounts, both contract and externally owned, and its own storage data. It does not have access to the overall ledger but does have limited information about the current transaction.

EVMs are the execution components in Ethereum. The purpose of an EVM is to execute code in a smart contract line by line. However, when a transaction is submitted, the transaction is not executed immediately. Instead, it is added to a transaction pool. These transactions are not yet written to the Ethereum ledger.

Mining nodes

The mining nodes are responsible for generating, validating, and adding blocks to the chain. Mining nodes can be a full node, a light node, or an archive node. A full node has all the blocks (headers and transactions) since the genesis block and complete global state and can participate in generating and validating the blocks. Light nodes have blocks with headers only and are dependent on connected full nodes for any information they need. They require more bandwidth because of their chatty nature with full nodes but need less computing and storage. They are also faster while synchronizing blocks from full nodes. Archival nodes are again full nodes, but their usage is more for querying a node for historical information and reporting.

Ethereum validators

Validator nodes are again nodes responsible for generating new blocks for the chain. They do not run any mining process, and they collect all available transactions from the transaction pool and create a block. They broadcast the blocks to other validators for attestation. As part of the attestation process, validators execute each transaction and modify their state while adding a block to their chain.

Ethereum accounts

Accounts are the main building blocks for the Ethereum ecosystem. It is an interaction between accounts that Ethereum wants to store as transactions in its ledger. There are two types of accounts available in Ethereum – externally owned accounts and contract accounts. Each account, by default, has a property named `balance` that helps in querying the current balance of Ether.

Externally owned accounts

Externally owned accounts are accounts that are owned by people on Ethereum. Accounts are not referred to by name in Ethereum. When an externally owned account is created on Ethereum by an individual, a public/private key is generated. The private key is kept safe with the individual while the public key becomes the identity of this externally owned account. This public key is generally of 256 characters; however, Ethereum uses the first 160 characters to represent the identity of an account.

If Bob, for example, creates an account on an Ethereum network, whether private or public, he will have his private key available to himself while the first 160 characters of his public key will become his identity. Other accounts on the network can then send Ether or other cryptocurrencies based on Ether to this account.

An account on Ethereum looks like the one shown in the following screenshot:

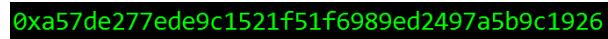


Figure 1.8 – An externally owned account identifier

An externally owned account can hold Ether in its `balance` and does not have any code associated with it. It can execute transactions with other externally owned accounts, and it can also execute transactions by invoking functions within contracts.

Contract accounts

Contract accounts are very similar to externally owned accounts. They are identified using their public address. They do not have a private key. They can hold Ether similar to externally owned accounts; however, they contain code for smart contracts consisting of functions and state variables.

Externally owned accounts are responsible for initiating transactions and each transaction in Ethereum requires gas, which is the cost of the transaction to be provided by the sender of the transaction. Gas has a relationship to its currency known as Ether. The next section will discuss the concepts related to Ether, gas, and transactions.

Ether, gas, and transactions

Ether is the currency of Ethereum. Every activity on Ethereum that modifies its state charges Ether as a fee. And miners who are successful in generating and writing a block in a chain are also rewarded Ether. Ether can easily be converted to dollars or other currencies through crypto exchanges.

Ethereum has a metric system of denominations known as units of Ether. The smallest denomination or base unit of Ether is called **wei**. The following is a list of the named denominations and their value in wei, which is available at <https://github.com/ethereum/web3.js/blob/0.15.0/lib/utils/utils.js#L40>:

```
var unitMap = {  
    ...  
    'wei' : '1'
```


Gas

In the previous section, it was mentioned that fees are paid using Ether for any transaction execution that leads to a state change in Ethereum. Ether is traded on public exchanges, and its price fluctuates daily. If Ether is used for paying fees, then the cost of using the same service can be high on a certain day while being low on other days. People will wait for the price of Ether to fall to execute their transactions. This is not ideal for a platform such as Ethereum. Gas helps in alleviating this problem. This is the internal currency of Ethereum. The execution and resource utilization costs are predetermined in Ethereum in terms of gas units. For each unit of gas, a price can be ascribed, known as the gas price. For example, for each unit of gas, a price of 10 gwei can be assigned. When both the number of gas units and gas price are multiplied together, it results in the **gas cost**. The **gas price** can be adjusted to a lower price when the price of Ether increases and a higher price when the price of Ether decreases.

Transactions

A transaction is an agreement between a buyer and a seller, a supplier and a consumer, or a provider and a consumer that there will be an exchange of assets, products, or services for currency, cryptocurrency, or some other asset, either in the present or in the future. Ethereum helps in executing the transaction. The following are the three types of transactions that can be executed in Ethereum:

- **The transfer of Ether from one account to another:** The accounts can be externally owned accounts or contract accounts. The following are the possible cases:
 - An externally owned account sending Ether to another externally owned account in a transaction

- An externally owned account sending Ether to a contract account in a transaction
 - A contract account sending Ether to another contract account in a transaction
 - A contract account sending Ether to an externally owned account in a transaction
 - **Deployment of a smart contract:** An externally owned account can deploy a contract using a transaction in an EVM.
 - **Using or invoking a function within a contract:** Executing a function in a contract that changes state is considered a transaction in Ethereum. If executing a function does not change a state, it does not require a transaction.

A transaction has some of the following important properties related to it:

- The `from` account property denotes the account that originates the transaction and represents an account that is ready to send some gas or Ether. Both gas and Ether concepts were discussed earlier in this chapter. The `from` account can be externally owned or a contract account.
 - The `to` account property refers to an account that is receiving Ether or benefits in lieu of an exchange. For transactions related to the deployment of the contract, the `to` field is empty. It can be externally owned or a contract account.
 - The `value` account property refers to the amount of Ether that is transferred from one account to another.
 - The `input` account property refers to the compiled contract bytecode and is used during contract deployment in an EVM. It is also used for storing data related to smart contract function calls along with their parameters. A typical transaction in Ethereum where a contract function is invoked is shown here. In the following screenshot, note the `input` field containing the function call to contract, along with its parameters:

Figure 1.9 – The transaction properties in a block

- The `blockHash` account property refers to the hash of the block to which this transaction belongs.
- The `blockNumber` account property is the block to which this transaction belongs.
- The `gas` account property refers to the amount of gas supplied by the sender who is executing this transaction.
- The `gasPrice` account property refers to the price per gas the sender was willing to pay in wei (we learned about wei in the *Ether* section earlier in this chapter). Total gas is computed at *gas units x gas price*.
- The `hash` account property refers to the hash of the transaction.
- The `nonce` account property refers to the number of transactions made by the sender prior to the current transaction.
- The `transactionIndex` account property refers to the serial number of the current transactions in the block.
- The `value` account property refers to the amount of Ether transferred in wei.
- The `v`, `r`, and `s` account properties relate to digital signatures and the signing of the transaction.

A typical transaction in Ethereum, where an externally owned account sends some Ether to another externally owned account, is shown here. Note that the `input` field is not used here. Since two Ethers were sent in the transaction, the `value` field is showing the value accordingly in wei, as shown in the following screenshot:

```
{ blockHash: '0x78ddc6d1d18a52811888dea659a69f35f424aa0ec48562b956d3524e80fcf893' ,  
  blockNumber: 105 ,  
  from: '0xa57de277ede9c1521f51f6989ed2497a5b9c1926' ,  
  gas: 90000 ,  
  gasPrice: BigNumber { s: 1 , e: 10 , c: [ 18000000000 ] } ,  
  hash: '0x93768f05999d54edde1982f82150b429b3cba0014233defab34701e6b6a7ec87' ,  
  input: '0x' ,  
  nonce: 2 ,  
  to: '0x9d2a327b320da739ed6b0da33c3809946cc8cf6a' ,  
  transactionIndex: 0 ,  
  value: BigNumber { s: 1 , e: 18 , c: [ 20000 ] } ,  
  v: '0x41' ,  
  r: '0x9efb14382840ab5fcdf2d33f32638e895beb9cee35d4d79675c183c7fddef8f5' ,  
  s: '0x658bac95226e3a8a90d497ce8c841e0833c01b5a2567bb8c2aa126ba95e1fb02' }
```

Figure 1.10 – Block properties in a block

One method to send Ether from an externally owned account to another externally owned account is shown in the following code snippet using the web3 JavaScript framework, which will be covered later in this book:

```
web.eth.sendTransaction({from: web.eth.accounts[0], to:
"0x9d2a327b320da739ed6b0da33
c3809946cc8cf6a", value: web.
toWei(2, 'ether')})
```

A typical transaction in Ethereum where a contract is deployed is shown in the following screenshot. In the following screenshot, note the `input` field containing the bytecode of the contract:

```
{ blockHash: '0x041cdd69390b130e0b54c53f2afb46d79a06708dcf8414aa1ba4bbb40a2786b7',
blockNumber: 6,
from: '0xa57de277ede9c1521f51f6989ed2497a5b9c1926',
gas: 1000000,
gasPrice: BigNumber { s: 1, e: 10, c: [ 1800000000 ] },
hash: '0x6f5a74e5191f745d0e38fa67841ba6b36cf03a7c0fd7729ef355fe77c86487a2',
input: '0x0606060405234156100057600080fd5b6102c380610004526004361061004b5763ffffffff7c0100
561005b57600080fd5b61006361012d565b5040516020808252819081018381815181526020019150805190602001908083360005b8381101
0191505b509250505060405180910390f35b34156100e557600080fd5b61012b60046024813581810190830135806020601f82018190048102
51561010002031660002900480601f0160208091040260200160405190810160405280929190818152602001828054600181600116156101000
01906020018083116101ae57829003601f168201915b50505050905b90565b60008180516101e99291602001906101ff565b5050565b60
1024057805160ff191683800117855561026d565b8280016001018555821561026d579182015b8281111561026d57825182559160200191906
77752e38ed74bb682568cb64c68f24734ab95e18740dee526b95eff79e40029',
nonce: 0,
to: null,
transactionIndex: 0,
value: BigNumber { s: 1, e: 0, c: [ 0 ] },
v: '0x42',
r: '0x29887013743c6fc9a4bb78c6d3ca2974eac6f41b21d2d0bb6fdf0e09b71202602',
s: '0xaaa9366553495c9c81ecb9806d0cdc4e0ed5d688797be099978e260ae53e34' }
```

Figure 1.11 – A sample transaction in a block

Blocks

Blocks are an important concept in Ethereum. They are containers for a transaction. A block contains multiple transactions. Each block has a different number of transactions based on the gas limit and block size. The gas limit will be explained in detail in later sections. The blocks are chained together to form a blockchain. Each block has a parent block, and it stores the hash of the parent block in its header. Only the first block, known as the genesis block, does not have a parent.

A typical block in Ethereum is shown in the following screenshot:

Figure 1.12 – A typical block in Ethereum

There are a lot of properties associated with a block, providing insights and metadata about it, and the following are some of the important properties along with their descriptions:

- The `difficulty` property determines the complexity of the puzzle/challenge given to miners for this block.
 - The `gasLimit` property determines the maximum gas allowed. This helps in determining how many transactions can be part of the block.
 - The `gasUsed` property refers to the actual gas used for this block for executing all transactions in it.
 - The `hash` property refers to the hash of the block.
 - The `nonce` property refers to the number that helps in solving the challenge.
 - The `miner` property is the account identifier of the miner, also known as coinbase or etherbase.
 - The `number` property is the sequential number of this block on the chain.
 - The `parentHash` property refers to the parent block's hash.
 - The `receiptsRoot`, `stateRoot`, and `transactionsRoot` properties refer to the Merkle trees discussed during the mining process.
 - The `transactions` property refers to an array of transactions that are part of this block.
 - The `totalDifficulty` property refers to the total difficulty of the chain.

An end-to-end transaction

Armed with the understanding of the foundational concepts of blockchain and Ethereum, it's time to see a complete end-to-end transaction and how it flows through multiple components and gets stored in the ledger.

In this example, Sam wants to send a digital asset (for example, dollars) to Mark. Sam generates a transaction message containing the `from`, `to`, and `value` fields and sends it across to the Ethereum network. The transaction is not written to the ledger immediately and instead is placed in a transaction pool.

The mining node creates a new block and takes all transactions from the pool honoring the gas limit criteria and adds them to the block. This activity is done by all miners on the network. Sam's transaction will also be a part of this process.

The miners compete, trying to solve the challenge thrown at them. The winner is the miner who can solve the challenge first. After a period (of seconds), one of the miners will advertise that they have found the solution to the challenge and that they are the winner and should write the block to the chain. The winner sends the solution of the challenge, along with the new block, to all other miners. The rest of the miners validate and verify the solution, and once satisfied that the solution is indeed correct and that the original miner has cracked the challenge, they accept the new block containing Sam's transaction to append in their instance of the ledger. This generates a new block on the chain that is persisted across time and space. During this time, the accounts of both parties are updated with the new balance. Finally, the block is replicated across every node in the network.

The preceding example can be well understood with the following diagram:

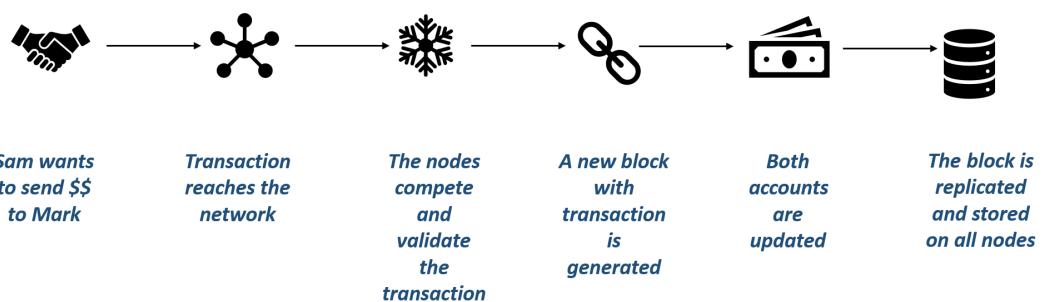


Figure 1.13 – The transaction flow in Ethereum

This is the time to introduce smart contracts after understanding some of the important concepts related to Ethereum. Smart contracts are one of the unique value propositions of Ethereum, and this book will delve deeper into smart contracts in subsequent chapters.

Smart contract

A contract is a legal document that binds two or more parties who agree to execute a transaction immediately or in the future. Since contracts are legal documents, they are enforced and implemented by law.

Examples of contracts are an individual entering into a contract with an insurance company for covering their health insurance, an individual buying a piece of land from another individual, or a company selling its shares to another company.

A smart contract is a custom logic and code deployed and executed within an Ethereum virtual environment. Smart contracts are digitized and codified rules of the transaction between accounts. Smart contracts help in transferring digital assets between accounts as an atomic transaction. Smart contracts can store data. The data stored can be used to record information, facts, associations, balances, and any other information needed to implement the logic for real-world contracts. Smart contracts are very similar to object-oriented classes. A smart contract can call another smart contract just as an object-oriented object can create and use objects of another class. Think of smart contracts as a small program consisting of functions. You can create an instance of the contract and invoke functions to view and update contact data along with the execution of some logic.

Writing smart contracts

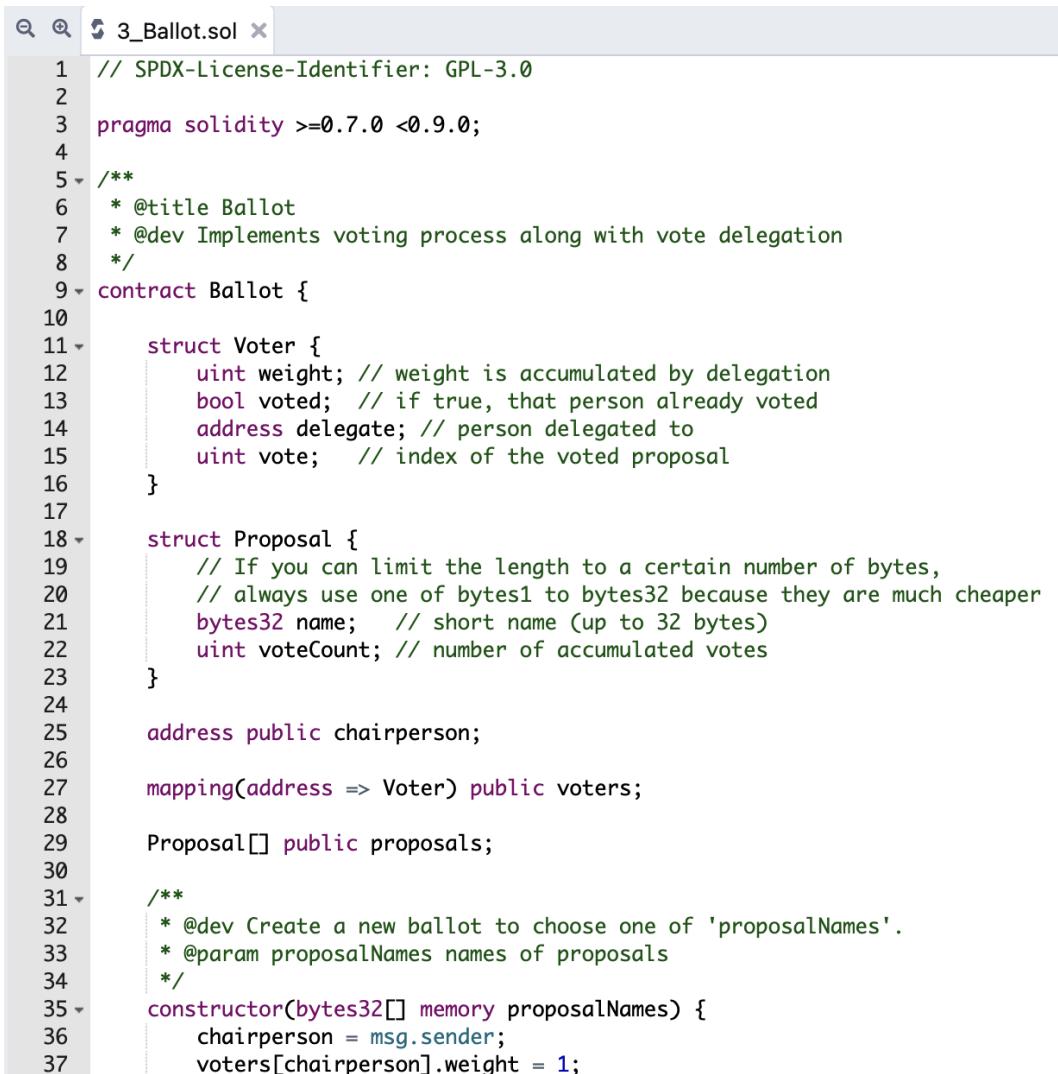
There are multiple smart contract authoring tools, including Visual Studio. However, the easiest and fastest way to develop smart contracts is to use a browser-based tool known as **Remix**. This is available at <http://remix.ethereum.org>. Remix is a new name and was earlier known as **browser-solidity**.

Remix provides a rich integrated development environment in a browser for authoring, developing, deploying, and troubleshooting contracts written using the Solidity language. All contract management-related activities such as authoring, deploying, and troubleshooting can be performed from the same environment without using other tools and utilities.

Not everyone is comfortable using the online version of Remix to author their smart contracts. Remix is an open source tool that can be downloaded from <https://github.com/ethereum/browser-Solidity> and compiled to run a private version on a local computer.

Another advantage of running Remix locally is that it can connect to local private chain networks directly; otherwise, users will first have to author the contract online and then copy it to a file, compile it, and deploy it manually to a private network. Let's explore Remix by performing the following steps:

1. Navigate to `remix.ethereum.org`. The site will open in a browser with a default contract, as shown in the following screenshot. If you do not need this contract, it can be deleted:



The screenshot shows the Remix IDE interface with the file `3_Ballot.sol` open. The code is a Solidity smart contract named `Ballot`. It defines two structures: `Voter` and `Proposal`. The `Voter` structure contains fields for weight, voted status, delegate address, and vote index. The `Proposal` structure contains a name (as a bytes32), a vote count, and a mapping from address to Voter. The contract also has a chairperson address and a mapping from address to Voter. A constructor is defined to initialize the chairperson and voters array.

```

1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6  * @title Ballot
7  * @dev Implements voting process along with vote delegation
8 */
9 contract Ballot {
10
11     struct Voter {
12         uint weight; // weight is accumulated by delegation
13         bool voted; // if true, that person already voted
14         address delegate; // person delegated to
15         uint vote; // index of the voted proposal
16     }
17
18     struct Proposal {
19         // If you can limit the length to a certain number of bytes,
20         // always use one of bytes1 to bytes32 because they are much cheaper
21         bytes32 name; // short name (up to 32 bytes)
22         uint voteCount; // number of accumulated votes
23     }
24
25     address public chairperson;
26
27     mapping(address => Voter) public voters;
28
29     Proposal[] public proposals;
30
31 /**
32  * @dev Create a new ballot to choose one of 'proposalNames'.
33  * @param proposalNames names of proposals
34  */
35 constructor(bytes32[] memory proposalNames) {
36     chairperson = msg.sender;
37     voters[chairperson].weight = 1;

```

Figure 1.14 – The default Ballot contract in Remix

2. The first thing we need to do is to create a new contract by selecting + from Remix's left menu bar.
3. Then, provide a name for a new Solidity file that has a .sol extension. Name the contract HelloWorld and hit *Enter* on the keyboard to create a new Solidity file, as shown in the following screenshot. This will create a blank file:

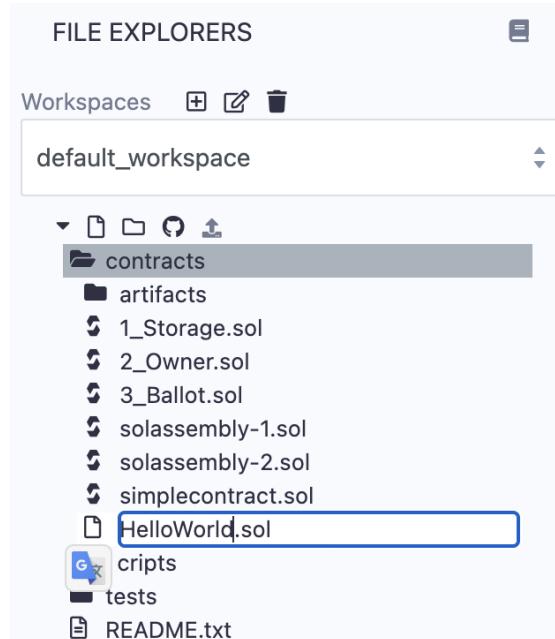


Figure 1.15 – The HelloWorld.sol file in Remix

4. Type the following code in the empty authoring pane to create your first contract. This contract will be explained in detail in *Chapter 3, Solidity Development*. For now, it is sufficient to understand that the contract is created using the `contract` keyword. You can declare global state variables and functions, and contracts are saved with the .sol file extension. In the following code snippet, the HelloWorld contract returns the Hello World literal value when the GetHelloWorld function is executed:

```
pragma solidity >=0.7.0 <0.9.0;
contract HelloWorld
{
    string private stateVariable = "Hello World";
    function GetHelloWorld() public view returns ( string
        memory)
```

```
{  
    return stateVariable;  
}  
}
```

The extreme left of Remix has a wunderbar comprising multiple tabs. The **FILE EXPLORERS** tab helps in managing workspace along with project folder hierarchy and organization. Contract files are also managed using this tab. It has good integration with GitHub and helps in pushing changes to repositories.

The next is the **SOLIDITY COMPILER** tab. This tab helps in choosing an appropriate compiler version that helps in compiling the contracts. The **SOLIDITY R COMPILER** tab compiles the contract into bytecode – code that is understood by Ethereum. It displays warnings and errors as you author and edit the contract. These warnings and errors should be taken seriously, and they really help in creating robust contracts.

The third tab, **DEPLOY AND RUN TRANSACTIONS**, helps in deploying and executing the smart contracts to an environment. There are out-of-the-box JavaScript VMs (London and Berlin) in-built within the Remix page, and each represents a unique environment. This tab helps in connecting to any of the available VMs. It is also possible to connect to a custom Ethereum network using the `Injected web3` and `web3 provider` environment options. Remix comes bundled with the Ethereum runtime within the browser. This tab allows you to deploy the contract to this runtime using the **JavaScript VM** environment in the **Environment** option. The **Injected Web3** environment is used along with tools such as Mist and MetaMask, which will be covered in the next chapter. **Web3 provider** can be used when using Remix in a local environment connecting to a private network. In our case for this chapter, the default, **JavaScript VM**, is sufficient. The rest of the options will be discussed later in *Chapter 3, Introducing Solidity*.

The **Solidity static analysis** tab helps in providing alerts related to implementing best practices within smart contracts. It provides a set of rules that are evaluated during compile time, and the user is notified of any rule not getting satisfied. There is also a **Solidity unit testing** tab that helps in executing unit tests against the smart contracts.

The previously mentioned tabs are available by default with Remix. They can be removed, and more features can be added using **PLUGIN MANAGER**, which is the last tab available with Remix.

5. Once a smart contract like the one shown in the next screenshot is authored using Remix, it can be deployed directed to the target environment (JavaScript VMs) using the **DEPLOY & RUN TRANSACTIONS** tab. This tab provides a **Deploy** button that will deploy the contract. It is always better to compile the contract before deploying, as it helps in identifying any issues with the contract. Deploying a contract will internally first compile it:

```
1 pragma solidity >=0.7.0 <0.9.0;
2
3 contract HelloWorld
4 {
5     string private stateVariable = "Hello World";
6     function GetHelloWorld() public view returns ( string memory)
7     {
8         return stateVariable;
9     }
10}
11
```

Figure 1.16 – The HelloWorld Smart contract

6. Click on the **Deploy** button (after selecting an appropriate contract) to deploy the contract to Ethereum JavaScript VM (either London or Berlin), and this will display the deployed contract along with its associated functions. Since we only had a single function, `GetHelloWorld`, the same function is displayed, as shown in the following screenshot:

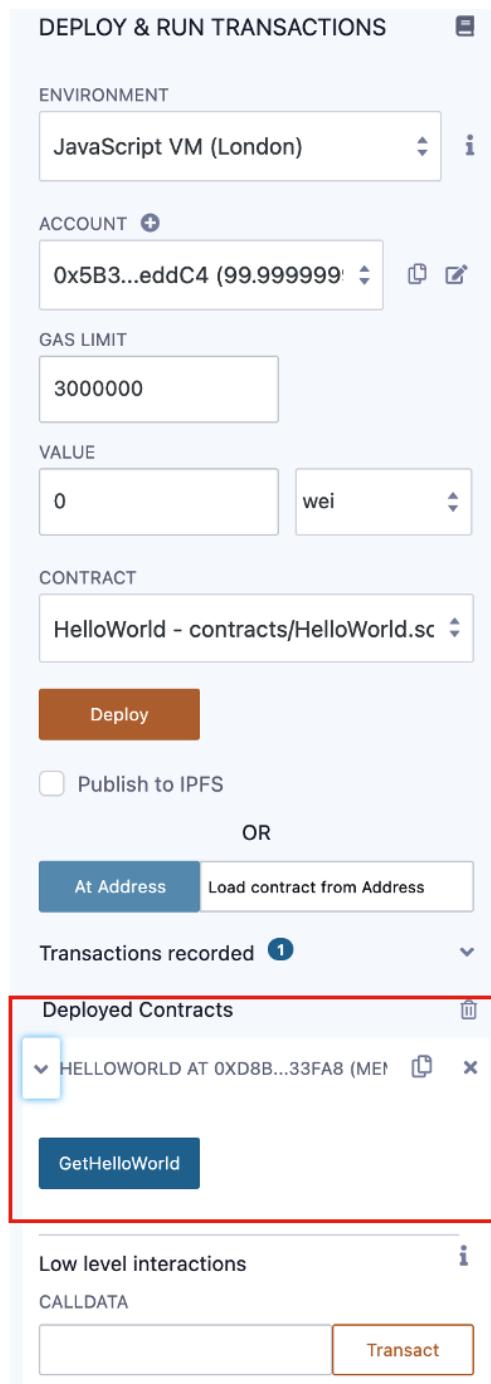


Figure 1.17 – Deploying the HelloWorld smart contract in Remix

7. Click on the **GetHelloWorld** button to invoke and execute the function. The results of execution are shown in the following screenshot:

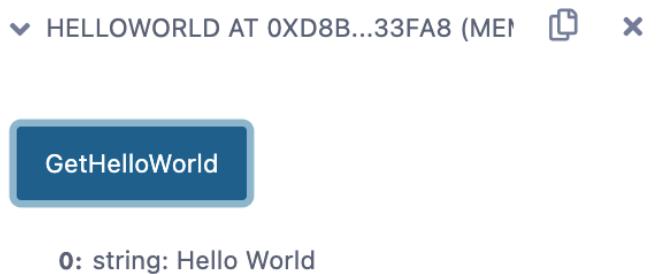


Figure 1.18 – Executing the GetHelloWorld function in Remix

Congratulations! You have created, deployed, and also executed a function on your first contract. The code for the `HelloWorld` contract is found in this chapter's GitHub repository and can be used in Remix if you are not interested in typing the contract.

The internals of smart contract deployment

Remix makes the deployment of contracts a breeze; however, it performs a lot of steps behind the scenes. It is always useful to understand the process of deploying contracts to have finer control over the deployment process.

The first step is the compilation of contracts. The compilation is done using the Solidity compiler. The next chapter will show you how to download and compile a contract using the Solidity compiler.

The compiler generates the following two major artifacts:

- The **Application Binary Interface (ABI)** definition
- Contracts bytecode

Think of the ABI as an interface consisting of all external and public function declarations along with their parameters and return types. The ABI defines the contract, and any caller wanting to invoke any contract function can use the ABI to do so.

The bytecode represents the low-level instruction set for the contract, and it is deployed in the Ethereum ecosystem. The bytecode is required during deployment, and the ABI is needed for invoking functions in a contract.

A new instance of a contract is created using the ABI definition.

Deploying a contract itself is a transaction. A transaction is created for deploying the contract on Ethereum. The bytecode and ABI are necessary inputs for deploying a contract.

As any transaction execution costs gas in Ethereum, an appropriate quantity of gas should be supplied while deploying the contract. As and when the transaction is mined, the contract will be available for interaction through the contract address.

Using the newly generated address, callers can invoke functions within the contract.

Summary

This chapter was an introduction to blockchains and, more specifically, to Ethereum. Having a good understanding of the big picture of how blockchains and Ethereum work will go a long way in writing robust, secure, and cost-effective smart contracts using Solidity.

This chapter covered the basics of blockchain, explained what blockchains are, why they are important, and how they help in building decentralized and distributed applications. The architecture of Ethereum was discussed in brief along with some of the important concepts, such as transactions, blocks, gas, Ether, accounts, cryptography, and mining.

This chapter also touched briefly on the topic of smart contracts and how to use Remix to author and execute them. I've kept this chapter brief, since the rest of the book will explain these concepts further and help you to quickly develop Solidity-based smart contracts.

You'll notice that this chapter does not contain any mention of Ethereum tools and utilities. This is what we will cover in the next chapter, by diving straight in and installing Ethereum and its toolset. The Ethereum ecosystem is quite rich, and there are lots of tools. We will cover important ones, such as `web3.js`, TestRPC, Geth, Mist, and MetaMask.

Questions

1. What are the two consensus mechanisms supported by Ethereum, although not together?
2. What two important artifacts are generated as part of compiling Solidity smart contracts?

Further reading

The Ethereum documentation is a great place to get started with basic concepts:
<https://ethereum.org/en/developers/docs/>.

2

Installing Ethereum and Solidity

In the previous chapter, we had an overview of all the major concepts related to blockchains, particularly focusing on ones related to Ethereum, and discussed the fundamentals related to working with blockchains in general. Ethereum-based blockchain solutions can be deployed to multiple networks. They can be deployed on public networks, test networks, or private networks.

This chapter focuses on introducing and deploying Ethereum-based tools and utilities that are needed for building Ethereum-based solutions. There are plenty of tools in the Ethereum ecosystem, and this chapter will focus on some of the most important and necessary ones. In this chapter, the tools will be deployed on a macOS machine; however, they can be deployed on Linux and Windows as well. This will also be used as our development environment for testing, deploying, creating, and using Solidity contracts throughout this book.

By the end of the chapter, you'll know how to install some of the most important utilities related to the Ethereum ecosystem along with Geth, which is one of the main Ethereum implementations.

In this chapter, we'll cover the following topics:

- Ethereum networks
- Installing and configuring Geth
- Creating a private network
- Installing ganache-cli
- Installing the Solidity compiler
- Installing the web3 framework
- Installing and working with MetaMask

Technical requirements

The tools and technologies used in this chapter are as follows:

- Geth
- ganache-cli
- Node.js
- Solc
- MetaMask
- The web3 framework
- The command-line terminal on macOS

Ethereum networks

Ethereum is an open source platform for creating and deploying distributed applications.

Ethereum is backed up by a large number of computers (also known as nodes) – all interconnected and storing data in a *distributed ledger*. The term here means that a copy of the ledger is available to each and every node on the network. It provides flexibility to its developers to deploy their solutions to multiple types of networks – that is, permissioned, private, or public. Developers should choose an appropriate network based on their requirements and use cases. These different networks also help in deploying solutions and smart contracts on networks that do not actually cost any ether or money. There are networks that are free along with ones that require their users to pay in terms of ether or other currencies for their usage. Let's discuss these different types of ether networks in more detail.

Main network

The main Ethereum network is a global public network that anybody can use. It can be accessed using an account, and anyone can deploy their solutions and smart contracts. Using a main network incurs costs in terms of gas. As the main network evolves due to hard forks, its codename also changes. At the time of writing, London is the code name for the main network, and some of the earlier ones were called Petersburg, Berlin, Istanbul, and Homestead. The main network is a public chain accessible over the internet, and anybody can connect to it and access both data and transactions stored in it.

Test network

A **test network** helps with testing facilities and increases the adoption of Ethereum blockchains. It is similar to the main network; however, it has a different ledger and storage. Using these networks does not cost anything for the deployment and usage of contracts; they are completely free of cost. This is because test ethers can be generated using faucets and used on these networks. There are multiple test networks available at the time of writing, such as Ropsten, Kovan, Goerli, and Rinkeby. Let's glance through each in brief.

Ropsten

Ropsten is one of the first test networks, which uses **Proof of Work (PoW)** consensus methods for generating blocks. It was earlier known as **Morden**. As mentioned before, it is completely free to use, and it can be used during the building and testing of smart contracts. It can be used by using the `--testnet` option available in Geth (which will be explained in detail in the next section). This is by far the most popular test network.

Rinkeby

Rinkeby is another Ethereum-based test network, which uses **Proof of Authority (PoA)** as its consensus mechanism. PoW and PoA are different mechanisms for building consensus among miners. PoW is robust enough to maintain immutability and decentralization of data; however, it has drawbacks in not having enough control over miners. PoA, on the other hand, has all the benefits of PoW along with having more control over miners.

Kovan

Kovan test networks can only be used by parity clients and hence won't be discussed or used in this book. However, more information is available at <https://kovan-testnet.github.io/website/>.

Goerli

The **Goerli** test network is again a PoA-based testnet, available at <https://goerli.net>, and it won't be discussed or used in this book.

Private network

A **private** network is created and hosted on private infrastructure. Private networks are controlled by a single organization, and it has full control over it. There are solutions, contracts, and use cases that an organization might not want to put on a public network, even for test purposes. They may want to use private chains for development, testing, and production environments. Organizations should create and host a private network, and they will have full control over it. Further in this chapter, we will see how to create your own private network.

Consortium network

A **consortium** network is also a private network but with a difference. The consortium network comprises nodes, each managed by a different organization. In effect, no organization has control over the data and chain. However, it is shared within an organization, and everyone in this organization can view and modify the current state. These might be accessible through the internet or completely private networks using a VPN.

Installing and configuring Geth

Implementation of Ethereum nodes and clients is available in multiple languages, including Go, C++, Python, JavaScript, Java, and Ruby. The functionality or usability of these clients is the same across languages, and developers should choose the language implementation they are most comfortable with. This book uses the Go implementation known as Geth, which acts as an Ethereum client to connect to public and test networks. It is also used to create the mining and EVM (transaction nodes) for private networks. Geth is a command-line tool written in Go for creating a node and miners on a private chain. It can be installed on Windows, Linux, and Mac. Now, it's time to install Geth.

Installing Geth on macOS

Installation of Geth on macOS is quite straightforward. Mac has Homebrew as its package manager, and Geth can be installed using Homebrew by executing a couple of commands.

`brew` is the executable that is used for the installation of packages. Ethereum is not available as part of the Homebrew native repository, so the Ethereum repository should be added as a repository to Homebrew. The `brew tap Ethereum/Ethereum` command adds the Ethereum repository to the list managed by Homebrew.

Once the repository has been added, Geth can be installed by executing the `brew install Ethereum` command.

Installing Geth on Windows

The first step in creating a private Ethereum network is to download and install the Geth (`go-ethereum`) tool.

In this section, the steps to download and install Geth on Windows are as follows:

1. Geth can be downloaded from the <https://ethereum.github.io/go-ethereum/downloads/> page. It is available for both 32- and 64-bit machines.
2. After downloading, start the installation process by executing the executable and following the steps, accepting the defaults. It's a recommended practice to install development tools in development environments.
3. Once Geth is installed, it should be available from Command Prompt or PowerShell.
4. Open Command Prompt and type `geth -help`.

Important Note

A word of caution here – just typing `Geth` and executing it will connect Geth to a public main network, and it will start syncing and downloading all the blocks and transactions.

The current chain has more than 1,100 GB of data at the time of writing, and it is growing every day. The `help` command shows all the commands and options available with Geth. It will also show the current version, as shown in the following screenshot:

```
[(base) riteshmodi@riteshm ~ % geth help
NAME:
  geth - the go-ethereum command line interface

  Copyright 2013-2021 The go-ethereum Authors

USAGE:
  geth [options] [command] [command options] [arguments...]

VERSION:
  1.10.7-stable

COMMANDS:
  account          Manage accounts
  attach           Start an interactive JavaScript environment (connect
  console          Start an interactive JavaScript environment
  db               Low level database operations
  dump              Dump a specific block from storage
  dumpconfig       Show configuration values
  dumpgenesis     Dumps genesis block JSON configuration to stdout
  export            Export blockchain into file
  export-preimages Export the preimage database into an RLP stream
  import            Import a blockchain file
  import-preimages Import the preimage database from an RLP stream
  init              Bootstrap and initialize a new genesis block
  js                Execute the specified JavaScript files
  license           Display license information
  makecache         Generate ethash verification cache (for testing)
  makedag          Generate ethash mining DAG (for testing)
  removedb         Remove blockchain and state databases
  show-deprecated-flags Show flags that have been deprecated
  snapshot          A set of commands based on the snapshot
  version           Print version numbers
  version-check    Checks (online) whether the current version suffers
  inerabilities   Manage Ethereum presale wallets
  wallet            Shows a list of commands or help for one command
```

Figure 2.1 – The Geth command options and version

Geth is based on the JSON-RPC protocol. It defines the specification for remote procedure calls, with payload encoded in the JSON format. Geth allows connectivity to JSON-RPC using the following three different protocols:

- **Inter-Process Communication (IPC):** This protocol is used for inter-process communication, generally within the same computer.
- **Remote Procedure Calls (RPC):** This protocol is used for inter-process communication across computers. This is generally based on TCP and HTTP protocols.

- **WebSocket (WS):** This protocol is used to connect to Geth using sockets over HTTP.

There are many commands, switches, and options for configuring Geth, which include the following:

- Configuring the IPC, RPC, and WS protocols
- Configuring network types to connect – private, Ropsten, and Rinkeby
- Mining options
- Console and API
- Networking
- Debugging and logging

Some of the important options for creating a private network will be discussed in the next section.

Geth can be used to connect to a public network by just running Geth without any options. London is the current name of public Ethereum. Its `networkid` and `ChainID` values are 1, as shown in the following screenshot:

```
INFO [12-15|20:38:55.594] Opened ancient database           databases/Users/riteshmodi/Library/Ethereum/geth/chaindata/ancient readonly=false
INFO [12-15|20:38:55.653] Initialised chain configuration config["ChainID": 1] Homestead: 1150000 DAO: 1920000 DAOsupport: true EIP150: 2463
000 EIP155: 2675000 EIP158: 2675000 Byzantium: 4370000 Constantinople: 7280000 Petersburg: 7280000 Istanbul: 9069000, Muir Glacier: 9200000, Berlin: 12244000, London: 12965000, Engine: ethash"
INFO [12-15|20:38:55.653] Disk storage enabled for ethash caches dir=/Users/riteshmodi/Library/Ethereum/geth/ethash count=3
INFO [12-15|20:38:55.656] Disk storage enabled for ethash DAGs dir=/Users/riteshmodi/Library/Ethash count=2
INFO [12-15|20:38:55.657] Initialising Ethereum protocol network=1 dbversion=8
```

Figure 2.2 – Geth's main network ID and name

The following are the network IDs used for connecting to the following different networks:

- The 1 chain ID represents a London public network.
- The 2 chain ID represents Morden (not used anymore).
- The 3 chain ID represents Ropsten.
- The 4 chain ID represents Rinkeby.
- A chain ID above 5 can be used to represent a private network.

Geth provides the `--ropsten` option to connect to the Ropsten network, the `--rinkeby` option to connect to the Rinkeby test network, and the `--goerli` option to connect to the Goerli test network. These should be used in conjunction with the `networkid` command option. It also provides the `-mainnet` option to connect to the main network.

Now that we understand Geth, we can use it to create a new private Ethereum network.

Creating a private network

After Geth is installed, it can be configured to run locally without connecting to any network on the internet. Every chain and network has a genesis block or first block. This block does not have any parent and is the first block of a chain. A `genesis.json` file is required to create this first block. A sample `genesis.json` file is shown in the following code snippet:

```
{  
  "config": {  
    "chainId": 9999,  
    "homesteadBlock": 0,  
    "eip150Block": 0,  
    "eip150Hash": 0,  
    "eip155Block": 0,  
    "eip158Block": 0,  
    "byzantiumBlock": 0,  
    "constantinopleBlock": 0,  
    "petersburgBlock": 0,  
    "istanbulBlock": 0,  
    "ethash": {}  
  },  
  "nonce": "0x0",  
  "timestamp": "0x62272fde",  
  "extraData": "0x00000000000000000000000000000000000000000000000000000000000000  
    00000000000000000000",  

```

Let's take a look at the following steps to create a private network:

1. The `genesis.json` file should be passed to Geth to initialize the private network. The Geth node also needs to store the blockchain data and account keys. This information should also be provided to Geth while initializing the private network.
2. The following `geth init` command initializes the node with the `genesis.json` file and target data directory location to store the chain data and key store information:

```
-th init ./genesis.json --datadir=.
```

The preceding command line will generate the following output:

```
(base) riteshmodi@riteshm Ethereum % geth init ./genesis.json --datadir=/
INFO [12-14|17:33:58.510] Maximum peer count
INFO [12-14|17:33:58.512] Set global gas cap
INFO [12-14|17:33:58.512] Allocated cache and file handles
INFO [12-14|17:33:58.614] Writing custom genesis block
INFO [12-14|17:33:58.615] Persisted trie from memory database
INFO [12-14|17:33:58.615] Successfully wrote genesis state
INFO [12-14|17:33:58.616] Allocated cache and file handles
INFO [12-14|17:33:58.717] Writing custom genesis block
INFO [12-14|17:33:58.717] Persisted trie from memory database
INFO [12-14|17:33:58.718] Successfully wrote genesis state
INFO [12-14|17:33:58.718] Maximum peer count
ETH=50 LES=0 total=50
cap=50,000,000
database=/Users/riteshmodi/Ethereum/geth/chaindata cache=16.00MiB handles=16
nodes=0 size=0.00B time="22.911us" gnodes=0 gcsiz=0.00B gctime=0s livenodes=1 livesize=0.00B
databasechaindata hash=18b9fa..a467f9
databasechaindata hash=18b9fa..a467f9
databasechaindata hash=18b9fa..a467f9
databasechaindata hash=18b9fa..a467f9
databasechaindata hash=18b9fa..a467f9
nodes=0 size=0.00B time="2.409us" gnodes=0 gcsiz=0.00B gctime=0s livenodes=1 livesize=0.00B
database=lightchaindata hash=18b9fa..a467f9
```

Figure 2.3 – The Geth initialization of a private network with the genesis file

3. After the Geth node is initialized with the genesis block, as shown in the preceding screenshot, Geth can be started. Geth uses the IPC protocol by default and will be enabled. To ensure that the Geth node is reachable using the RPC protocol, the RPC options need to be provided explicitly.
4. To set up an environment as a Geth node, execute the following command line:

```
geth --datadir= ./ --networkid=9999 --identity="ritesh" --
http - http.api="admin,debug,txpool,mi" --eth,net,web3,personal" --snapshot=false
```

The preceding command line will generate the following output:

```
INFO [12-14|17:38:41.613] Maximum peer count
INFO [12-14|17:38:41.614] Snap sync requested, enabling --snapshot
INFO [12-14|17:38:41.614] Set global gas cap
INFO [12-14|17:38:41.615] Allocated trie memory caches
INFO [12-14|17:38:41.615] Allocated cache and file handles
INFO [12-14|17:38:41.756] Opened ancient database
INFO [12-14|17:38:41.757] Initialised chain configuration
: 0 Constantinople : Petersburg: 0 Istanbul: 0 Muir Glacier: <nil>, Berlin: 0, London: 0, Engine: unknown"
INFO [12-14|17:38:41.758] Disk storage enabled for ethash caches
INFO [12-14|17:38:41.758] Disk storage enabled for ethash DAGs
INFO [12-14|17:38:41.758] Initialising Ethereum protocol
INFO [12-14|17:38:41.769] Loaded most recent local header
INFO [12-14|17:38:41.770] Loaded most recent local full block
INFO [12-14|17:38:41.770] Loaded most recent fast block
INFO [12-14|17:38:41.771] Loaded local transaction journal
INFO [12-14|17:38:41.772] Regenerated local transaction journal
INFO [12-14|17:38:41.773] Gasprice oracle is ignoring threshold set
INFO [12-14|17:38:41.773] Starting peer-to-peer node
INFO [12-14|17:38:41.948] New local node record
INFO [12-14|17:38:41.949] Started P2P networking
f70fe3b7c03e72ddcd7b6227fe433eeef517fe@127.0.0.1:30303
INFO [12-14|17:38:41.951] IPC endpoint opened
INFO [12-14|17:38:41.951] HTTP server started
INFO [12-14|17:38:48.559] New local node record
ETH=50 LES=0 total=50
cap=50,000,000
clean=154.000MiB dirty=256.00MiB
database=/Users/riteshmodi/Ethereum/geth/chaindata cache=512.00MiB handles=5120
database=/Users/riteshmodi/Ethereum/geth/chaindata/ancient readonly=false
config=(ChainID: 9999 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: 0 EIP155: 0
: 0 Constantinople : Petersburg: 0 Istanbul: 0 Muir Glacier: <nil>, Berlin: 0, London: 0, Engine: unknown"
dir=/Users/riteshmodi/Ethereum/geth/ethash count=3
dir=/Users/riteshmodi/Library/Ethash
dir=/Users/riteshmodi/Library/ethash
network=9999 abversion=8
number=0 hash=18b9fa..a467f9 td=131,072 age=52y8mo2w
number=0 hash=18b9fa..a467f9 td=131,072 age=52y8mo2w
number=0 hash=18b9fa..a467f9 td=131,072 age=52y8mo2w
transactions=0 dropped=0
transactions=0 accounts=0
threshold=2
instance=geth/ritesh/v1.10.7-stable/darwin-amd64/go1.16.6
seq=3 id=9aa07e69434898 ip=127.0.0.1 udp=30303 tcp=30303
self=enode://e195935e91fcbae8401c082d659899d2adcc29615740e877278b30809b78a223db11209
url=/Users/riteshmodi/Ethereum/geth.ipc
endpoint=127.0.0.1:8545 prefix= cors= vhosts=localhost
seq=4 id=9aa07e69434898 ip=171.49.239.88 udp=30303 tcp=30303
```

Figure 2.4 – Configuring and using Geth to the private network

There are a lot of important activities happening when this command is executed.

The command is executed with the `datadir` information, enabling RPCs, modules, and APIs that are exposed from this node instance when using an RPC to connect, and a `networkid` value of 9999 denotes that it is a private network. The result of executing this command also provides useful insights.

First, `etherbase` or `coinbase` is not set. The `coinbase` or `etherbase` account should be created and set before mining is started. As of now, mining has not started, although it was possible to auto-start mining with this command itself. The information about the current database location is printed on the screen. The output also displays `ChainID` and whether it is connected to a Homestead public network; a value of zero means it is not. The output also contains the `enode` value, which is a node identifier on the network. If more nodes want to join this network, they should provide this `enode` value to join this chain and network. Toward the end, the output shows that the RPC protocols are up and running and accepting requests. The `iendpoint` RPC endpoint is available at `http://127.0.0.1:8545` or `http://localhost:8545`. Take a look at the following command line:

```
geth --data-dir= ./ --networkid=9999 --identity=ritesh  
--http -- http.  
api="admin,debug,txpool,miner,eth,net,web3,personal" --  
snapshot=false
```

The preceding command will get the private Ethereum node up and running. However, note that the command runs as a service. Additional commands cannot be executed through it. To manage existing running Geth nodes, open another command window on the same computer and type the `geth attach ./geth.ipc` command to connect using the IPC protocol. You will get the following output:

```
[(base) riteshmodi@riteshm Ethereum % geth attach ./geth.ipc  
Welcome to the Geth JavaScript console!  
  
instance: Geth/ritesh/v1.10.7-stable/darwin-amd64/go1.16.6  
at block: 0 (Thu Jan 01 1970 05:30:00 GMT+0530 (IST))  
datadir: /Users/riteshmodi/Ethereum  
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0  
  
To exit, press ctrl-d
```

Figure 2.5 – Attaching to the Geth instance from another prompt

5. To connect to a private Geth instance through an RPC endpoint, use the `Geth attach http://localhost:5,8545`, or `Geth attach http://127.0.0.1:8545` commands to connect to a local running instance of Ethereum. If you see a different output than shown here, it's because the `coinbase` account is already set in my case. Adding a `coinbase` account is shown later in this section.

The default RPC port on which these endpoints are hosted is 8545, which can be changed using the `-rpcport` Geth command-line option. The IP address can be changed using the `-rpccaddr` option.

- After connecting to a Geth node, it's time to set up the `coinbase` or `etherbase` account. For this, a new account should be created first. To create a new account, use the `newAccount` method of the `personal` object. While creating a new account, provide `passphrase`, which acts like a password for the account. The output of this execution is the account ID, as shown in the following screenshot:

```
[> personal.newAccount()
[Passphrase:
[Repeat passphrase:
"0xcb9bd98d0132fa03814fed42ba268972b4a6cf84"]
```

Figure 2.6 – Creating a new account on a private network

- With the account ID provisioned, it should be tagged as a `coinbase` or `etherbase` account. To do this, the Geth provider has to change the `coinbase` address.`.miner` object with the `setEtherBase` function. This method will change the current `coinbase` to the provided account. The output of the command is `true` or `false`, as shown in the following screenshot:

```
[> miner.setEtherbase("0xcb9bd98d0132fa03814fed42ba268972b4a6cf84")
true
```

Figure 2.7 – Setting the coinbase account for mining on a private network

- Now, run the following query to find the current `coinbase` account by executing the following command:

```
eth.coinbase
```

It should output the same account address that was recently created, as shown in the following screenshot:

```
[> eth.coinbase
"0xcb9bd98d0132fa03814fed42ba268972b4a6cf84"]
```

Figure 2.8 – Validating the coinbase account on a private network

With `coinbase` set with a valid account and the Geth node up and running, mining can now get started, and since we just have one miner, all rewards will go to this miner, and its `coinbase` account will be credited with ethers.

- To start mining, execute the following command:

```
miner.start()
```

You can also use the following command line:

```
miner.start(4)
```

The parameter to the `start` method represents the number of threads used for mining. This will result in mining getting started, and the same can be viewed from the original command window:

```
INFO [12-15|21:06:10.370] Successfully sealed new block
INFO [12-15|21:06:10.371] ↘ block reached canonical chain
INFO [12-15|21:06:10.372] ↗ mined potential block
INFO [12-15|21:06:10.373] Commit new mining work
INFO [12-15|21:06:10.412] Successfully sealed new block
INFO [12-15|21:06:10.413] ↘ block reached canonical chain
INFO [12-15|21:06:10.413] ↗ mined potential block
INFO [12-15|21:06:10.413] Commit new mining work
INFO [12-15|21:06:10.836] Successfully sealed new block
INFO [12-15|21:06:10.837] ↘ block reached canonical chain
INFO [12-15|21:06:10.837] ↗ mined potential block
number=76 sealhash=29238d..18e425 hash=212ab0..7edd4c elapsed=8.883s
number=69 hash=b79ab8..fe8927
number=76 hash=212ab0..7edd4c
number=77 sealhash=735990..4c52e8 uncles=0 txs=0 gas=0 fees=0 elapsed=2.996ms
number=77 sealhash=735990..4c52e8 hash=0ea032..4466aa elapsed=39.875ms
number=78 hash=95475a..4466a8
number=77 hash=0ea032..4466aa
number=78 sealhash=644788..5ab5f0 uncles=0 txs=0 gas=0 fees=0 elapsed="116.332µs"
number=78 sealhash=644788..5ab5f0 hash=ca552b..60596b elapsed=423.861ms
number=77 hash=d17e8..a822df
number=78 hash=ca552b..60596b
number=78 hash=ca552b..60596b
```

Figure 2.9 – Mining using a private network

Mining can be stopped from the second command window using the `miner.stop()` command.

Using a live Ethereum network for development and testing can be a productivity dampener. Instead, another utility known as `ganache-cli` can be used for this purpose. The installation of `ganache-cli` is described next.

Installing Ganache

There are two distinct phases in the overall modification and writing of transactions to a ledger using Ethereum:

- The first phase involves creating a transaction and putting it in a transaction pool.
- The second phase that happens periodically is to get all transactions from a transaction pool and mine them. Mining here means writing those transactions to the Ethereum database or ledger.

From this description, you'd know it would be a time-consuming process if the same process was used for development and testing purposes. To ease the process of the development and testing of solutions and smart contracts on Ethereum, Ganache was created, originally known as TestRPC. `ganache-cli`, by itself, contains both the Ethereum transaction processing and mining functionality. Moreover, by default, there is no waiting period for the mining of transactions. This can be overridden using the Ganache configuration settings. The transactions are written as they are generated. It means developers can use `ganache-cli` as their Ethereum node and do not need mining activity to write transactions to a ledger. Instead, the transactions are stored in a ledger as they are created.

ganache-cli is dependent on Node.js, and it should be available on the machine before deploying ganache-cli. If Node.js is not installed, it can be downloaded from <https://nodejs.org/en/download/>. Based on processor architecture (32- or 64-bit) and an operating system, an appropriate package can be downloaded and installed from the given link, as shown in the following screenshot:

The screenshot shows the Node.js download page. At the top, there's a navigation bar with links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below the navigation bar, there are two main sections: 'LTS' (Recommended For Most Users) and 'Current' (Latest Features). Under 'LTS', there are links for 'Windows Installer' (node-v16.13.1-x86.msi), 'macOS Installer' (node-v16.13.1.pkg), and 'Source Code' (node-v16.13.1.tar.gz). Under 'Current', there are links for 'Windows Installer (.msi)', 'Windows Binary (.zip)', 'macOS Installer (.pkg)', 'macOS Binary (.tar.gz)', 'Linux Binaries (x64)', 'Linux Binaries (ARM)', and 'Source Code'. A large table below lists the available architectures for each platform: Windows (32-bit, 64-bit), macOS (32-bit, 64-bit / ARM64), Linux (64-bit, ARM64, ARMv7, ARMv8), and Source Code (node-v16.13.1.tar.gz).

	32-bit	64-bit
Windows Installer (.msi)		
Windows Binary (.zip)		
macOS Installer (.pkg)		64-bit / ARM64
macOS Binary (.tar.gz)	32-bit	64-bit
Linux Binaries (x64)		ARM64
Linux Binaries (ARM)	64-bit	ARMv7
Source Code		ARMv8
		node-v16.13.1.tar.gz

Figure 2.10 – The Node.js download page

Here, we have downloaded the 64-bit version of the MSI Windows installer, and both **node package manager (npm)** and Node.js are installed with it.

The Node.js version is v16.2.0, as shown in the following screenshot:

```
(base) riteshmodi@riteshm ~ % node --version
v16.2.0
```

Figure 2.11 – Validating the Node.js deployed version

The npm version is 7.13.0, as shown in the following screenshot:

```
[(base) riteshmodi@riteshm ~ % npm --version  
7.13.0
```

Figure 2.12 – Validating the npm deployed version

ganache-cli can be installed using the following `npm install` command:

```
npm install -g ganache
```

The preceding command line generates the following output:

```
[(base) riteshmodi@riteshm ~ % npm install -g ganache-cli  
changed 6 packages, and audited 102 packages in 5s  
2 packages are looking for funding  
  run `npm fund` for details  
9 vulnerabilities (8 moderate, 1 high)  
To address all issues, run:
```

Figure 2.13 – Installing ganache-cli

After the installation of ganache-cli, an Ethereum node based on it can be started using the following command:

```
ganache -p 9090
```

The port value is optional, and a different value from the default value is provided to ensure that there is no conflict if Geth with the default port is already executing in the environment. The result of executing the preceding command is that it creates 10 accounts by default, each having 100 ethers in balance with them, and it can be used in the same way any other private network is used, as shown in the following screenshot:

```

Ganache CLI v6.12.2 (ganache-core: 2.13.2)

Available Accounts
=====
(0) 0xdF7c15652A412096D39621B90A6A3c2E4dd2BfC7 (100 ETH)
(1) 0xfF3a6EF862c603D2E719d01c6F870ac6898d78BA (100 ETH)
(2) 0x3DF3c8909E53560e8070260DC41696eBF75C3130 (100 ETH)
(3) 0x065fA75721399e488C239fcfb298BB541FF6E9fd (100 ETH)
(4) 0x8169935288C94E68Fc0611DCc3E316f984bD0d1c (100 ETH)
(5) 0xa4c80FCBC682069b0054D9ecC7C4c8EEED139874 (100 ETH)
(6) 0x00FE52330a6C4e63Ad35D93653eA61afc1cbDb65 (100 ETH)
(7) 0x3e5F743AA4764512CCBF780cd05297d9AFCCc7a0 (100 ETH)
(8) 0x929F42aBc25537370d7df8d0cda69410f02bBE68 (100 ETH)
(9) 0x4152EeEdE975813c5d0FA9dF6a65aEFDaBB3a52 (100 ETH)

Private Keys
=====
(0) 0x81db4225116382ecbabbd0fc201099f7e9d4ad5438bd40a2c56d227f85a5827b2
(1) 0xbfdaf4e91b1ebd4142de629ee4d8c1f0a7a82ff332b78ae5cf5c5886b5f5b023
(2) 0x0691982fbf30491074a636b5b6044df17063da0841d1e526001c17021bdeb822
(3) 0x2a339d6525b3e54a90b5d5f27ca4514d9a3222cd95f637d36a27390bc6730644
(4) 0x542cb3654d2964345db4698dfa86a4cf3e329b4c8be63bac1920a94727523ba2
(5) 0xbbbe7e8701fcbb80036bf6c968a3c96609c0343cca23fe3e3a7f4fe66593ccb32
(6) 0xc4ef64ac2eab886d1fb1f38786d6fbe444f54aa2caa689cf9f86dc3d2661221b
(7) 0x847d18572d1de5d8e3f05b93c90d8d92f0c6340844f8bae65d6debbc1463a4fa
(8) 0xa69bee83cfdeda7d74924de692d8f8f512eb4985cbcdfa9e525aa4a4950e5913
(9) 0x72490f2eddbab183ba4d952081cde7d68c85b8553f28d5b8aa15b031b9c51913e

HD Wallet
=====
Mnemonic: share pole brown produce kingdom powder similar village midnight camera giraffe road
Base HD Path: m/44'/60'/0'/0/{account_index}

Gas Price
=====
20000000000

Gas Limit
=====
6721975

Call Gas Limit
=====
9007199254740991

Listening on 127.0.0.1:9090

```

Figure 2.14 – Running an instance of ganache-cli on a local machine on a custom port

Another command window can be attached to the Geth command line, as shown in the following screenshot:

```

(base) riteshmodi@riteshm ~ % geth attach http://127.0.0.1:9090
Welcome to the Geth JavaScript console!

instance: EthereumJS TestRPC/v2.13.2/ethereum-javascript
coinbase: 0xc4ac8e1c3c01be66c51a81b885873537850f9a0e
at block: 0 (Wed Dec 15 2021 21:11:47 GMT+0530 (IST))
modules: eth:1.0 evm:1.0 net:1.0 personal:1.0 rpc:1.0 web3:1.0

```

Figure 2.15 – Attaching a terminal to the Ganache instance on a custom port

Note that ganache-cli is running on port 9090, and so the same port is used to connect to an existing running instance of Ganache.

ganache-cli is an important tool for the development and testing of Ethereum-based decentralized applications. Smart contracts are core to decentralized applications, and they need to be compiled before deployment to the Ethereum network, and `solc` is one important compiler that helps to do that. The installation of `solc` is described next.

Installing the Solidity compiler

Solidity is one of the languages that is used to author smart contracts. Smart contracts will be dealt with in detail in the following chapters. The code written using Solidity is compiled using a Solidity compiler, which outputs byte code and other artifacts needed for the deployment of smart contracts. Earlier, Solidity was part of the Geth installation; however, it has moved out of Geth and should be deployed using its own installation.

The Solidity compiler, also known as `solc`, can be installed using `npm`:

```
npm install -g solc
```

The preceding command line generates the following output:

```
npm install -g solc
added 26 packages, and audited 27 packages in 3s
2 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Figure 2.16 – Installing the Solidity compiler

The Solidity compiler helps in compiling smart contracts, while another important framework known as `web3` is used for interacting with an existing Ethereum network. Next, we are going to look at installing and using the `web3` framework.

Installing the web3 framework

The web3 library is an open source JavaScript library that can be used to connect to Ethereum nodes from the same or a remote computer. It allows IPC as well as an RPC to connect to Ethereum nodes. web3 is a client-side library and can be used alongside a web page and query, and can submit transactions to Ethereum nodes. It can be installed using npm as a node module like the Solidity compiler. Let's take a look at the following steps to install the web3 JavaScript library:

1. The command used to install web3 is as follows:

```
npm install web3
```

The preceding command generates the following output:

```
((base) riteshmodi@riteshm ~ % npm install -g web3
npm WARN deprecated mkdirp-promise@5.0.1: This package is broken and no longer maintained. 'mkdirp' itself sup
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated multicodec@1.0.4: This module has been superseded by the multiformats module
npm WARN deprecated uuid@3.4.0: Please upgrade to version 7 or higher. Older versions may use Math.random()
.dev/blog/math-random for details.
npm WARN deprecated uuid@3.3.2: Please upgrade to version 7 or higher. Older versions may use Math.random()
.dev/blog/math-random for details.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues
npm WARN deprecated multibase@0.6.1: This module has been superseded by the multiformats module
npm WARN deprecated multibase@0.7.0: This module has been superseded by the multiformats module
npm WARN deprecated multicodec@0.5.7: This module has been superseded by the multiformats module
npm WARN deprecated cids@0.7.5: This module has been superseded by the multiformats module

added 376 packages, and audited 377 packages in 42s

57 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 2.17 – Installing the web3 JavaScript framework

2. After web3 is installed, it can be used using the Node.js REPL. From Command Prompt, enter the Node.js workspace by executing the node command, as shown in the following screenshot:

```
Last login: Wed Dec 15 21:11:52 on ttys004
((base) riteshmodi@riteshm ~ % node
Welcome to Node.js v16.2.0.
Type ".help" for more information.
> █
```

Figure 2.18 – Starting a node session

3. Once in the Node.js workspace, type the following commands to connect to an Ethereum node. The Ethereum node could be Ganache or a custom Geth-based private network. web3 can use WS, IPC, or RPC to connect to an Ethereum node. The following example shows the RPC endpoint protocol used to connect web3 to an Ethereum node:

```
const Web3 = require('web3')
const web3 = new Web3("http://127.0.0.1:8545")
```

The first command loads the web3 module, and the second command creates an instance of `HttpProvider` and connects to the locally hosted Ethereum node on port 8545.

Now, that we have created our own private network and started working with it, it is a good time to introduce MetaMask – the most used and prevalent wallet in the Ethereum ecosystem.

Installing and using MetaMask

MetaMask is a lightweight Chrome browser-based extension that helps in interacting with Ethereum networks. It is a wallet that helps in sending and receiving ether. MetaMask can be downloaded from <https://metamask.io/>. Since MetaMask runs in a browser, it does not download the entire chain data locally; instead, it uses the Infura gateway behind the scenes, which has the complete data stored locally and helps users connect to their store using the browser. Let's take a look at the following steps:

1. MetaMask should be added as an extension, as shown in the following screenshot:

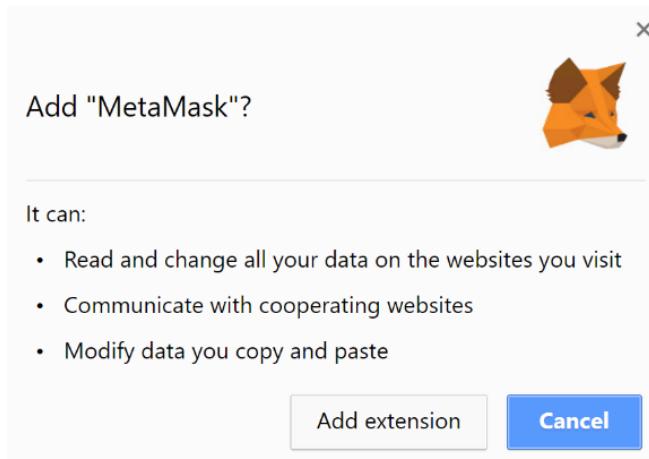


Figure 2.19 – Enabling MetaMask on the Chrome browser

- Accept the privacy notice and terms of use, and a small icon will appear next to the **Go** button. MetaMask allows you to connect to multiple networks. Connect to the **localhost 8545** private network, as shown in the following screenshot:

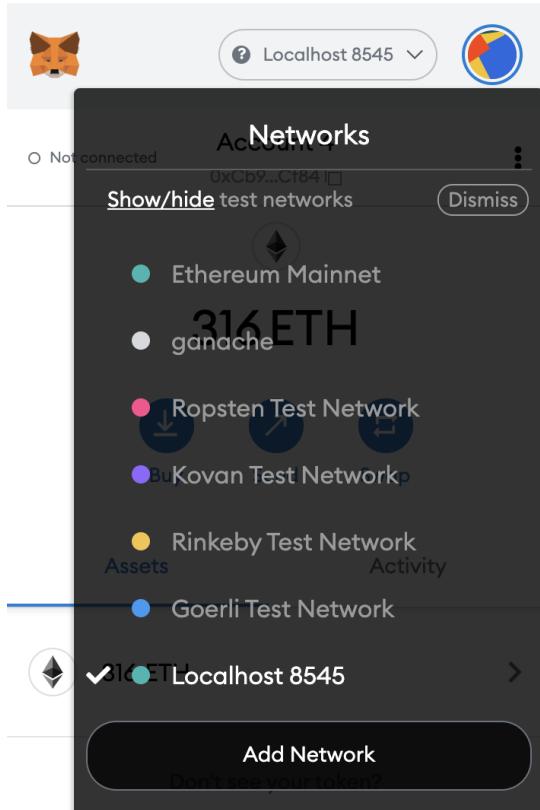


Figure 2.20 – Connecting MetaMask to an existing running instance of a private network

3. Provide a password to create a new key that is used by MetaMask to identify the user. It is stored in a key vault on the MetaMask central server, as shown in the following screenshot:

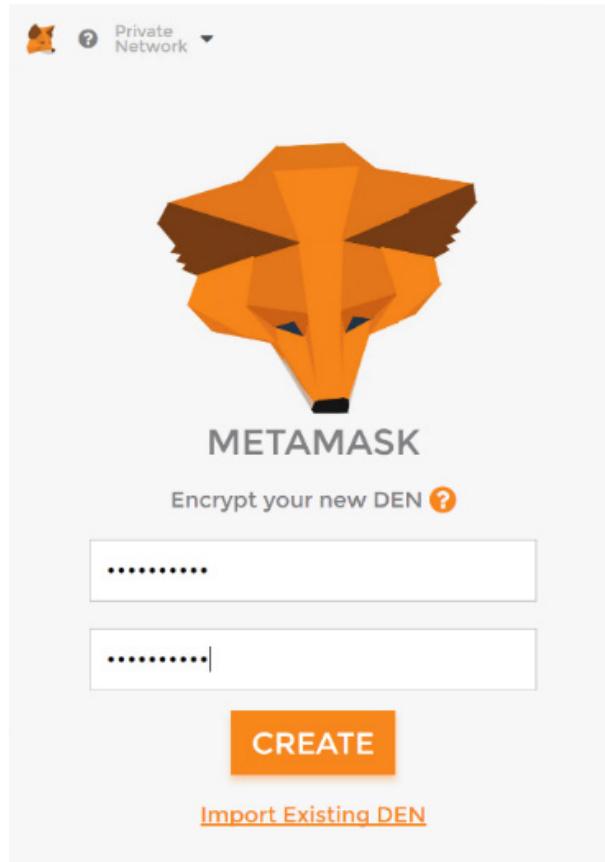


Figure 2.21 – Log in to the MetaMask wallet

- Click on the **Account** icon and import the accounts already created earlier using the **Import Account** menu in MetaMask. We need two accounts for transferring ethers from one account to another. If there is a single account in the environment, create a new account using the `personal.newAccount()` command, as shown previously:

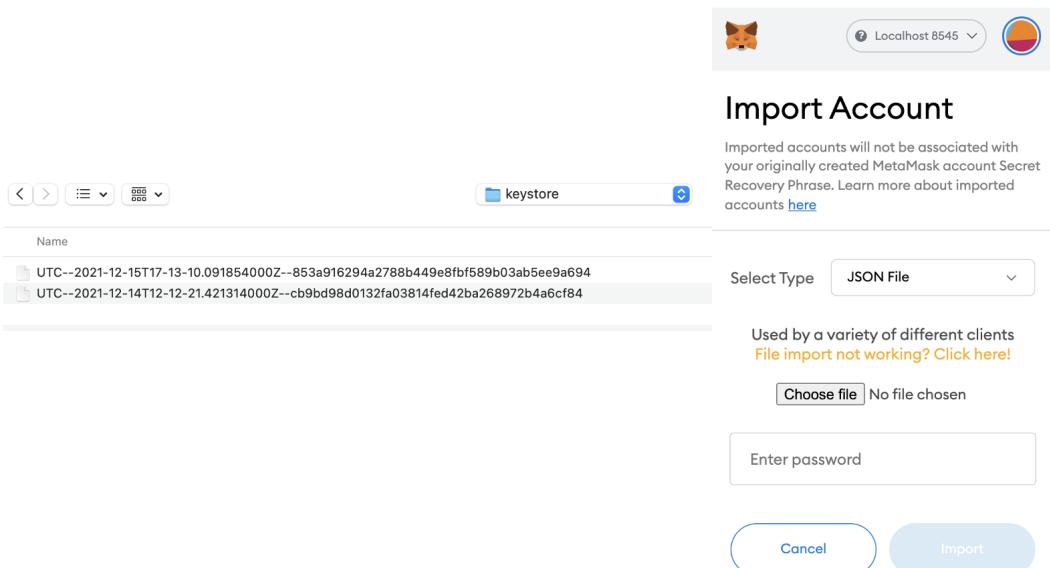


Figure 2.22 – Importing existing Geth accounts into MetaMask

5. After the accounts are imported, one account balance should be zero while another one, being a miner, will have some ether balance. We will transfer 100 ethers from one imported account to another using MetaMask:

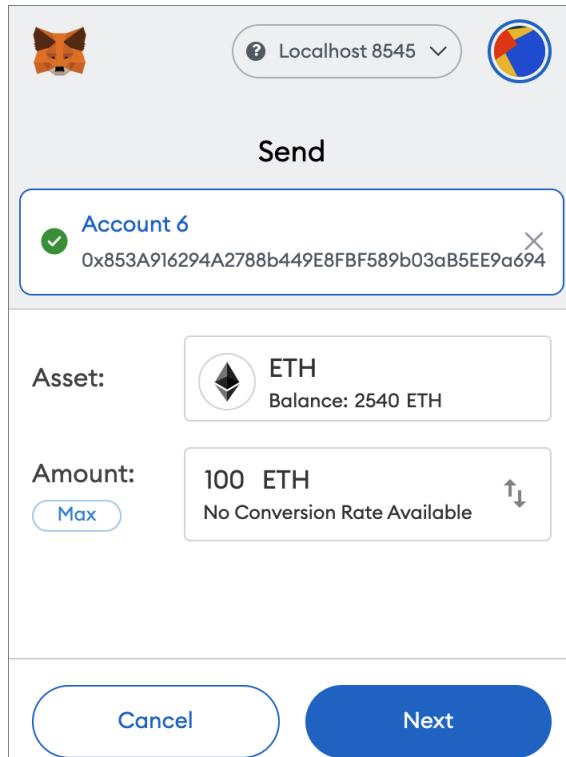


Figure 2.23 – Sending 100 ethers from a coinbase account to another account

To send ether from one account to another, select the imported account and click on the **Send** button. On the resultant window, provide a target account address and amount, and click on the **Next** and **Confirm** buttons one after another.

6. Submit the transaction. This will be in a pending state within the transaction pool. Mining should be started to write this transaction into permanent storage.

- Start mining using the Geth console, and the transaction will be mined, as shown in the following screenshot:

Figure 2.24 – A transaction initiated by MetaMask being processed by a private network

- After a while, the transaction is written in the ledger, and balances for both the accounts are updated in MetaMask:

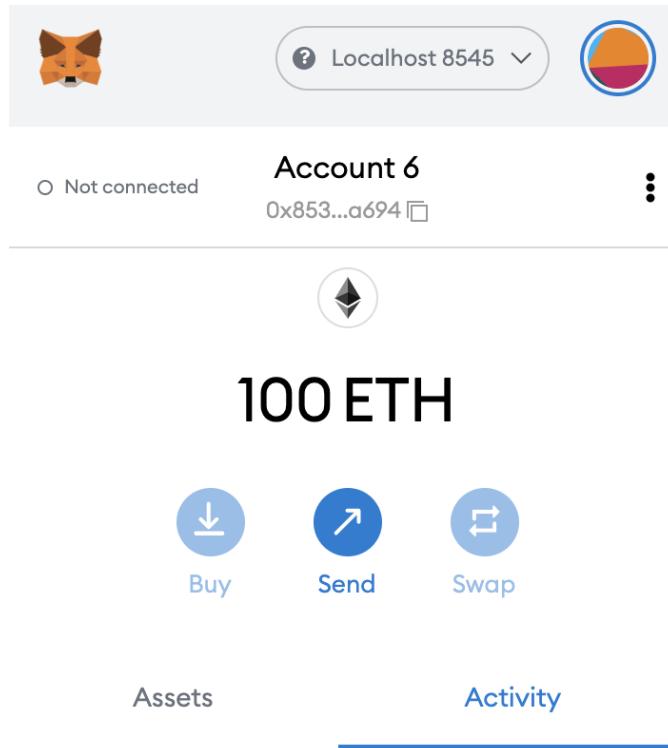


Figure 2.25 – The result of the transaction execution with the receiver balance increasing by 100 eth

If there are failures while sending the transaction from MetaMask to a private network, it could be because MetaMask is not configured with the same chain ID of the private network. Open the MetaMask network settings configuration and ensure that the chain ID has the same value used in `genesis.json`, as well as with the `Geth` command. This is shown in the following screenshot:

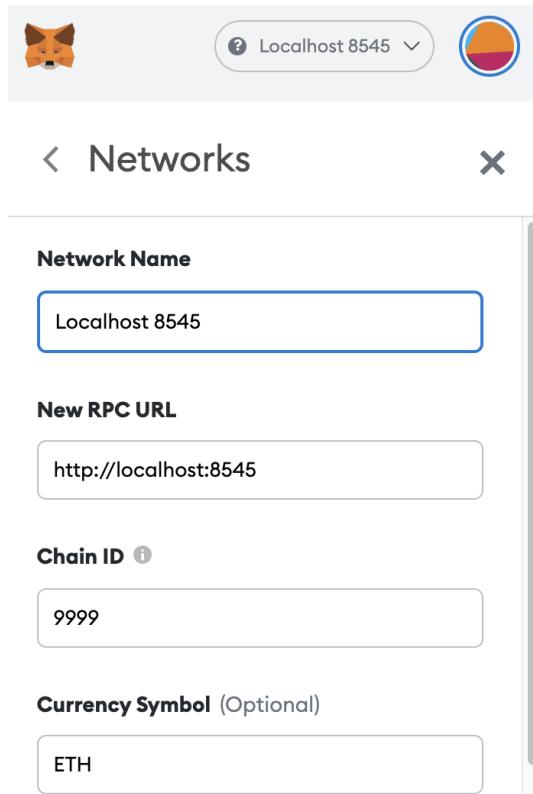


Figure 2.26 – Configuring MetaMask with the right chain ID for a private network

MetaMask is one of the leading software wallets used widely within the Ethereum ecosystem. It is used for storing ERC20-based tokens alongside assets such as NFTs. It is used for sending tokens to smart contracts as well as other externally owned accounts. With this, we conclude the deployment of important tools related to the Ethereum ecosystem, which we will use in subsequent chapters.

Summary

There was a lot of substance covered in this chapter. To give you a quick recap, we saw how Ethereum nodes implement JSON-RPC endpoints that can be connected using WS, IPC, and RPC. In this chapter, we discussed various forms of networks – public, main, test, and private. The chapter also discussed and implemented a private network. We followed steps to create a development environment that will be used in subsequent chapters. We also focused on deploying multiple tools and utilities on a Windows operating system. While each tool has its own workings and functionality, some tools might eventually do the same thing – for example, a Geth-based private chain and ganache-cli are essentially Ethereum nodes but with differences. The deployment of Geth, the Solidity compiler, ganache-cli, the web3 framework, and MetaMask were covered in this chapter. While some of you will like working with ganache-cli, others will be interested in using a private Geth-based Ethereum node. There is also another important utility known as Truffle, which will be covered in subsequent chapters.

In the next chapter, we will focus on Solidity as a language, which is the title of the book. Solidity supports object orientation, provides both native as well as complex data types, helps in declaring and defining functions that accept parameters and return values, provides control structures and expressions, and has many more features. The next chapter will discuss variables and data types in depth. Variables and data types are core to any programming language and more so in Solidity, since it has to store them within the distributed ledger.

Questions

1. How can a coinbase account be set with Geth?
2. What command is used to create a new account using Geth?
3. What is the purpose of MetaMask?
4. Why is ganache-cli useful?

Further reading

- Here's a link to the official site that provides details on installing Geth: <https://geth.ethereum.org/docs/install-and-build/installing-geth>.
- Here's a link to the official site that provides more details on options available inside Geth: <https://geth.ethereum.org/docs/interface/command-line-options>.

3

Introducing Solidity

From this chapter, we will embark on a journey – learning the Solidity language. The previous two chapters introduced blockchains, Ethereum, and related toolsets. Some important concepts related to blockchains, which are essential for having a better understanding and writing efficient code in Solidity, were also discussed. There are multiple languages that target the **Ethereum Virtual Machine (EVM)**. Some of them are deprecated and others are used with varying degrees of acceptance. Solidity is by far the most popular language for EVM. From this chapter onward, the book will focus on Solidity and its concepts, as well as constructs to help write efficient smart contracts.

In this chapter, we will jump right into understanding Solidity, and its structure, data types, and variables. We will cover the following topics in this chapter:

- Solidity and Solidity files
- The structure of a contract
- Data types in Solidity
- Storage and memory data locations
- Literals
- Integers
- Booleans
- The byte data type
- Arrays

- The structure of an array
- Enumeration
- The address type
- The mapping type

Technical requirements

There are not many technical requirements for this chapter. The following will suffice for this chapter:

- An operating system – Windows, macOS, and Linux work equally well
- A Chrome browser for navigation to `remix.org` and writing smart contracts

All code from the chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter03>

The Ethereum Virtual Machine

Solidity is a programming language targeting the EVM. Ethereum blockchain helps extend its functionality by writing and executing code known as smart contracts. We will get into the details of smart contracts in subsequent chapters, but for now, it is enough to know that smart contracts are similar to object-oriented classes written in Java or C++; however, they are not the same.

The EVM executes code that is part of smart contracts. Smart contracts are written in Solidity; however, the EVM does not understand the high-level constructs of Solidity, although it does understand lower-level instructions called **bytecode**.

Solidity code needs a compiler to take its code and convert it into bytecode that is understandable by the EVM. Solidity comes with a compiler to do this job, known as the Solidity compiler or `solc`. We downloaded and installed the Solidity compiler in the last chapter using the `Node.js npm` command.

The entire process is shown in the following diagram, from writing code in Solidity to executing it in the EVM:



Figure 3.1 – The process of executing smart contracts on the EVM

We already explored our first Solidity contract in the previous chapter when writing the HelloWorld contract, and now, we will get into the top-level constructs used within a Solidity contract.

Understanding Solidity and Solidity files

Solidity is a programming language that has similarities with JavaScript. Solidity is a statically-typed **Object-Oriented Programming (OOP)** language. Solidity is case-sensitive, and variables should use the same case during its definition and usage. The statement terminator in Solidity is the semicolon – ;.

Solidity code is written in Solidity files that have the .sol extension. They are human-readable text files that can be opened as text files in any editor, including Notepad.

A Solidity file is composed of the following four high-level constructs:

- Pragma
- Comments
- Import
- Contract/library/interface

Let's have a look at each in brief.

Pragma

Pragma is generally the first line of code within any Solidity file. It is a directive that specifies the target compiler version to be used for compiling the current Solidity file. Solidity is a new language and is subject to continuous improvement on an ongoing basis. Whenever a new feature or improvement is introduced, it comes out with a new version. The current version, at the time of writing, is 0.8.9. With the help of the pragma directive, you can choose the compiler version and target your code accordingly, as shown in the following code example:

```
pragma solidity ^0.8.9;
```

Although it is not mandatory, it is a good practice to declare the pragma directive as the first statement in a Solidity file. The syntax for the pragma directive is as follows:

```
pragma solidity <<version number>> ;
```

Also, note the case sensitivity of the directive. Both `pragma` and `solidity` are lowercase, with a valid version number and statement terminated with a semicolon. The version number comprises two numbers – a **major build** and a **minor build** number. The former in the preceding example is 8 and the latter is 9. Generally, there are fewer or no breaking changes within minor versions, but there can be significant changes between major versions. You should choose a version that best suits your requirements.

The `^` character, also known as a **caret**, is optional in version numbers but plays a significant role in deciding the version number based on the following rules:

- The `^` character, along with the version number, denotes the minimum version required by the contract – for example, a version with `0.8.9` means that the smart contract will compile with version `0.8.9` and above but not below.
- The `^` character ensures that the contract will compile for any version above the version mentioned but less than the next major version – for example, a version with `0.8.9` will not work with the `0.9.x` compiler version, but lesser than it.

As a good practice, it is better to compile Solidity code with an exact compiler version rather than using `^`. There are changes in the newer version that can deprecate your code while using `^` in `pragma` – for example, the `throw` statement got deprecated, and newer constructs such as `assert`, `require`, and `revert` were recommended for use in newer versions. You do not want to get surprised one day when your code starts behaving differently.

Comments

Any programming language provides the facility to comment code, and so does Solidity. There are the following three types of comments in Solidity:

- Single-line comments
- Multiline comments
- **Ethereum Natural Specification (Natspec)**

Single-line comments are denoted by a double forward slash (`//`), while multiline comments are denoted using `/*` and `*/`. Natspec has two formats – `///` for single-line and a combination of `/**` for the beginning and `*/` for the end of multiline comments. Natspec is used for documentation purposes, and it has its own specification. The entire specification is available at <https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format>.

Let's take a look at Solidity's comments in the following code:

```
// This is a single-line comment in Solidity
/* This is a multiline comment
In Solidity. Use this when multiple consecutive lines
Should be commented as a whole */
```

The pragma directive and comments are shown in the next code snippet:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// This is a single line comment in Solidity

/* This is a multi-line comment
   In solidity. Use this when multiple consecutive lines
Should be commented as a whole */

contract FirstContract {

}
```

Importing Solidity code

The `import` keyword helps import other Solidity files, and we can access its code within the current Solidity file. This helps us to write modular Solidity code by splitting code into multiple small logical files. The syntax for using `import` is as follows:

```
import <<filename>> ;
```

Filenames can be fully qualified explicit or implicit paths. The forward slash (/) is used for separating directories from other directories and files, while . is used to refer to the current directory and .. is used to refer to the parent directory. This is very similar to the Linux and Windows way of referring to a file in relative terms. A typical `import` statement is shown in the following code. Also, note the semicolon toward the end of the statement:

```
import 'CommonLibrary.sol';
```

Contracts

Apart from `pragma`, `import`, and comments, we can define contracts, libraries, and interfaces at a global level. We will explore contracts, libraries, and interfaces in depth in subsequent chapters. This chapter ensures that you understand that multiple contracts, libraries, and interfaces can be declared within the same Solidity file. The `library`, `contract`, and `interface` keywords shown in the next code listing are case-sensitive in nature:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// This is a single line comment in Solidity

/* This is a multi-line comment
   In solidity. Use this when multiple consecutive lines
Should be commented as a whole */

contract FirstContract {
}

contract SecondContract {
}

library StringLibrary {
}

library MathLibrary {
}

interface IBank{
}

interface IAccount {
```

Now that we understand the way to declare smart contracts, it's time to understand the elements declared within a contract.

The structure of a contract

The primary purpose of Solidity is to write smart contracts for Ethereum. Smart contracts are the basic unit of deployment and execution for EVMs. Although all chapters in this book are dedicated to writing and developing smart contracts, the basic structure of smart contracts is discussed in this section.

Technically, smart contracts have two major elements – variables and functions. There are multiple facets to both variables and functions in Solidity, and that is again something that will be discussed throughout this book. This section is about describing the general structure of a smart contract using the Solidity language.

A contract consists of the following multiple constructs:

- State variables
- Structure definitions
- Modifier definitions
- Event declarations
- Enumeration definitions
- Function definitions

The next code listing shows various elements available to a smart contract. Each of these elements will be discussed in depth throughout this book:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

//contract definition
contract generalStructure {
    //state variables
    int public stateIntVariable; // variable of integer type
    string stateStringVariable; //variable of string type
    address personIdentifier; // variable of address type
    MyStruct human; // variable of structure type
    bool constant hasIncome = true; //variable of constant
    nature

    //structure definition
    struct MyStruct {
```

```
        string name; //variable for type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array
        of unsigned integer
    }

//modifier declaration
modifier onlyBy() {
    if (msg.sender == personIdentifier) {
        _;
    }
}

// event declaration
event ageRead(address, int);

//enumeration declaration
enum gender {male, female}

//function definition
function getAge (address _personIdentifier) onlyBy()
payable external returns (uint) {

    human = MyStruct("Ritesh", 10, true, new uint[] (3));
    //using struct MyStruct

    gender _gender = gender.male; //using enum

    emit ageRead(personIdentifier, stateIntVariable);
    //raising event
}

}
```

State variables

Variables in programming parlance refer to a storage location that can contain values. The values associated with variables can be changed during runtime. The variable can be used at multiple places within the code, and they will all refer to the same value stored within it. Solidity provides two types of variables – state and memory variables. In this section, we will introduce state variables. Memory variables are declared within functions, and they will be covered in the subsequent chapter.

One of the most important aspects of Solidity contracts is state variables. The state variables are permanently stored in a blockchain/Ethereum global state by miners. Variables declared in a contract that is not within any function are called **state variables**. State variables store the current values, and they keep getting updated by function execution within the contract as part of the transaction. The allocated memory for a state variable is statically assigned, and it cannot change (the size of memory allocated) during the lifetime of the contract. Each state variable has a type that must be defined statically. The Solidity compiler must ascertain the memory allocation details for each state variable, and so the state variable data type must be declared.

State variables also have additional qualifiers associated with them. They can be any one of the following:

- **internal**: By default, the state variable has the `internal` qualifier if nothing is specified. This means that this variable can only be used within current contract functions and any contract that inherits from it. These variables cannot be accessed externally for modification; however, they can be read. An example of an internal state variable is as follows:

```
int internal StateVariable ;
```

- **private**: This qualifier is similar to `internal` with additional constraints. Private state variables can only be used in contracts containing them. They cannot be used even within derived contracts. An example of a private state variable is as follows:

```
int private privateStateVariable ;
```

- **public**: This qualifier enables external access to state variables. The Solidity compiler generates a getter function for each public state variable. An example of a public state variable is as follows:

```
int public stateIntVariable ;
```

- **constant**: This qualifier makes state variables immutable. The value must be assigned to the variable at declaration time. In fact, the compiler will replace references of this variable everywhere in code with the assigned value. An example of a constant state variable is as follows:

```
bool constant hasIncome = true;
```

As mentioned previously, each state variable has an associated data type. A data type helps us determine the memory requirements for the variable and ascertain the values that can be stored in them – for example, a state variable of the `uint8` type, also known as an **unsigned integer**, is allocated a predetermined memory size, and it can contain values ranging from 0 to 255. Any other value is regarded as a foreign value and is not acceptable by the compiler and runtime for storage in this variable.

Solidity provides the following multiple out-of-the-box data types:

- `bool`
- `uint/int`
- `bytes`
- `address`
- `mapping`
- `enum`
- `struct`
- `bytes/String`

Using `enum` and `struct`, it is possible to declare custom user-defined data types as well. Later in this chapter, a complete section has been dedicated to data types and variables.

Structure

Structures or structs help implement custom user-defined data types. A structure is a composite data type, consisting of multiple variables of different data types. They are very similar to contracts; however, they do not contain any logic within them. They consist of only variables.

There may be times when you want to store related data together. Suppose you want to store information about an employee, such as the employee's name, age, marriage status, and bank account numbers. To represent this, using these individual variables related to a single employee, a structure in Solidity can be declared using the `struct` keyword. The variables within a structure are defined within opening and closing brackets {}, as shown in the following code snippet:

```
//structure definition
struct MyStruct {
    string name; //variable for type string
    uint myAge; // variable of unsigned integer type
    bool isMarried; // variable of boolean type
    uint[] bankAccountsNumbers; // variable - dynamic array
    of unsigned integer
}
```

To create an instance of a structure, the following syntax is used. There is no need to explicitly use the `new` keyword. This can only be used to create an instance of contracts or arrays, as shown in the following code snippet.

```
human = MyStruct("Ritesh", 10, true, new uint [] (3));
//using struct MyStruct
```

Multiple instances of `struct` can be created in functions. Structs can contain arrays and the mapping variables, while mappings and arrays can store values of the `struct` type.

Modifiers

In Solidity, a modifier is always associated with a function. A modifier in programming languages refers to a construct that changes the behavior of code under execution. Since a modifier is associated with a function in Solidity, it has the power to change the behavior of functions that it is associated with. For an easy understanding of modifiers, think of them as a function that will be executed before the execution of the target function. Suppose you want to invoke the `getAge` function but, before executing it, you want to execute another function that could check the current state of the contract, the values of incoming parameters, the current value in state variables, and so on and, accordingly, decide whether the `getAge` target function should be executed. This helps in writing cleaner, smaller functions without cluttering them with validation and verification rules. Moreover, the modifier can be associated with multiple functions. This ensures cleaner, more readable, and more maintainable code.

A modifier is defined using the `modifier` keyword followed by the `modifier` identifier, any parameters it should accept, and then code within the `{ }` brackets. An underscore, `_`, in a modifier is a placeholder for executing the target function. You can think of this as the underscore being replaced by the target function inline. `payable` is an out-of-the-box modifier provided by Solidity, which, when applied to any function, allows that function to accept an Ether value.

A `modifier` keyword is declared at the contract level, as shown in the following code snippet:

```
//modifier declaration
modifier onlyBy() {
    if (msg.sender == personIdentifier) {
        _;
    }
}
```

As we can see, in the preceding code snippet, a modifier named `onlyBy()` is declared at the contract level. It checks the value of the incoming address using `msg.sender` with an address stored in the state variable. `msg.sender` is a new concept not yet explained, which you might not understand; we will cover this in-depth in the next chapter.

The modifier is associated with a `getAge` function, as shown in the following code snippet:

```
//function definition
function getAge (address _personIdentifier) onlyBy()
    payable external returns (uint) {
    human = MyStruct("Ritesh", 10, true, new uint[](3));
    //using struct MyStruct
    gender _gender = gender.male; //using enum
    emit ageRead(personIdentifier, stateIntVariable);
    //raising event
}
```

The `getAge` function can only be executed by an account that has the same address as that stored in the contract's `_personIdentifier` state variable. The function will not be executed if any other account tries to invoke it.

Note that anybody can invoke the `getAge` function, but successful execution will only happen for the address stored within the state variable.

Events

Solidity supports events. Events in Solidity are just like events in other programming languages. Events are fired from contracts so that anybody interested in them can trap/catch them and execute code in response. Events in Solidity are used primarily for informing the calling application about the current state of the contract by means of the logging facility of the EVM. They are used to notify applications about changes in contracts, and applications can use them to execute their dependent logic. Instead of applications, keep polling the contract for certain state changes; the contract can inform them by means of events.

Events are declared within the contract at the global level and invoked within its functions. An event is declared using the `event` keyword, followed by an identifier and parameter list, and terminated with a semicolon. The values in parameters can be used to log information or execute conditional logic. Event information and its values are stored as part of transactions within blocks. In the last chapter, while discussing the properties of a transaction in *Chapter 1, Introduction to Blockchain, Ethereum, and Smart Contracts*, a property named `LogsBloom` was introduced. Events raised as part of a transaction are stored within this property.

There is no need to explicitly provide parameter variables – only data types are sufficient, as shown in the following code listing:

```
// event declaration
event ageRead(address, int);
```

An event can be invoked from any function by its name and by passing the required parameters, as shown in the following code snippet:

```
function getAge (address _personIdentifier) onlyBy()
payable external returns (uint) {
    human = MyStruct("Ritesh", 10, true,new uint[] (3));
    //using struct MyStruct
    gender _gender = gender.male; //using enum
    emit ageRead(personIdentifier, stateIntVariable); //
    raising event
}
```

Enumeration

The `enum` keyword is used to declare enumerations. Enumerations help in declaring a custom user-defined data type in Solidity. `enum` consists of an enumeration list – a predetermined set of named constants.

Constant values within an `enum` can be explicitly converted into integers in Solidity. Each constant value gets an integer value, with the first one having a value of 0, and the value of each successive item increasing by 1.

An `enum` declaration uses the `enum` keyword, followed by an enumeration identifier and a list of enumeration values within the `{ }` brackets. Note that an `enum` declaration does not have a semicolon as its terminator and that there should be at least one member declared in the list.

An example of `enum` is as follows:

```
enum gender {male, female}
```

A variable of enumeration can be declared and assigned a value, as shown in the following code:

```
gender _gender = gender.male ;
```

It is not mandatory to define `enum` in a Solidity contract. `enum` should be defined if there is a constant list of items that do not change like the example shown previously. These become a good example for an `enum` declaration. They help make your code more readable and maintainable.

Functions

Functions are at the heart of Ethereum and Solidity. Ethereum maintains the current state of state variables and executes transactions to change values in state variables. When a function in a contract is called or invoked externally, it results in the creation of a transaction. Functions are the mechanism to read and write values from/to state variables. Functions are a unit of code that can be executed on-demand by calling it. Functions can accept parameters, execute their logic, and optionally return values to the caller. They can be named as well as being anonymous. Solidity provides named functions, with the possibility of only one unnamed function in a contract, called the fallback function. We will learn more about fallback functions later in the book.

The syntax for declaring functions in Solidity is as follows:

```
function getAge (address _personIdentifier) onlyBy()  
    payable external returns (uint) {  
}  
}
```

A function is declared using the `function` keyword followed by its identifier – `getAge`, in this case. It can accept multiple comma-separated parameters. The parameter identifiers are optional, but data types should be provided in the parameter list. Functions can have modifiers attached, such as `onlyBy()` in this case.

There are a couple of additional qualifiers that affect the behavior and execution of a function. Functions have visibility qualifiers that determine the visibility scope of a function. Functions can also return data, and this information is declared using the `returns` keyword, followed by a list of return parameters. Solidity can return multiple parameters.

Functions have visibility qualifiers associated with them, similar to state variables. The visibility of a function can be any one of the following:

- `public`: This visibility makes functions accessible directly from outside. They become part of the contract's interface and can be called both internally and externally.
- `internal`: By default, the state variable has an `internal` qualifier if nothing is specified. This means that this function can only be used within the current contract and any contract that inherits from it. These functions cannot be accessed from outside. They are not part of the contract's interface.
- `private`: Private functions can only be used in contracts declaring them. They cannot even be used within derived contracts. They are not part of the contract's interface.
- `external`: This visibility makes functions accessible directly externally, but not internally. These functions become part of the contract's interface.

Functions can also have the following additional qualifiers that change their behavior in terms of having the ability to change contract state variables:

- `constant`: These functions do not have the ability to modify the state of a blockchain. They can read the state variables and return to the caller, but they cannot modify any variable, invoke an event, create another contract, call other functions that can change state, and so on. Think of the `constant` functions as functions that can read and return current state variable values.

- **view**: These functions are aliases of constant functions.
- **pure**: Pure functions further constrain the ability of functions. These can neither read nor write – in short, they cannot access state variables. Functions that are declared with this qualifier should ensure that they will not access the current state and transaction variables.
- **payable**: Functions declared with the `payable` keyword have the ability to accept Ether from the caller. The call will fail if Ether is not provided by the sender. A function can only accept Ether if it is marked payable.

We will discuss the preceding qualifiers in detail in subsequent chapters.

Exploring data types in Solidity

Every value used in Solidity is of a type, which determines the kind of value it can store. This helps Solidity to use the value based on the rules of that type. Solidity is a statically typed language, and this ensures that the compiler knows the type during compilation itself. Solidity data types can broadly be classified into the following two types:

- Value types
- Reference types

These two types in Solidity differ, based on the way they are assigned to a variable and stored in the EVM. Assigning a variable to another variable can be done by creating a new copy or just by copying the reference. Value types maintain independent copies of variables, and changing the value in one variable does not affect the value in another variable. However, changing values in reference type variables ensures that anybody referring to those variables gets an update value.

Value types

A type is referred to as a value type if it holds the data (value) directly within the memory owned by it. These types have values stored with them, instead of elsewhere. The same is illustrated in the following diagram. In this example, a variable of the **unsigned integer (uint)** data type is declared with 13 as its data (value). The `a` variable has memory space allocated by the EVM, which is referred to as `0x123`, and this location has the 13 value stored. Accessing this variable will provide us with the 13 value directly.

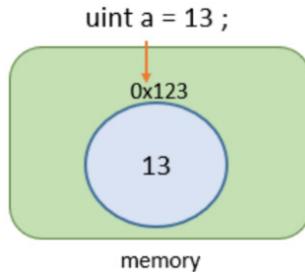


Figure 3.2 – Storage of a value type in Solidity

Value types are types that do not take more than 32 bytes of memory in size. Solidity provides the following value types:

- `bool`: The Boolean value that can hold `true` or `false` as its value.
- `uint`: These are unsigned integers that can hold 0 and positive values only.
- `int`: These are signed integers that can hold both negative and positive values.
- `address`: This represents an address of an account in the Ethereum environment.
- `byte`: This represents a fixed-sized byte array (`byte1` to `bytes32`).
- `enum`: Enumerations that can hold predefined constant values.

Passing by value

When a value type variable is assigned to another variable or when a value type variable is sent as an argument to a function, the EVM creates a new variable instance and copies the value of the original value type into the target variable. This is known as passing by value. Changing values in original or target variables will not affect the value in another variable. Both the variables will maintain their independent, isolated values, and they can change without the other knowing about it.

Reference types

Reference types, unlike value types, do not store their values directly within the variables themselves. Instead of the value, they store the address of the location where the value is stored. The variable holds the pointer to another memory location that holds the actual data. Reference types are types that can take more than 32 bytes of memory in size. Reference types will be shown in the following figure by means of a diagram.

In the following example, an array variable of the **uint** data type is declared with a size of 6. Arrays in Solidity are based at zero, so this array can hold seven elements. The `a` variable has memory space allocated by the EVM, which is referred to as `0x123`, and this location has a pointer value, `0x456`, stored in it. This pointer refers to the actual memory location where the array data is stored. When accessing the variable, the EVM dereferences the value of the pointer and shows the value from the array index, as shown in the following diagram:

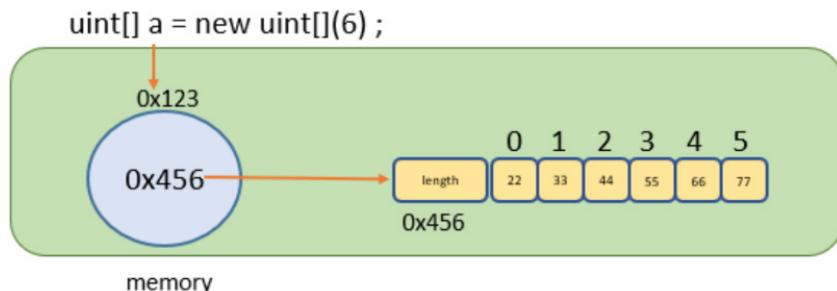


Figure 3.3 – Storage of reference types in Solidity

Solidity provides the following reference types:

- **Arrays:** These are dynamic arrays and include both string and byte arrays.
- **Structs:** These are custom, user-defined structures.
- **Mappings:** This is similar to a hash table or dictionary in other languages that store key-value pairs.

Passing by reference

When a reference type variable is assigned to another variable or a reference type variable is sent as an argument to a function, the EVM creates a new variable instance and copies the pointer from the original variable into the target variable. This is known as passing by reference. Both the variables are pointing to the same address location. Changing values in the original or target variables will change the value in other variables also. Both the variables will share the same values, and any change committed by one is reflected in the other variable.

Solidity variables can have storage or memory as their location, and after understanding value and reference types, it's time to understand Solidity variable data locations.

Storage and memory data locations

Each variable declared and used within a contract has a data location. The EVM provides the following four data structures for storing variables:

- **Storage:** This is global memory available to all functions within a contract. This storage is permanent storage that Ethereum stores on every node within its environment.
- **Memory:** This is local memory available to every function within a contract. This is short-lived and fleeting memory that gets torn down when a function completes its execution.
- **Calldata:** This is where all incoming function execution data, including function arguments, is stored. This is a non-modifiable memory location.
- **Stack:** The EVM maintains a stack for loading variables and intermediate values for working with the Ethereum instruction set. This is the working set memory for the EVM. A stack is 1,024 levels deep in the EVM, and if it stores anything more than this, it raises an exception.

The data location of a variable is dependent on the following two factors:

- The location of the variable declaration
- The data type of the variable

Based on the preceding two factors, there are rules that govern and decide the data location of a variable. The rules are mentioned in the following sub-sections. Data locations also affect the way the assignment operator works. Both assignment and data locations are explained by means of the rules that govern them.

Rule one

Variables declared as state variables are always stored in the storage data location.

Rule two

Variables declared as function parameters are either stored at a call data or memory data location.

Rule three

Variables declared within functions, by default, are stored in a memory data location. However, there are the following few caveats:

- The location for value type variables is memory within a function, while the memory location should be explicitly declared for a reference type variable.
- Reference type variables can be stored at the memory data location. The reference types referred to are arrays, structs, and strings.
- Reference types declared within a function with storage data location should always point to a state variable.
- Value type variables declared in a function cannot be overridden and stored at the storage location.
- Mappings are always declared as the state variable. This means that they cannot be declared within a function. They cannot be declared as memory types. However, mappings in a function can refer to mappings declared as state variables.

Rule four

Arguments supplied by callers to function parameters are either stored in call data or the memory data location.

Rule five

Assignments to a state variable from another state variable overwrite the value. Since there are two state variables already defined, there is neither a copy nor a change in reference. Two value type state variables, `stateVar1` and `stateVar2`, are declared. Within the `getUInt` function, `stateVar2` is assigned to `stateVar1`. At this stage, the values in both variables are 40. The next line of code changes the value of `stateVar2` to 50 and returns `stateVar1`. The returned value is 40, illustrating that each variable maintains its own independent value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract StorageToStorageValueTypeAssignment {
    uint stateVar1 = 20;
```

```
uint stateVar2 = 40;

// does not copy. these are two separate state variables.
The values are individual

function getUInt() public returns (uint)
{
    stateVar1 = stateVar2;

    stateVar2 = 50;
    return stateVar1; // returns 40
}

}
```

Two array type state variables, `stateArray1`, and `stateArray2`, are declared. Within the `getUInt` function, `stateArray2` is assigned to `stateArray1`. At this stage, the values in both variables are the same. The next line of code changes one of the values in `stateArray2` to 5 and returns the element at the same location as the `stateArray1` array. The returned value is 4, illustrating that each variable maintains its own independent value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract StorageToStorageReferenceTypeAssignment {
    uint [2] stateArray1 = [uint(1), 2];
    uint [2] stateArray2 = [uint(3), 4];

    // does not copy. these are two separate state variables.
    The values are individual

    function getUInt() public returns (uint)
    {
        stateArray1 = stateArray2;
        stateArray2[1] = 5;
        return stateArray1[1]; // returns 4
    }
}
```

Rule six

Assignments to storage variables from another memory variable always create a new copy. A fixed array of the `stateArray` uint is declared as a state variable. Within the `getUInt` function, a local memory-located fixed array of the `localArray` uint is defined and initialized. The next line of code assigns `localArray` to `stateArray`. At this stage, the values in both variables are the same. The next line of code changes one of the values in `localArray` to 10 and returns the element at the same location as the `stateArray[1]` array. The returned value is 2, illustrating that each variable maintains its own independent value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MemoryToStorageReferenceTypeAssignment {

    uint[2] stateArray ;

    function getUInt() public returns (uint)
    {
        uint[2] memory localArray = [uint(1), 2];
        stateArray = localArray;
        localArray[1] = 10;
        return stateArray[1]; // returns 2
    }
}
```

`stateVar` value type state variables are declared and initialized with a value of 20. Within the `getUInt` function, a `localVar` local variable is declared with a value of 40. In the next line of code, the `localVar` local variable is assigned to `stateVar`. At this stage, the values in both variables are 40. The next line of code changes the value of `localVar` to 50 and returns `stateVar`. The returned value is 40, illustrating that each variable maintains its own independent value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MemoryToStorageValueTypeAssignment {
```

```
uint stateVar = 20;

function getUInt() public returns (uint)
{
    uint localVar = 40;
    stateVar = localVar;
    localVar = 50;
    return stateVar; // returns 40
}
}
```

Rule seven

Assignments to a memory variable from another state variable always create a new copy. A value type state variable, `stateVar`, is declared and initialized with the value of 20. Within the `getUInt` function, a local variable of the `uint` type is declared and initiated with the value of 40. The `stateVar` variable is assigned to the `localVar` variable. At this stage, the values in both variables are 20. The next line of code changes the value of `stateVar` to 50 and returns `localVar`. The returned value is 20, illustrating that each variable maintains its own independent value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract StorageToMemoryValueTypeAssignment {
    uint stateVar = 20;

    function getUInt() public returns (uint)
    {
        uint localVar = 40;
        localVar = stateVar;
        stateVar = 50;
        return localVar; // returns 20
    }
}
```

A fixed array of the `stateArray` uint is declared as the state variable. Within the `getUInt` function, a local memory is located and the fixed array of uint, `localArray`, is defined and initialized with the `stateArray` variable. At this stage, the values in both variables are the same. The next line of code changes one of the values in `stateArray` to 5 and returns the element at the same location as the `localArray1` array. The returned value is 2, illustrating that each variable maintains its own independent value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract StorageToMemoryReferenceTypeAssignment {

    uint[2] stateArray3 = [uint(1), 2];

    function getUInt() public returns (uint)
    {
        uint[2] memory localArray = stateArray3;
        stateArray3[1] = 5;
        return localArray[1]; // returns 2
    }
}
```

Rule eight

Assignments to a memory variable from another memory variable do not create a copy for reference types; however, they do create a new copy for value types. The code listing shown in the following code listing illustrates that value type variables in memory are copied by value. The value of `localVar1` is not affected by the change in the value of the `localVar2` variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MemoryToMemoryValueTypeAssignment {
    function getUInt() public returns (uint)
    {
        uint localVar1 = 40;
```

```

        uint localVar2 = 80;
        localVar1 = localVar2;
        localVar2 = 100;
        return localVar1; // returns 80
    }
}

```

The code listing shown next illustrates that reference type variables in memory are copied by reference. The value of `otherVar` is affected by a change in the `someVar` variable:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MemoryToMemoryReferenceTypeAssignment {
    function getUInt() public returns (uint)
    {
        uint[] memory someVar = new uint[](1);
        someVar[0] = 23;
        uint[] memory otherVar = someVar;
        someVar[0] = 45;
        return (otherVar[0]); //returns 45
    }
}

```

This concludes the details about declaring and using storage and memory data locations within smart contracts. Now, it's time to move on to data types in smart contracts.

Using literals

Solidity provides the use of literals for assignments to variables. Literals do not have names; they are the values themselves. Variables can change their values during program execution, but a literal retains the same value throughout. Take a look at the following examples of various literals:

- Examples of integer literals are 1, 10, 1,000, -1, and -100.
- Examples of string literals are "Ritesh" and 'Modi'. String literals can be in single or double quotation marks.

- Examples of address literals are `0xca35b7d915458ef540ade6068dfe2f44e8fa733c` and `0x11111111111111111111111111111111`.
- Hexadecimal literals are prefixed with the `hex` keyword. An example of a hexadecimal literal is `hex"1A2B3F"`.

Solidity supports decimal literals with the use of the `.aa` dot – examples include `4 . 5` and `0 . 2`.

This concludes the details about declaring and using literal values within smart contracts. Now, it's time to move to integer data types in smart contracts.

Understanding integers

Integers help in storing numbers in contracts. Solidity provides the following two types of integer:

- **Signed integers:** Signed integers can hold both negative and positive values.
- **Unsigned integers:** Unsigned integers can hold positive values along with zero. They can also hold negative values apart from positive and zero.

There are multiple flavors of integers in Solidity for each of these types. Solidity provides the `uint8` type to represent an 8-bit unsigned integer and thereon in multiples of 8 till it reaches 256. In short, there can be 32 different declarations of `uint` with different multiples of 8, such as `uint8`, `uint16`, and `unit24`, as far as the `uint256` bit. Similarly, there are equivalent data types for integers, such as `int8` and `int16`, up to `int256`.

Depending on requirements, an appropriately sized integer should be chosen – for example, when storing values between 0 and 255, `uint8` is appropriate, and when storing values between -128 to 127, `int8` is more suitable. For higher values, larger integers can be used.

The default value for both signed and unsigned integers is zero, to which they are initialized automatically at the time of declaration. Integers are value types; however, when used as an array, they are referred to as reference types.

Mathematical operations such as addition, subtraction, multiplication, division, exponential, negation, post-increment, and pre-increment can be performed on integers. The following code listing shows some of these examples:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
```

```
contract AllAboutInts {  
  
    uint stateUInt = 20; //state variable  
    uint stateInt = 20; //state variable  
  
    function UIntAssignment(uint incomingValue) public  
    {  
        uint memoryuint = 256;  
        uint256 memoryuint256 = 256;  
        uint8 memoruint8 = 8; //can store value from 0 up to  
        255  
  
        //addition of two uint8  
        uint256 result = memoruint8 + memoryuint;  
    }  
  
    function IntAssignment(int incomingValue) public  
    {  
        int memoryInt = 256;  
        int256 memoryInt256 = 256;  
        int8 memoryInt8 = 8; //can store value from -128 to 127  
    }  
}
```

This concludes the details about declaring and using integers within smart contracts. Now, it's time to move to another data type that is very important for storing Boolean values in smart contracts.

Understanding Boolean

Solidity, as with any programming language, provides a Boolean data type. The `bool` data type can be used to represent scenarios that have binary results, such as `true` or `false`, and `1` or `0`. The valid values for this data type are `true` and `false`. Note that `bool` data types in Solidity cannot be converted to integers as they can in other programming languages. It's a value type, and any assignment to other Boolean variables creates a new copy. The default value for `bool` in Solidity is `false`.

A `bool` data type is declared and assigned a value, as shown in the following code:

```
bool isPaid = true;
```

It can be modified within contracts and can be used in both incoming and outgoing parameters and the return value, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract BoolContract {
    bool isPaid = true;

    function manageBool() public returns (bool) {
        isPaid = false;
        return isPaid; //returns false
    }

    function convertToInt() public returns (uint8) {
        isPaid = false;
        return uint8(isPaid); //error
    }
}
```

This concludes the details about declaring and using Boolean values within smart contracts. Now, it's time to move to another data type that is very important for storing values in bytes in smart contracts.

The byte data type

A byte refers to 8-bit signed integers. Everything in memory is stored in bits consisting of binary values – 0 and 1. Solidity also provides the byte data type to store information in binary format. Generally, programming languages have a single data type for representing bytes. However, Solidity has multiple flavors of the byte type. It provides data types that range from `bytes1` to `bytes32` inclusive, representing varying byte lengths, as required. These are called **fixed-sized byte arrays** and are implemented as value types. The `bytes1` data type represents 1 byte, and `bytes2` represents 2 bytes. The default value for byte is `0x00`, and it gets initialized with this value. Solidity also has a `byte` type that is an alias of `bytes1`.

A byte can be assigned byte values in hexadecimal format, as follows:

```
bytes1 aa = 0x65;
```

A byte can be assigned integer values in decimal format, as follows:

```
bytes1 bb = 10;
```

A byte can be assigned negative integer values in decimal format, as follows:

```
bytes1 ee = -100;
```

A byte can be assigned character values, as follows:

```
bytes1 dd = 'a';
```

In the following code snippet, a value of 256 cannot fit in a single byte and needs a bigger byte array:

```
bytes2 cc = 256;
```

The code listing in the following code listing shows how to store binary, positive, and negative integers, and character literals in fixed-sized byte arrays.

We can also perform bitwise operations such as `and`, `or`, `xor`, and `not`, and left and right shift operations on the `byte` data type:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract BytesContract {
    bytes1 aa = 0x65;
    bytes1 bb = bytes1(uint8(10));
    bytes2 cc = bytes2(uint16(256));
    bytes1 dd = 'a';
    bytes1 ff;

    function getintff() public returns (bytes1) {
        return ff; //returns 0x00
    }
    function getintaa() public returns (uint) {
```

```
        return uint8(aa); //returns 101
    }
    function getbyteaa() public returns (bytes1) {
        return aa; //returns 0x65
    }
    function getbytebb() public returns (bytes1) {
        return bb; //returns 0x0a
    }
    function getintbb() public returns (uint) {
        return uint8(bb); //returns 10
    }
    function getbytecc() public returns (bytes2) {
        return cc; //returns 0x0100
    }
    function getintcc() public returns (uint) {
        return uint16(cc); //returns 256
    }
    function getbytedd() public returns (bytes2) {
        return dd; //returns 0x6100 or 0x61 for bytes1
    }
    function getintdd() public returns (uint) {
        return uint8(dd); //returns 97
    }
}
```

This concludes the details about declaring and using `byte` data type within smart contracts. Now, it's time to move to another data type that is very important for storing multiple values of the same type in smart contracts using arrays.

Understanding arrays

Arrays are discussed as data types, but more specifically, they are data structures that are dependent on other data types. Arrays refer to groups of values of the same type. Arrays help in storing these values together and ease the process of iterating, sorting, and searching for individuals or subsets of elements within this group. Solidity provides a rich array of constructs that cater to different needs. Each element within the set gets assigned memory in multiple of 32 bytes. For example an array of `uint256` containing 5 elements would have each item stored in 32 bytes.

An example of an array in Solidity is as follows:

```
uint [5] intArray;
```

Arrays in Solidity can be fixed or dynamic.

Fixed arrays

Fixed arrays refer to arrays that have a pre-determined size mentioned at the declaration. Examples of fixed arrays are as follows:

```
int [5] age; // array of int with 5 fixed storage space  
allocated  
byte [4] flags; // array of byte with 4 fixed storage space  
allocated
```

Fixed arrays cannot be initialized using the new keyword. They can only be initialized inline, as shown in the following code:

```
int [5] age = [int(10), 20, 30, 40, 50];
```

They can also be initialized inline within a function later, as follows:

```
int [5] age;  
age = [int(10), 2, 3, 4, 5];
```

Dynamic arrays

Dynamic arrays refer to arrays that do not have a pre-determined size at the time of declaration; however, their size is determined at runtime. Take a look at the following code:

```
int [] age; // array of int with no fixed storage space  
allocated. Storage is allocated during assignment  
byte [] flags; // array of byte with no fixed storage space  
allocated
```

Dynamic arrays can be initialized inline or by using the new operator. The initialization can happen at the time of declaration, as follows:

```
int [] age = [int(10), 20, 30, 40, 50];  
int [] age = new int [] (5);
```

The initialization can also happen within a function later in the following two different steps:

```
int [] age;
age = new int [] (5);
```

Dynamic arrays provides special methods that help in managing them. Items can be pushed into a dynamic array using its push method. The push method accepts an element and that gets added to the end of dynamic array:

```
uint256 [] myarray;

function addtoarray(uint256 _number) public returns (uint256,
uint256) {
    myarray.push(_number);
    return (_number, myarray.length);
}
```

It also provides a length property that returns back the number of elements currently available within the array. The length is not zero based. It starts from one. The length is zero for an empty array:

```
uint256 [] myarray;

function arraylength() public view returns (uint256) {
    return myarray.length;
}
```

Items can be popped or removed from dynamic array using its pop method. The pop method removes the last item from the array and it also adjust the length:

```
uint256 [] myarray;

function popValue() public {
    myarray.pop();
}
```

And obviously items from within all arrays including static arrays can be retrieved by using indexing on the array. The index is zero based. It means if there are 10 items in the array, index ranging from 0 to 9 provides access to each of them:

```
uint256 [] myarray;

function getarrayvalue(uint256 _index) public view returns
(uint256) {
    return myarray[_index];
}
```

Elements within an array (static as well as dynamic) can be assigned their default values (zero for int family of data types) by deleting them. Deleting does not remove the element from the array neither it reduces its length. It just assigns default data type value to it:

```
uint256 [] myarray;

function deleteItem(uint256 _index) public {
    delete myarray[_index];
}
```

Special arrays

Solidity provides the following two special arrays:

- The bytes array
- The String array

The bytes array

The `bytes` array is a dynamic array that can hold any number of bytes. It is not the same as `byte []`. The `byte []` array takes 32 bytes for each element, whereas `bytes` tightly hold all the bytes together. Bytes might take less than 32 bytes for each element.

Bytes can be declared as a state variable with an initial length size, as shown in the following code:

```
bytes localBytes = new bytes(0);
```

This can be also divided into the following two code lines, similar to the previously discussed arrays:

```
bytes localBytes ;  
localBytes = new bytes (10);
```

Bytes can be assigned values directly, as follows:

```
localBytes = "Ritesh Modi";
```

Also, values can be pushed into them, as shown in the following code, if they are located at the storage location:

```
localBytes.push(byte(10));
```

Bytes also provide a read/write `length` property, as follows:

```
return localBytes.length; //reading the length property
```

Take a look at the following code as well:

```
localBytes.length = 4; //setting bytes length to 4 bytes
```

The String array

Strings are dynamic data types that are based on `bytes` arrays, as discussed in the previous section. They are very similar to `bytes` with additional constraints. Strings cannot be indexed or pushed and do not have the `length` property. To perform any of these actions on string variables, they should first be converted into `bytes` and then converted back to strings after the operation.

Strings can be composed of characters within single or double quotation marks.

Strings can be declared and assigned values directly, as follows:

```
String name = 'Ritesh Modi';
```

They can also be converted to `bytes`, as follows:

```
Bytes byteName = bytes(name);
```

Solidity unlike other programming languages do not have any base framework to depend on utility functions. String type in Solidity do not provide string manipulation methods. One of the common task related to string is to compare them for equality. Instead of writing custom code to compare strings, it is easier to convert each of them to their equivalent hash and compare the hashes. Solidity provides keccak256 function that generates hash of its argument (this function is discussed in later chapters). String values can be used as arguments to this function as shown next.

```
Keccak256(''ritesh modi'') == keccak256(''ritesh modi'')
```

Array properties

There are basic properties supported by arrays. In Solidity, due to the multiple types of array, not every type supports all of these properties.

These properties are as follows:

- `index`: This property used for reading individual array elements is supported by all types of arrays, except for the `string` type. The `index` property for writing to an individual array element is supported for dynamic arrays, fixed arrays, and the `bytes` type only. Writing is not supported for string and fixed-sized byte arrays.
- `push`: This property is supported by dynamic arrays only.
- `length`: This property is supported by all arrays from a read perspective, except for the `string` type. Only dynamic arrays and bytes support modifying the `length` property.

Knowing more about the structure of an array

We have already briefly touched on the topic of structures. Structures help in defining custom user-defined data structures. Structures help to group a multiple variables of different data types into a single type. A structure does not contain any programming logic or code for execution; it just contains a variable declaration. Structures are reference types and treated as complex types in Solidity.

Structures can be defined as state variables, as shown in the next code snippet. A struct composed of the `string`, `uint`, `bool`, and `uint` arrays is defined. There are two state variables, which are in the storage location. While the first `stateStructure1` state variable is initialized at the time of declaration, the other `stateStructure1` state variable is left to be initialized later within a function.

A local structure at the memory location is declared and initialized within the `getAge` function. Another structure is declared that acts as a pointer to the `stateStructure` state variable. A third local structure is declared that refers to the previously created `localStructure` local structure.

A change in one of the properties of `localStructure` is performed while the previously declared state structure is initialized, and finally, the age from `pointerLocalStructure` is returned:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

//contract definition
contract generalStructure {
    //structure definition
    struct MyStruct {
        string name; //variable for type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array
        of unsigned integer
    }
    MyStruct stateStructure = MyStruct("Ritesh", 10, true, new
        uint[](2));
    MyStruct stateStructure1;
    function getAge () public returns (uint) {
        // local structure
        MyStruct memory localStructure = MyStruct("Modi", 20
            ,false, new uint[](2));
        //local pointer to State structure
        MyStruct memory pointerStructure = stateStructure;
        // pointerlocalStructure is reference to localStructure
        MyStruct memory pointerlocalStructure = localStructure;
        //changing value in localStructure
        localStructure.myAge = 30;
        //assigning values to state variable
        stateStructure1 = MyStruct("Ritesh", 10, true, new
            uint[](2));
    }
}
```

```
//returning pointerlocalStructure.Age -- returns 30
    return pointerlocalStructure.myAge;
}
}
```

It returns the new value that was assigned to `localStructure`, as shown in the preceding code listing.

Enumerations

We briefly touched on the concept of enumerations while discussing the layout of the Solidity file earlier in this chapter. Enums are value types comprising a pre-defined list of constant values. They are passed by values, and each copy maintains its own value. Enums cannot be declared within functions and are declared within the global namespace of a contract.

Predefined constants are assigned consecutively, increasing integer values starting from zero.

The following code snippet declares `enum`, identified as a status consisting of five constant values – `CREATED`, `APPROVED`, `PROVISIONED`, `REJECTED`, and `DELETED`. They have the 0, 1, 2, 3, 4 integer values assigned to them.

An instance of `enum` named `status` is created with an initial value of `REJECTED`.

The `getChoice` function returns the current status in terms of the `enum` value. Note that `web3` and **Decentralized Applications (DApp)** do not understand an `enum` value declared within a contract. They will get an integer value corresponding to the `enum` constant.

The `set*` functions set the value of the status to a different `enum` value, and `getDefaultChoice` returns the integer representation of the `enum` value. The next code listing shows a way to declare and use enums within smart contracts:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EnumExample {
    enum VMStatus{
        CREATED, APPROVED, PROVISIONED, REJECTED, DELETED
    }
}
```

```
VMStatus status;
VMStatus defaultChoice = VMStatus.REJECTED;

function setCreated() public {
    status = VMStatus.APPROVED;
}

function setApproved() public {
    status = VMStatus.APPROVED;
}

function setProvisioned() public {
    status = VMStatus.APPROVED;
}

function setRejected() public {
    status = VMStatus.REJECTED;
}

function setDeleted() public {
    status = VMStatus.REJECTED;
}

function getChoice() public view returns (VMStatus) {
    return status;
}

function getDefaultChoice() public returns (uint) {
    return uint(defaultChoice);
}
```

This concludes the details about declaring and using enums within smart contracts. Now, it's time to move to another data type that is very important for storing Ethereum addresses in smart contracts.

Understanding the address data type

An address is a 20 bytes-long data type. It is specifically designed to hold account addresses in Ethereum, which are 160 bits or 20 bytes in size. It can hold contract addresses as well as externally owned account addresses. The address is a value type, and it creates a new copy while being assigned to another variable.

There are two variations for the address type:

- `address`: This is the general address type that can hold Ethereum addresses as part of a contract. It cannot be used to send or receive Ethers. They are meant for address management within smart contracts but cannot be used for receiving and sending Ether as part of smart contracts.
- `Payable address`: These are similar to the `address` type with the additional capability of receiving as well as sending Ether to other accounts. It has additional methods, `send()` and `transfer()`. This function usage will be covered in subsequent chapters. Since the payable address is a superset of the `address` type, it can implicitly be converted into a simple address; however, the reverse needs an explicit conversion.

The `address` type has a `balance` property that returns the amount of Ether available with the account and has a few functions for invoking functions in other contracts.

It also provides the following three functions for invoking the `contract` function:

- `Call`
- `DelegateCall`
- `Callcode`

The payable address provides the following two functions to transfer Ether:

- `transfer`
- `send`

The `transfer` function is a better alternative for transferring Ether to an account than the `send` function. The `send` function returns a Boolean value, depending on the successful execution of the Ether transfer, while the `transfer` function raises an exception and returns the Ether to the caller.

Again, these concepts will be explained using code in the subsequent chapter, since some of the concepts related to transferring Ether are not yet covered.

Working with mappings

Mappings are one of the most widely used complex data types in Solidity. Mappings are similar to hash tables or dictionaries in other languages. They help in storing key-value pairs and enable the retrieval of values based on a supplied key.

Mappings are declared using the `mapping` keyword, followed by data types for both keys and values separated by the `=>` notation. Mappings have identifiers, as with any other data type, and they can be used to access the mapping.

An example of a mapping is as follows:

```
Mapping ( uint => address ) Names ;
```

In the preceding code, the `uint` data type is used for storing the keys, and the `address` data type is used for storing the values. `Names` is used as an identifier for the mapping.

Although it is similar to a hash table and dictionary, Solidity does not allow iteration through mapping. A value from mapping can be retrieved if the key is known. The next example illustrates working with mapping. A counter of the `uint` type is maintained in a contract that acts as a key, and address details are stored and retrieved with the help of functions.

To access any particular value in mapping, the associated key should be used along with the mapping name, as shown here:

```
Names [counter]
```

To store a value in a mapping, use the following syntax:

```
Names [counter] = <<some value>>
```

Take a look at the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract GeneralMapping {
    mapping (uint => address) Names;
    uint counter;
```

```
function addtoMapping(address addressDetails) public
    returns (uint)
{
    counter = counter + 1;
    Names[counter] = addressDetails;

    return counter; //returns false
}

function getMappingMember(uint id) public view returns
(address)
{
    return Names[id];
}
```

Although mapping doesn't support iteration, there are ways to work around this limitation. The next example illustrates one of the ways to iterate through mapping. Please note that iterating and looping are expensive operations in Ethereum in terms of gas usage and should generally be avoided. In this example, a separate counter is maintained to keep track of the number of entries stored within the mapping. This counter also acts as the key within the mapping. A local array can be constructed for storing the values from mapping. A loop can be executed using counter and can extract and store each value from the mapping in the local array, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MappingLooping {

    mapping (uint => address) Names;
    uint counter;

    function addtoMapping(address addressDetails) public
        returns (uint)
    {
        counter = counter + 1;
```

```
Names[counter] = addressDetails;

return counter;
}

function getMappingMember() public view returns (address[]
memory) {
address[] memory localBytes = new address[](counter);
for(uint i=1; i<= counter; i++){
localBytes[i - 1] = Names[i];
}

return localBytes;
}
}
```

A mapping can only be declared as a state variable whose memory location is of the storage type. A mapping cannot be declared within functions as memory mappings. However, mappings can be declared in functions if they refer to mappings declared in state variables, as shown in the following example:

```
Mapping (uint => address) localNames = Names;
```

This is valid syntax, as the `localNames` mapping is referring to the `Names` state variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MappinginMemory {
mapping (uint => address) Names;
uint counter;

function addtoMapping(address addressDetails) public
returns (uint) {
counter = counter + 1;
mapping (uint => address) storage localNames = Names;

localNames[counter] = addressDetails;
```

```

        return counter;
    }

    function getMappingMember(uint id) public view returns
        (address) {
        return Names[id];
    }
}

```

It is also possible to have nested mapping – that is, mapping consisting of mappings. The next example illustrates this. In this example, there is an apparent mapping that maps uint to another mapping. The child mapping is stored as a value for the first mapping. The child mapping has the address type as the key and the string type as the value. There is a single mapping identifier, and the child or inner mapping can be accessed using this identifier itself, as shown in the following code:

```
mapping (uint => mapping(address => string)) accountDetails;
```

To add an entry to this type of nested mapping, the following syntax can be used:

```
accountDetails[counter][addressDetails] = names;
```

Here, `accountDetails` is the mapping identifier and `counter` is the key for parent mapping. The `accountDetails [counter]` mapping identifier retrieves the value from the parent mapping, which in turn happens to be another mapping. Adding the key to the returned value, we can set the value for the inner mapping. Similarly, the value from the inner mapping can be retrieved using the following syntax:

```
accountDetails[counter][addressDetails]
```

Take a look at the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract DemoInnerMapping {
    mapping (uint => mapping(address => string))
        accountDetails;
    uint counter;
}
```

```
function addtoMapping(address addressDetails, bytes memory
    name) public returns (uint) {
    string memory names = string(name);
    counter = counter + 1;
    accountDetails[counter][addressDetails] = names;
    return counter;
}

function getMappingMember(address addressDetails) public
    view returns (bytes memory) {
    // 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
    return bytes( accountDetails[counter][addressDetails]);
}
```

By now, we have covered most of the important concepts and types of variables and data types available in Solidity, including mappings, data locations, types of variables, and data types. Mappings, arrays, and structures are the most important composite types used for writing meaningful smart contracts.

Summary

This is the first chapter that has explored Solidity in depth. This chapter introduced Solidity and the layout of Solidity files, including elements that can be declared at the top level in it. Constructs such as pragma, contracts, and elements of contracts were discussed from a layout perspective. A complete immersion into the world of Solidity data types forms the core of this chapter. Value types and reference types were discussed in depth, and types such as `int`, `uint`, fixed-sized byte arrays, `bytes`, arrays, strings, structures, enumerations, addresses, Booleans, and mappings were discussed in great length along with examples. Solidity provides additional data locations from complex types such as structs and arrays, which were also discussed in depth along with the rules that govern their usage. Solidity variables and their types are foundational concepts for writing smart contracts, and this knowledge will help you to write functional ones.

In the next chapter, we will focus on using some out-of-the-box variables and functions of smart contracts. Solidity provides numerous global variables and functions to help ease the task of obtaining the current transaction and block context. These variables and functions provide contextual information and Solidity code can utilize them for logic execution. They play a very important role in authoring enterprise-scale smart contracts.

Questions

1. You are writing a smart contract that stores employee data, such as the employee ID, as well as their name and age. The contract will store information about multiple employees. You want to retrieve the employees using their employee ID without iterating through all the employees. Which data type should be used to store the employees?
2. Can a mapping be defined within a `contract` function?
3. Which data types are referred to as reference types?

Further reading

- The Solidity docs have a good introduction on types, available here: <https://docs.soliditylang.org/en/v0.8.9/types.html>

4

Global Variables and Functions

In *Chapter 3, Introducing Solidity*, you learned about Solidity data types in detail. Data types can be of values or reference types. Some reference types such as structs and arrays also have data locations – memory and storage associated with them. Variables can be state variables or variables defined locally within functions. This chapter will focus on variables, their scoping rules, declaration and initialization, conversion rules, and variables available globally to all contracts. Some of the important global functions will be discussed in this chapter, while others will be discussed in subsequent chapters. These global variables are functions that are very important while authoring smart contracts. Global variables provide runtime information about the current execution of the smart contract and internal state, while global functions help write advanced smart contracts, such as destroying smart contracts and calling functions in other smart contracts. Knowing these global out-of-the-box available variables and functions will make it easier for the solidity developer who's authoring them.

We will cover the following topics in this chapter:

- Variable scoping
- Type conversion
- Block-related global variables

- Transaction-related global variables
- Cryptographic global variables
- Address-related global properties and functions
- Contract-related global variables and functions
- Recovering addresses using `ecrecover`

So, let's dive right in!

Technical requirements

The following tools are required for working along with this chapter:

- A Chrome browser
- An online Remix editor

All code from this chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter04>.

Variable scoping

Scoping refers to the availability of a variable within a function and a contract in Solidity. Solidity provides the following two locations where variables can be declared:

- Contract-level global variables (also known as state variables)
- Function-level local variables

It is quite easy to understand function-level local variables. They are only available anywhere within a function and not outside.

Contract-level global variables are variables that are available to all functions, including constructors, fallbacks, and modifiers within a contract. Contract-level global variables can have a visibility modifier attached to them. It is important to understand that state data can be viewed across an entire network, irrespective of the visibility modifier. The following state variables can only be modified using functions:

- `public`: These state variables are accessible directly from external calls. A getter function is implicitly generated by the compiler to read the value of public state variables.

- **internal**: These state variables are not accessible directly from external calls. They are accessible from functions within a current contract and child contracts deriving from it.
- **private**: These state variables are not accessible directly from external calls. They are also not accessible from functions from child contracts. They are only accessible from functions within a current contract.

Let's take a look at the preceding state variables in the following code block:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ScopingStateVariables {
    // uint64 public myVar = 0;

    // uint64 private myVar = 0;

    // uint64 internal myVar = 0;
}
```

Now that we understand the scoping rules for variables, you must know that each variable has a data type, and at times, these need to be converted from one type to another. Let's dive into type conversion in Solidity to convert values from one type to another.

Type conversion

By now, we know that Solidity is a statically typed language, where variables are defined with specific data types at compile time. The data type cannot be changed for the lifetime of the variable. This means that it can only store values that are legal for a data type – for example, uint8 can store values from 0 to 255. It cannot store negative values or values greater than 255. Take a look at the following code to better understand this:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ErrorDataType {
```

```
function HoistingDemo() public pure returns (uint) {  
  
    uint8 someVar = 100;  
    someVar = 300; //error  
  
}  
  
}
```

However, there are times when these conversions are required to copy a value from a variable of one type to another, and these are called **type conversions**. The examples for type conversions are shown in the next section.

In Solidity, we can perform various kinds of conversions. Let's learn a bit more about these in the next subsections.

Implicit conversion

Implicit conversion means that there is no need for an operator, or no external help is required for conversion. These types of conversions are perfectly legal, and there is no loss of data or mismatch of values. They are completely type-safe. Solidity allows for implicit conversion from smaller to larger integral types – for example, converting uint8 to uint16 happens implicitly.

Explicit conversion

Explicit conversion is required when a compiler does not perform implicit conversion, either because of a loss of data or a value containing data not falling within a target data type range. Solidity provides a function for each value type for explicit conversion. An example of explicit conversion is uint16 to uint8. Data loss is possible in such a case.

The following code listing shows examples for both implicit and explicit conversions:

- `ConversionExplicitUINT8toUINT256`: This function executes an explicit conversion from uint8 to uint256. Note that this conversion is also possible implicitly:

```
function ConversionExplicitUINT8toUINT256() pure public  
    returns (uint){  
    uint8 myVariable = 10;  
    uint256 someVariable = myVariable;  
    return someVariable;  
}
```

- `ConversionExplicitUINT256toUINT8`: This function executes an explicit conversion from uint256 to uint8. This conversion will raise a compile-time error if it happens implicitly:

```
function ConversionExplicitUINT256toUINT8() pure public
    returns (uint8) {
    uint256 myVariable = 10;
    uint8 someVariable = uint8(myVariable);
    return someVariable;
}
```

- `Conversions`: This function shows an example of implicit and explicit conversions. Some conversions are not allowed and fail, while the legal ones succeed. In the following code listing, comments are placed to show the result of each conversion:

```
function Conversions() pure public {
    uint256 myVariable = 10000134;
    uint8 someVariable = 100;
    bytes4 byte4 = 0x65666768;

    // bytes1 byte1 = 0x656667668; //error
    bytes1 byte1 = 0x65;
    // byte1 = byte4; //error, explicit conversion
    // needed here
    byte1 = bytes1(byte4); //explicit conversion
    byte4 = byte1; //Implicit conversion
    // uint8 someVariable = myVariable; // error,
    // explicit conversion needed here

    myVariable = someVariable; //Implicit conversion
    string memory name = "Ritesh";
    bytes memory nameInBytes = bytes(name);
    //explicit string to bytes conversion
    name = string(nameInBytes); //explicit bytes to
    //string conversion
}
```

All the preceding three functions can be placed within a single contract for the sake of completeness. The next contract has three function definitions, already shown previously:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ConversionDemo {

    function ConversionExplicitUINT8toUINT256() pure public
        returns (uint) {
        ....
    }

    function ConversionExplicitUINT256toUINT8() pure public
        returns (uint8) {
        ....
    }

    function Conversions() pure public {
        ....
    }
}
```

This section focused on converting values from one data type to another and showed both implicit and explicit ways of performing conversions. Solidity provides access to internal state by means of global variables and certain advanced functionalities by way of global functions. In the next section, we will discuss the transaction and block-level global variables.

Block and transaction global variables

Solidity provides access to a few global variables that are not declared within contracts but are accessible from code within contracts. Contracts cannot access a ledger directly. A ledger is maintained by miners only; however, Solidity provides some information about the current transaction and blocks to the contracts so that they can utilize them. Solidity provides both block- and transaction-related variables.

The following code illustrates examples of using global transaction, block, and message variables:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract TransactionAndMessageVariables {
    event logstring(string);
    event loguint(uint);
    event logbytes(bytes);
    event logaddress(address);
    event logbyte4(bytes4);
    event logblock(bytes32);

    function GlobalVariable() public payable {
        emit logaddress( block.coinbase ); // miners address
        emit loguint( block.difficulty ); // 70762765929000
        emit loguint( block.gaslimit ); // 160000000
        emit loguint( gasleft() ); // 79975149
        emit loguint( tx.gasprice ); // 1
        emit loguint( block.number ); // 123
        emit loguint( block.timestamp ); // 1513061946
        emit logbytes( msg.data ); // 0x4048d797
        emit logbyte4( msg.sig ); // // 0x4048d797
        emit loguint( msg.value ); // 0 or in Wei if ether are
        send
        emit logaddress( msg.sender ); // sender address
        emit logaddress( tx.origin ); // caller address
        emit logblock ( blockhash( block.number) ); // current
        number
    }
}
```

This section showed a simple example of reading values from global variables and emitting them as event values for easy viewing. These values can be used in a smart contract to dynamically base them on the current smart contract runtime context.

Transaction- and message-related global variables

The following is a list of global variables along with their data types and a description, provided as a ready reference:

<u>Variable name</u>	<u>Description</u>
block.coinbase (address payable)	Same as etherbase. Refers to the miner's address.
block.difficulty (uint)	Difficulty level of current block.
block.gaslimit (uint)	Gas limit for current block.
block.number (uint)	Block number in sequence.
block.timestamp (uint)	Time when block was created.
msg.data (bytes)	Information about the function and its parameters that created the transaction.
msg.sender (address)	Address of caller who invoked the function.
msg.sig (bytes4)	Function identifier using first four bytes after hashing function signature.
msg.value (uint)	Amount of wei sent along with transaction.
Gasleft() (uint256)	Gas left provides the gas available after every instruction execution in a function of a smart contract.
tx.gasprice (uint)	The gas price caller is ready to pay for each gas unit.
tx.origin (address)	The first caller of the transaction.
block.blockhash(uint blockNumber) returns (bytes32)	Hash of the block containing the transaction.
Block.chainid (uint)	Returns the current chain identifier.
Block.basefee (uint)	Base fees for the current block.

Figure 4.1 – A list of global variables and their data types

As mentioned before, the global variables are quite important, and this list contains all the important ones available in Solidity. However, it might not be clear to you how tx.origin and msg.sender are different, since they provide output as an address. This is covered next.

The difference between tx.origin and msg.sender

If you're sharp-eyed, you might have noticed, in the previous code block, that both tx.origin and msg.sender show the same result and output. The tx.origin global variable refers to the original external account that started the transaction, while msg.sender refers to the immediate account (which could be external or another contract account) that invokes the function. The tx.origin variable will always refer to the external account, while msg.sender can be a contract or an external account. If there are multiple function invocations on multiple contracts, tx.origin will always refer to the account that started the transaction, irrespective of the stack of contracts invoked. However, msg.sender will refer to the immediate previous account (contract/external) that invokes the next contract. Therefore, it is recommended to use msg.sender over tx.origin.

Cryptographic global variables

Solidity provides cryptographic functions for hashing values within contract functions. There are two hashing functions – SHA2 and SHA3.

The sha3 or keccak256 function converts the input into a hash based on the sha3 algorithm, while sha256 converts the input into a hash based on the sha2 algorithm. It is recommended to use the keccak256 function for hashing needs.

The following code block illustrates this:

```
pragma solidity >=0.7.0 <0.9.0;

contract CryptoFunctions {

    function CryptoDemo() pure public returns (bytes32,
        bytes32) {
        return (sha256("r"), keccak256("r"));
    }

}
```

The result of executing this function is shown in the following screenshot. The result of both the keccak256 and sha3 functions is the same:

```
"0": "bytes32: 0x454349e422f05297191ead13e21d3db520e5abef52055e4964b82fb213f593a1",
"1": "bytes32: 0x414f72a4d550cad29f17d9d99a4af64b3776ec5538cd440cef0f03fef2e9e010"
```

Figure 4.2 – The result of the SHA function

Both the functions work on tightly packed arguments, meaning that multiple parameters can be concatenated together to find a hash, as shown in the following code snippet:

```
keccak256(abi.encodePacked(uint(97), uint(98), uint(99)));
```

Cryptographic functions are used quite a lot within smart contracts to generate a hash from a function signature typically used for function invocation and to obfuscate and store data. These use symmetric-based encryption and provide uniqueness for storing keys within a mapping.

Address global variables

Every address, externally owned or contract-based, has five global functions and three global properties. These functions and properties will be explored in depth in subsequent chapters on Solidify functions. The global properties related to the address are balance, code, and codehash. The balance property is a `getter` property and helps in retrieving the current ether balance (in wei denomination) held by an address. Both code and codehash provide the bytecode responsible for contract creation in a different format.

The functions are as follows:

- `<address>.transfer(uint256 amount)`: This function sends the given amount of wei to address and throws an exception in the event of failure.
- `<address>.send(uint256 amount) returns (bool)`: This function sends the given amount of wei to address and returns `false` in the event of failure.

`<address>.call(...)` `returns (bool)`: This function issues a low-level `call` and returns a tuple containing the status and result from function call.

- `<address>.staticcall(...)` `returns (bool)`: This function issues a low-level `callcode` and returns a tuple containing the status and result from function call.
- `<address>.delegatecall(...)` `returns (bool)`: This function issues a low-level `delegatecall` and returns a tuple containing the status and result from function call.

These are quite important functions and are used extensively within smart contracts. These functions will be discussed in depth in *Chapter 6, Writing Smart Contracts*. The transfer and send functions help in transferring ether from one contract to another or externally owned accounts, while the other three functions are used to call functions in other contracts and libraries.

Contract global variables

Every contract has the following three global functions:

- `this`: This represents the current contract and can be used from within a contract. It is a short form to refer to the current contract.

It can call any method within a contract as long as they are tagged as public or have external scope visibility, as shown next:

```
This.functionname(parameters)
```

It can be converted into an address type. This is generally required to call low-level functions such as `transfer`, `send`, and `call` on a target contract to either call a method or transfer some tokens. It also helps in distinguishing existing contracts from other contract variables. The usage of this is shown next:

```
Address(this).transfer(1 ether)
```

- `selfdestruct`: This function accepts an address recipient argument. It destroys the current contract, sending its available funds to the given address, as shown next:

```
Selfdestruct(0x5B38Da6a701c568545dCfcB03FcB875f56beddc4)
```

- `suicide`: This is an alias of the `selfdestruct` function. It also accepts an address to which the current balance is transferred. However, it has been deprecated since Solidity 0.5.0.

The first two functions are quite frequently used within smart contracts. `selfdestruct` is often used within smart contracts related to tokens or for those whose smart contracts have a fixed lifespan, after which the contract should be invalidated.

There is an additional global function provided by Solidity that helps to recover Ethereum addresses from a message hash and the associated digital signature.

Recovering addresses using ecrecover

Solidity provides a very powerful function known as `ecrecover`, which is used to derive the address of a sender based on the digital signature.

The concept of digital certificates was covered in *Chapter 1, Introduction to Blockchain, Ethereum, and Smart Contracts*. In summary, digital signatures help in identifying the sender of a message. The message is hashed and encrypted using the private key of the sender, known only to the sender. Anyone decrypting the message using the corresponding public key can be assured that it is originating from the holder of the private key.

There are situations when it is necessary to verify the sender of a message using the signature itself. It is especially useful in scenarios involving self-sovereign identities, decentralized exchanges, and off-chain computations.

Ethereum cryptography is based on the **Elliptic Curve Digital Signature Algorithm (ECDSA)**, and the `ecrecover` function helps in recovering the address from the signature. The signature comprises three elements, where `r` is the `x` coordinate of the ECDSA curve and `s` is derived from `r`. The recovery identifier is `v`, and its value ranges from 27 to 30.

The `ecrecover` function is available in Solidity assembly only. Without assembly, this function cannot be used. It accepts four parameters – `Ecrecover` (`messageHash`, `v`, `r`, and `s`).

Here, `messageHash` is the hashed message sent to a contract; `s` is the first 32 bytes of data in the digital signature; `r` is the data in the digital signature, starting from position 66 and ending at 130, with a length of 32 bytes; and `v` is the last 1 byte in the digital signature.

Before getting into contract implementation using `ecrecover`, we should know the process of generating a message hash and digital signature using an Ethereum account. This is shown next.

The following steps assume that Ganache-cli is executing in the background, and the first dummy address available in Ganache will be used for signing the message. From another console, we can attach to Ganache and send commands to it.

The Geth `attach` command attaches the console to the existing running instance of Ganache using the `127.0.0.1:8545` HTTP endpoint, and the output confirms that the console is successfully connected to Ganache (previously known as TestRPC):

```
> geth attach 127.0.0.1:8545

instance: EthereumJS TestRPC/v2.13.2/ethereum-js
coinbase: 0x1fd60057985434174d44b9098992d397b2cee491
at block: 0 (Fri Apr 15 2022 08:07:10 GMT+0530 (IST))
modules: eth:1.0 evm:1.0 net:1.0 personal:1.0 rpc:1.0 web3:1.0
```

Ganache provides 10 addresses out of the box every time it is executed on the console. These steps use the first account from Ganache, which can be accessed using the `eth.accounts` command. Your output will be different to what is shown here:

```
> eth.accounts[0]
"0x1fd60057985434174d44b9098992d397b2cee491"
```

Next, a message, "hello ritesh", is hashed using the `sha3` algorithm and stored in the `msg` variable:

```
> var msg = web3.sha3("hello ritesh")
```

The hashed output is needed later, so it is printed on the console. The output should be noted and stored for later use:

```
> console.log(msg)  
0x23ad06b0e032848201fe7dccf69320f381a6de007e  
7e9a0896f5cf04821cc95f
```

Finally, a signature is generated for messageHash using the Ganache-provided account and is stored in the sig variable:

```
> var sig = eth.sign(eth.accounts[0], msg)
```

The signature is printed on the console, as it will be needed for later use. The signature is 132 bytes in length:

```
> console.log(sig)  
0x692beda0e15876f154e6b385842941d404c7447b9729a213ee290027  
e9d2f75737ab  
a255952ff08dbe9ada64a224076e245c24ea01428f68053d5b42ac9434f01
```

The next three lines of code extract the three arguments expected by the ecrecover function. The r part is 66 bytes long. The s part is also 66 bytes long; 0x is appended with 64 bytes from the signature data, starting from the 66th position. The last part, v, is 2 bytes in length. It is a static value, ranging from 27 to 30. All the three parts of the signature, r, s, and v, are printed on the console so that we can copy and use them to send a transaction from the remix editor:

```
> var r = sig.substr(0, 66)  
  
> var s = "0x" + sig.substr(66, 64)  
  
> var v = 28  
  
> console.log(r)  
0x692beda0e15876f154e6b385842941d404c7447b9729a213ee2900  
27e9d2f757  
  
> console.log(s)
```

```
0x37aba255952ff08dbe9ada64a224076e245c24ea01428f68053d
5b42ac9434f

> console.log(v)
28
```

Now that we have the message hash and the signature data broken into its three constituents, `r`, `s`, and `v`, the focus can be moved toward writing a contract that can cull or extract the address from them.

A new contract named `AddressValidation` in `ecrecover.sol` is created in the remix editor with the following code. The contract has a single function, `ExtractAddress`, that accepts four arguments – these are the same arguments that are expected by the `ecrecover` function. The function returns the Ethereum address extracted from the message hash and signature data:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract AddressValidations {
    function ExtractAddress(bytes32 hash, uint8 v, bytes32 r,
                           bytes32 s) public returns(address) {
        bytes memory prefix = "\x19Ethereum Signed
Message:\n32";
        bytes32 prefixedHash = keccak256(abi.encodePacked(
            prefix, hash));
        return ecrecover(prefixedHash, v, r, s);
    }
}
```

Every Ethereum-based message should start with `\x19Ethereum Signed Message:\n32` so that it is recognizable and ensures that the signature generated is from an Ethereum-based account only. Within the function code, a hash is generated from the data obtained after combining the message hash with the special Ethereum-provided literal value, `\x19Ethereum Signed Message:\n32`. The newly generated hash along with the three arguments are passed to the `ecrecover` function to extract the address.

The input and output from executing the `ExtractAddress` function in the remix editor are shown next:

```
{
    "bytes32 hash": "0x23ad06b0e032848201fe7dccf69320f381a6de007e7e9a0896f5cf04821cc95f",
    "uint8 v": 28,
    "bytes32 r": "0x692beda0e15876f154e6b385842941d404c7447b9729a213ee290027e9d2f757",
    "bytes32 s": "0x37aba255952ff08dbe9ada64a224076e245c24ea01428f68053d5b42ac9434f"
} ⌂

{
    "0": "address: 0x1fd60057985434174D44B9098992D397b2cEE491"
} ⌂
}
```

Figure 4.3 – The input and output related to the ecrecover function; the recovered address is the output. The output shows the same address we used for generating the signature from the hashed data using Ganache.

It is also possible to send the complete signature to the `contract` function, which extracts the arguments needed by the `ecrecover` function using assembly code, as shown here:

```
function ExtractAddressAnotherVersion(bytes32 hash, bytes
memory sig)
public returns(address) {
    if (sig.length != 65) {
        return address(0);
    }
    bytes32 rValue;
    bytes32 sValue;
    uint8 vValue;

    assembly {
        rValue := mload(add(sig, 32))
        sValue := mload(add(sig, 64))
        vValue := byte(0, mload(add(sig, 96)))
    }

    if (vValue < 27) {
        vValue += 27;
    }

    bytes memory prefix = "\x19Ethereum Signed Message:\\n\\n";
    bytes32 r;
    bytes32 s;
    uint8 v;
}
```

```
    bytes32 prefixedHash = keccak256(abi.encodePacked(  
        prefix, hash));  
    address recoveredAddress = ecrecover(prefixedHash,  
        vValue, rValue, sValue);  
  
    return (recoveredAddress);  
  
}
```

Essentially, this code is the same as the previously shown code for recovering an address using `ecrecover`, with the only difference being that the three `r`, `s`, and `v` arguments are extracted using the `mload` assembly function from the signature. If the signature length is less than 65 bytes, it is not a valid signature.

Summary

This chapter, in many ways, was a continuation of previous chapters. Variables were discussed in depth in the first half of this chapter. Variable scoping and data type conversions were elaborated on, along with code examples. The latter half of the chapter focused on globally available variables and functions. Transaction- and message-related variables, such as `block.coinbase` and `msg.data`, were explained. The difference between `msg.sender` and `tx.origin`, along with their usage, was also explained in this chapter. This chapter also discussed cryptographic-, address-, and contract-level functions. However, we will focus on these functions in another chapter later in this book.

The following chapter will focus on Solidity expressions and control structures, covering programming details about loops and conditions. This will be an important chapter because every program needs some kind of looping to perform repetitive tasks, and Solidity control structures help implement these. Loops are based on conditions, and conditions are written using expressions. These expressions are evaluated and return either a `true` or `false` value. Stay tuned while we plunge into control structures and expressions in the following chapter.

Questions

1. If a user invokes a function on a contract that, in turn, invokes another function in another contract, which global variable will provide the value for the user?
2. What are the different scopes for variables in Solidity?

Further reading

To know more about global variable and data type conversion related to Solidity, go to <https://docs.soliditylang.org/en/v0.5.3/units-and-global-variables>.

5

Expressions and Control Structures

Taking decisions in code is an important aspect of a programming language, and Solidity should also be able to execute different instructions based on circumstances. Solidity provides the `if...else` statement for this purpose. It is also important to loop through multiple items, and Solidity provides multiple constructs such as `for` loops and `while` statements for this purpose. In this chapter, we will discuss in detail the programming constructs that will help you make decisions and loop through a set of values. The goal of this chapter is to understand the programming constructs and expressions available in Solidity, helping you to write functional smart contracts with loops and conditional statements.

This chapter covers the following topics:

- Understanding Solidity expressions
- The `if...else` statement
- Exploring the `while` loop
- Understanding the `for` loop
- Understanding the `do while` loop
- Understanding the `break` and `continue` keywords
- Understanding the `return` statement

Technical requirements

There are not many technical requirements for this chapter. The following will suffice:

- An operating system – Windows, macOS, and Linux work equally well
- A Chrome browser for navigation to `remix.org` and writing smart contracts

All code from this chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter05>.

Understanding Solidity expressions

An expression refers to a statement (comprising multiple operands and, optionally, zero or more operators) that results in a single value, object, or function. The operand can be a literal, variable, function invocation, or another expression itself.

An example of an expression is as follows:

```
Age > 10
```

In the preceding example, `Age` is a variable, and `10` is an integer literal. `Age` and `10` are operands, and the `>` (greater than) symbol is the operator. This expression returns a single Boolean value (`true` or `false`), depending on the value stored in `Age`.

Expressions can be more complex comprising multiple operands and operators, as follows:

```
((Age > 10) && (Age < 20)) || ((Age > 40) && (Age < 50))
```

In the preceding code, there are multiple operators in play. The `&&` operator acts as an AND operator between two expressions, which in turn comprises operands and operators. There is also an OR operator, represented by the `||` operator between two complex expressions.

Solidity has the following comparison operators that help in writing expressions that return Boolean values:

Operator	Meaning	Sample example
<code>==</code>	Equals	<code>myVar == 10</code>
<code>!=</code>	Not equals	<code>myVar != 10</code>
<code>></code>	Greater than	<code>myVar > 10</code>
<code><</code>	Less than	<code>myVar < 10</code>
<code>>=</code>	Greater than or equal to	<code>myVar >= 10</code>
<code><=</code>	Less than or equal to	<code>myVar <= 10</code>

Solidity also provides the following logical operators that help in writing expressions that return Boolean values:

Operator	Meaning	Sample example
&&	AND	(myVar > 10) && (myVar < 10)
	OR	myVar != 10
!	NOT	myVar > 10

The following operators have precedence in Solidity, just like other languages:

Precedence	Description	Operator
1	Postfix increment and decrement	++ , --
New expression	new <typename>	NA
Array subscripting	<array>[<index>]	NA
Member access	<object>. <member>	NA
Function-like call	<func>(<args...>)	NA
Parentheses	(<statement>)	NA
2	Prefix increment and decrement	++ , --
Unary plus and minus	+ , -	NA
Unary operations	delete	NA
Logical NOT	!	NA
Bitwise NOT	~	NA
3	Exponentiation	**
4	Multiplication, division, and modulo	*, /, %
5	Addition and subtraction	+, -
6	Bitwise shift operators	<<, >>
7	Bitwise AND	&
8	Bitwise XOR	^
9	Bitwise OR	
10	Inequality operators	<, >, <=, >=
11	Equality operators	==, !=
12	Logical AND	&&
13	Logical OR	
14	Ternary operator	<conditional> ? <if-true> : <if-false>
15	Assignment operators	=, =, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
16	Comma operator	,

This concludes the details about understanding Solidity expressions and authoring them within smart contracts. Now, it's time to move on to writing conditional statements in smart contracts.

Understanding the if and if...else decision controls

Solidity provides conditional code execution with the help of the `if...else` instructions. The general structure of `if...else` is as follows:

```
if (this condition/expression is true) {  
    Execute the instructions here  
}  
  
else if (this condition/expression is true) {  
    Execute the instructions here  
}  
  
else {  
    Execute the instructions here  
}
```

`if` and `if...else` are keywords in Solidity, and they inform the compiler that they contain a decision control condition – for example, `if (a > 10)`. Here, `if` contains a condition that can evaluate as either `true` or `false`. If `a > 10` evaluates to `true`, then the code instructions that follow in the pair of double-brackets, `{` and `}`, should be executed.

`else` is also a keyword that provides an alternate path if none of the previous conditions are `true`. It also contains a decision control instruction and executes the code instructions if `a > 10` tends to be `true`.

The following example shows the usage of the `if-else` conditions. An `enum` with multiple constants is declared. A `StateManager` function accepts an `uint8` argument, which is converted into an `enum` constant and compared within the `if...else` decision control structure. If the value of `_state` is 1, then the returned result is 1; if the argument contains 2 or 3 as a value, then the `else...if` portion of code gets executed; and if the value is other than 1, 2, or 3, then the `else` part is executed:

```
// SPDX-License-Identifier: MIT  
pragma solidity >=0.7.0 <0.9.0;
```

```

contract IfElseExample {

    enum requestState {created, approved, provisioned,
                      rejected, deleted, none}

    function StateManagement(uint8 _state) public returns
        (int result) {

        requestState currentState = requestState(_state);

        if(currentState == requestState(1)) {
            result = 1;
        } else if ((currentState == requestState.approved)
                   || (currentState == requestState.provisioned)) {
            result = 2;
        } else {
            currentState == requestState.none;
            result = 3;
        }
    }
}

```

Now that we've learned about using conditional statements within smart contracts, it's now time to move on to learn about while loops in smart contracts.

Exploring while loops

There are times when we need to execute a code segment repeatedly based on a condition. Solidity provides while loops precisely for this purpose. The general form of the while loop is as follows:

```

Declare and initialize a counter
while (check the value of counter using an expression or
      condition) {
    Execute the instructions here
    Increment the value of counter
}

```

`while` is a keyword in Solidity, and it informs the compiler that it contains a decision control instruction. If this expression evaluates to `true`, then the code instructions that follow in the pair of double-brackets (`{` and `}`) should be executed. The `while` loop keeps executing until the condition turns `false`.

In the following example, `mapping` is declared along with `counter`. This helps loop `mapping`, since there is no out-of-the-box support in Solidity to do so.

An event is used to get details about transaction information. We will discuss events in detail in the *Events and logging* section in *Chapter 8, Exceptions, Events, and Logging*. For now, it is enough to understand that you are logging information whenever an event is invoked. The `setNumber` function adds data to `mapping`, and the `getNumbers` function runs a `while` loop to retrieve all entries within the `mapping` and log them using events.

Important Note

A temporary variable is used as a counter that is incremented by one at every execution of the `while` loop.

The `while` condition checks the value of the temporary variable and compares it with the global counter variable. Based on whether it's true or false, the code within the `while` loop is executed. Within this set of instructions, the value of a counter should be modified so that it can help to exit the loop by making the `while` condition `false`, as shown in the following code listing:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract whileLoop {
    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function setNumber() public {
        blockNumber[counter++] = block.number;
    }
}
```

```

function getNumbers() public {
    uint i = 0;
    while (i < counter) {
        emit uintNumber( blockNumber[i] );
        i = i + 1;
    }
}

```

With this, we can conclude this section on using `while` loops within smart contracts. Let's move on to understand another type of loop in smart contracts.

Understanding the do...while loop

The `do...while` loop is very similar to the `while` loop. The general form of a `do...while` loop is as follows:

```

Declare and Initialize a counter
do {
    Execute the instructions here
    Increment the value of counter
} while(check the value of counter using an expression or
       condition)

```

There is a subtle difference between the `while` and `do...while` loops. Note that the condition in `do...while` is placed toward the end of the loop instructions. The instructions in the `while` loop are not executed at all if the condition is `false`; however, the instruction in the `do...while` loop gets executed at least once before the condition is evaluated. So, if you want to execute the instructions at least once, the `do...while` loop should be preferred to the `while` loop. Take a look at the following code snippet on the same subject:

```

// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract DowhileLoop {

    mapping (uint => uint) blockNumber;
}

```

```
uint counter;

event uintNumber(uint);

function setNumber() public {
    blockNumber[counter++] = block.number;
}

function getNumbers() public {
    uint i = 0;
    do {
        emit uintNumber( blockNumber[i] );
        i = i + 1;
    } while (i < counter);
}
}
```

This concludes details about using `do...while` loops within smart contracts. Now, it's time to move on to learn about taking control over path execution using breaks in smart contracts.

Understanding the `for` loop

One of the most famous and most used loops is the `for` loop, and we can use it in Solidity. The general structure of a `for` loop is as follows:

```
for (initialize loop counter; check and test the counter;
     increase the value of counter;) {
    Execute multiple instructions here
}
```

`for` is a keyword in Solidity, and it informs the compiler that it contains information about looping a set of instructions. It is very similar to the `while` loop; however, it is more succinct and readable, since all information can be viewed in a single line.

The following code example shows the same solution – looping through a mapping. However, it uses the `for` loop instead of the `while` loop. The `i` variable is initialized, incremented by 1 in every iterator, and checked to see whether it is less than the value of `counter`. The loop will stop as soon as the condition becomes `false` – that is, the value of `i` is equal to or greater than `counter`:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ForLoopExample {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function SsetNumber() public {
        blockNumber[counter++] = block.number;
    }

    function getNumbers() public{

        for (uint i=0; i < counter; i++){
            emit uintNumber( blockNumber[i] );
        }
    }
}
```

That's it! Now that we have learned how to use `for` loops within smart contracts, it's time to move to one more variant of a loop within smart contracts.

Understanding breaks

Loops help to iterate over from the start till they arrives on a vector data type. However, there are times when you might want to stop the iteration in between and jump out or exit from the loop without executing the conditional test again. The `break` statement helps us do that. It helps us terminate the loop by passing the control to the first instruction after the loop.

In the following code example, the `for` loop is terminated and control moves out of the `for` loop when the value of `i` is 1 because of the use of the `break` statement. It literally breaks the loop, as shown in the following code block:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ForLoopExampleBreak {
    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function setNumber() public {
        blockNumber[counter++] = block.number;
    }

    function getNumbers() public {
        for (uint i=0; i < counter; i++){
            if (i == 1)
                break;
            emit uintNumber(blockNumber[i]);
        }
    }
}
```

This concludes the details about using `break` statements within smart contracts. Now, it's time to move on to learning about taking control over path execution using `continue` in smart contracts.

Understanding continue

Loops are based on expressions. The logic of the expression decides the continuity of the loop. However, there are times when you are in between loop execution and want to go back to the first line of code without executing the rest of the code for the next iteration. The `continue` statement helps us do that.

In the following code block, the `for` loop is executed till the end; however, the values after 5 are not logged at all:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ForLoopExampleContinue {
    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function setNumber() public {
        blockNumber[counter++] = block.number;
    }

    function getNumbers() public {
        for (uint i=0; i < counter; i++) {
            if ((i > 5))
                { continue; }
            emit uintNumber(blockNumber[i]);
        }
    }
}
```

With this, we conclude the details about using `continue` statements within smart contracts. Let's now move on to learning about returning values from functions in smart contracts.

Understanding return

Returning data is an integral part of a Solidity function. Solidity provides two different syntaxes for returning data from a function. In the following code sample, two functions – `getBlockNumber` and `getBlockNumber1` – are defined. The `getBlockNumber` function returns `uint` without naming the `return` variable. In such cases, developers can resort to using the `return` keyword explicitly to return from the function.

The `getBlockNumber1` function returns `uint` and also provides a name for the variable. In such cases, developers can directly use and return this variable from a function without using the `return` keyword, as shown in the following code listing:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract ReturnValues {
    uint counter;

    function setNumber() public {
        counter = block.number;
    }

    function getBlockNumber() public view returns (uint) {
        return counter;
    }

    function getBlockNumber1() public view returns (uint
        result) {
        result = counter;
    }
}
```

This concludes the details about using return values from functions within smart contracts, and this also concludes all the topics to be covered as part of Solidity expressions and control structures.

Summary

Expressions and control structures are an integral part of any programming language, and they are an important element of the Solidity language as well. Solidity provides a rich infrastructure for decision and looping constructs. It provides `if...else` decision control structures and the `for`, `do...while`, and `while` loops for looping over data variables that can be iterated. Solidity also allows us to write conditions and logical, assignments, and other types of statement that any programming language supports. Knowledge of expressions and control structures helps in writing conditions and functional code that can evaluate expressions at runtime, loop over multiple values, and return values to callers from functions. They are an important part of any programming language, and knowing them helps to write better smart contracts in Solidity.

The following chapter will discuss Solidity and contract functions in detail; these are core elements for writing contracts. Blockchain is about executing and storing transactions, which are created when contract functions are executed. Functions can change the state of Ethereum or just return the current state. Functions that change state and those that return the current state will be discussed in detail in the following chapter.

Questions

1. Which loop is ideal in a scenario where the looping condition is based on a number?
2. Which loop is ideal for a scenario in which the loop should exit based on a condition?
3. What keywords are available to exit from a loop before it gets completed?

Further reading

The Solidity docs are a good introduction on expressions and control structures, available at <https://docs.soliditylang.org/en/v0.8.9/control-structures.html>.

Part 2: Writing Robust Smart Contracts

Armed with an understanding of the fundamentals of Solidity, in this section, we will explore more complex Solidity topics, such as exception handling using `try-catch` blocks, object orientation in Solidity, in-depth discussions on the `address` type and its low-level functions, and special functions such as `fallback` and `receive`. We will also show ways to debug smart contracts and adopt engineering practices such as unit testing.

This part contains the following chapters:

- *Chapter 6, Writing Smart Contracts*
- *Chapter 7, Functions, Modifiers, and Fallbacks*
- *Chapter 8, Exceptions, Events, and Logging*
- *Chapter 9, Truffle Basics and Unit Testing*
- *Chapter 10, Debugging Contracts*

6

Writing Smart Contracts

Solidity is an object-oriented programming language used to write smart contracts. This chapter will discuss the design aspects of writing smart contracts, defining and implementing a contract, and deploying and creating contracts using different mechanisms—using the new keyword and contract addresses. Object orientation is based on four concepts—abstraction, encapsulation, inheritance, and polymorphism—and this chapter will delve deep into object-oriented concepts and implementations.

This chapter covers the following topics:

- Writing a simple contract
- Creating contracts
- Creating contracts via the new and existing address
- Contract constructor
- Contract composition
- Inheritance
- Encapsulation
- Polymorphism

- Method overloading
- Abstract contracts
- Interfaces
- Advance interfaces
- Library

Technical requirements

To follow the instructions in this chapter, you will need the following:

- The Chrome or Firefox browser
- The Remix editor

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter06>.

Smart contracts

Smart contracts are programs (bytecode) deployed and executed in the **Ethereum Virtual Machine (EVM)**. A contract is a term generally used in the legal world and has little relevance in the programming world. But writing a smart contract in Solidity does not mean writing a legal contract. Instead, contracts are like any other programming code, containing Solidity code, and are executed when someone invokes them.

There is nothing smart about smart contracts. It is a blockchain term; a piece of jargon used to refer to programming logic and code that executes within a blockchain virtual machine. A blockchain virtual machine is an interpreter that understands programming constructs related to smart contracts.

A smart contract is very similar to a C++, Java, or C# class. Just as a class is composed of state (variables) and behaviors (methods), contracts contain state variables and functions. The purpose of state variables is to store data within a contract. Functions are responsible for executing logic, performing the update, and reading operations in the current state.

We have already seen some examples of smart contracts in *Chapter 5, Expressions and Control Structure*; however, it's time to dive deeper into the subject.

Writing a smart contract

A contract is declared using the `contract` keyword, along with an identifier, as shown in the following code snippet:

```
contract SampleContract {  
}
```

Within the brackets comes the declaration of state variables and function definitions. *Chapter 3, Introducing Solidity*, introduced a smart contract that contained almost all important constructs to build a smart contract. The same contract is shown again for quick reference. This contract has state variables, struct definitions, enum declarations, function definitions, modifiers, and events. State variables, structs, and enums were discussed in detail in *Chapter 4, Global Variables and Functions*. Functions, modifiers, and events will be discussed in detail over the next two chapters. Take a look at the following code listing depicting the contract:

```
// SPDX-License-Identifier: GPL-3.0  
pragma solidity >=0.7.0 <0.9.0;  
  
contract GeneralStructure {  
    int public stateIntVariable;  
    string stateStringVariable;  
    address personIdentifier;  
    MyStruct human;  
    bool constant hasIncome = true;  
  
    struct MyStruct {  
        string name;  
        uint myAge;  
        bool isMarried;  
        uint[] bankAccountsNumbers;  
    }  
  
    modifier onlyBy() {  
        if (msg.sender == personIdentifier) {  
            _;
```

```
}

}

event ageRead(address, int);
enum gender {male, female}

function getAge (address _personIdentifier) onlyBy()
    payable external returns (uint) {
    human = MyStruct("Ritesh", 10, true, new uint [] (3));
    gender _gender = gender.male; //using enum
    emit ageRead(personIdentifier, stateIntVariable);
}
}
```

The previous contract shows some of the major elements of smart contracts. This contract does not do anything meaningful and it is used just for showing the important elements of a smart contract. This smart contract call can be invoked externally or can be used by other smart contracts. Smart contracts can use other smart contracts by creating a reference to them using either the new keyword or their existing deployed address. The next section shows these different ways of creating a reference to a smart contract.

Creating contracts

There are the following two ways of creating and using a contract in Solidity:

- Using the new keyword
- Using the address of the already-deployed contract

Using the new keyword

The new keyword in Solidity deploys and creates a new contract instance. It initializes the contract instance by deploying the contract, initializing the state variables, running its constructor, setting the nonce value to 1, and, eventually, returning the address of the instance to the caller.

Deploying a contract involves checking whether the sender has provided enough gas to complete the deployment, generating a new account/address for contract deployment using the requestor's address and nonce values, and passing on any Ether sent along with it.

In the following code snippet, two contracts, `HelloWorld` and `client`, are defined. In this scenario, one contract (`client`) deploys and creates a new instance of another contract (`HelloWorld`). It does so using the `new` keyword:

```
HelloWorld myObj = new HelloWorld();
```

Let's take a look at the following code:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract HelloWorld {
    uint private simpleInt;
    function getValue() public view returns (uint) {
        return simpleInt;
    }

    function setValue(uint _value) public {
        simpleInt = _value;
    }
}

contract Client {
    function UseNewKeyword() public returns (uint) {
        HelloWorld myObj = new HelloWorld();
        myObj.setValue(10);
        return myObj.getValue();
    }
}
```

The previous contract shows the use of the `new` keyword to create a new reference to a contract. The `new` keyword would deploy the contract and return the instance using which functions can be invoked.

The new keyword generates a new address for the contract. The value of an address is based on the current address of the parent contract and a number known as nonce (this is a counter maintained internally by a contract referring to the number of contracts created by it). The value of a contract address to be generated can be predicted based on these inputs. It is also possible to influence and change the value of the new address for the contract using a salt value. Salt adds randomness while generating contract addresses and it becomes more difficult to predict the next generated address by the parent contract.

The same client contract shown before can be rewritten to use a salt value with the new keyword while creating a new contract instance:

```
contract Client {
function UseNewKeyword(bytes32 _salt) public returns (uint) {
HelloWorld myObj = (new HelloWorld){salt: _salt}();
myObj.setValue(10);
return myObj.getValue();
}
}
```

It is also possible to send ethers while creating a new instance of the contract using the `new` keyword, as shown here:

```
contract Client {
function UseNewKeyword(bytes32 _salt) public returns (uint) {
HelloWorld myObj = (new HelloWorld){value:
    10000000000000000000, salt: _salt}();
myObj.setValue(10);
return myObj.getValue();
}
}
```

It is important to note that ether can only be sent to the target contract if the current contract holds Ethers as its own balance and the target contract has a payable constructor.

Now, let's focus on using another method for creating a reference to an already-deployed contract using its address.

Using the address of a contract

This method of creating a contract instance is used when a contract is already deployed and instantiated. It uses the address of an existing, deployed contract. No new instance is created; rather, an existing instance is reused. A reference to the existing contract is made using its address.

In the following code example, two contracts, `HelloWorld` and `Client`, are defined. In this scenario, one contract (`Client`) uses an already-known address of another contract to create a reference to it (`HelloWorld`). The `Client` contract defines a function called `setObject` that accepts an address. This address should be the address of the `HelloWorld` contract. The address is stored in the `obj` state variable such that it can be used across all functions within the contract. The address can now be cast to the `HelloWorld` contract whenever its function needs to be invoked. This is shown in the `UseExistingAddress` function in the client contract:

```
HelloWorld myObj = HelloWorld(obj);
```

Let's take a look at the following code:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract HelloWorld {
    uint private simpleInt;
    function GetValue() public view returns (uint) {
        return simpleInt;
    }

    function SetValue(uint _value) public {
        simpleInt = _value;
    }
}

contract Client {
    address obj ;
    function setObject(address _obj) external {
        obj = _obj;
    }
}
```

```
}

function UseExistingAddress() public returns (uint) {
    HelloWorld myObj = HelloWorld(obj);
    myObj.SetValue(10);
    return myObj.GetValue();
}
```

The client contract expects an address and so the `HelloWorld` contract should already be deployed on an Ethereum network. In effect, a dependence on the `HelloWorld` contract is made implicitly, and the address should be known before any of the `Client` contract functions are called. This is also one of the ways to reuse an already-deployed contract, along with its state, in another contract. Now, let's focus on using another important concept, known as a constructor, within smart contracts.

Contract constructor

Solidity supports declaring a constructor within a contract. They are optional and the compiler induces a default constructor when none is explicitly defined.

The constructor is executed once while deploying the contract. It is quite different from other programming languages. In other languages, a constructor is executed whenever a new object instance is created. Deployment of the contract also happens using the `new` keyword and each time it deploys a new contract instance with a new address. In short, constructor code is invoked every time a new contract instance is created using the `new` keyword, or it is deployed using frameworks such as Truffle/Hardhat.

Constructors should be used to initialize state variables and set up the context. The constructor code is the initial code executed for a contract. There can be at most one constructor in a contract, unlike constructors in other programming languages. Constructors can take parameters and corresponding arguments must be supplied while deploying the contract.

A constructor function does not have a name. Earlier versions of Solidity defined a constructor using the name of the contract. A constructor cannot have a visibility scope applied to it. They are executed automatically while creating and deploying a contract. A contract constructor can have a payable attribute associated with it. This will enable it to accept Ether during deployment and contract instance creation time.

In the following example, a constructor is defined without a name and any scope visibility attribute. It sets the storage variable value to 5:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract HelloWorld {
    uint private simpleInt;
    constructor() {
        simpleInt = 5;
    }

    function GetValue() public view returns (uint) {
        return simpleInt;
    }

    function SetValue(uint _value) public {
        simpleInt = _value;
    }
}
```

This was a simple example of defining a constructor within a smart contract. There are multiple ways for a smart contract to reuse another smart contract. Reusing an existing contract as either a deployed instance or code helps in building modular contracts and breaks down a larger solution into multiple smaller contracts. The next section discusses composing solutions using multiple smaller contracts using the concept of contract composition.

Contract composition

Solidity supports contract composition. Composition refers to combining multiple contracts to create complex data structures and contracts. We have already seen numerous examples of contract composition before. Refer to the code snippet for creating contracts using the new keyword shown in the *Using the new keyword* section. In this example, the `Client` contract is composed of the `HelloWorld` contract. Here, `HelloWorld` is an independent contract and `Client` is a dependent contract. It is dependent on the `HelloWorld` contract for its completeness. It is a good practice to break down problems into multicontract solutions and compose them together using contract composition.

Inheritance

Inheritance is one of the pillars of object orientation and Solidity supports inheritance between smart contracts. Inheritance is the process of defining multiple contracts that are related to each other through parent-child relationships. In inheritance, there is a parent contract and child contracts deriving from the parent contract. Inheritance is defined using an `is-a` relationship between parent and child contracts. The child contract inherits all the functions of the parent contract. There is another terminology to refer to parent and child contracts—base (parent) contract and derived (child) contracts.

Inheritance is about code reusability. As mentioned before, there should be an `is-a` relationship between parent and child contracts, and all public-, external- and internal-scoped functions and state variables are available to derived contracts. In fact, internally, the Solidity compiler copies the base contract bytecode into the derived contract bytecode. The `is` keyword is used to inherit the base contract in the derived contract. It is one of the most important concepts that should be mastered by every Solidity developer because of the way contracts are versioned and deployed.

Solidity supports multiple types of inheritance, including single, multilevel, and multiple inheritance. A single contract is generated from all contracts in inheritance by the Solidity compiler copying all contracts in the chain into a final contract. This final contract is deployed to the Ethereum network and accessible using the generated address.

We will learn more about the types of inheritance in the following sections.

Single inheritance

Single inheritance helps in inheriting the variables, functions, modifiers, and events of base contracts into the derived class. Take a look at the following diagram. **Contract B** is the contract deriving from the base contract, **Contract A**. There is an `is-a` relationship between them:

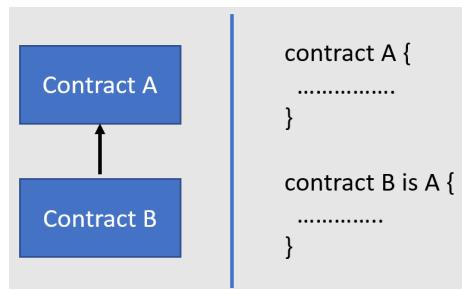


Figure 6.1 – Single inheritance between contracts

The following code snippets help to explain single inheritance. You will observe that there are two contracts, ParentContract and ChildContract. The ChildContract contract inherits from ParentContract. ChildContract will inherit all public and internal variables and functions. Anybody using ChildContract, as seen in the Client contract, can invoke both GetInteger and SetInteger functions as if they were defined in ChildContract. The single inheritance implementation is shown in the following code block:

```

// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ParentContract {
    uint internal simpleInteger;
    function SetInteger(uint _value) external {
        simpleInteger = _value;
    }
}

contract ChildContract is ParentContract {
    bool private simpleBool;
    function GetInteger() public view returns (uint) {
        return simpleInteger;
    }
}

contract Client {
    ChildContract pc = new ChildContract();
    function workWithInheritance() public returns (uint) {

```

```
pc.SetIntege(100);  
return pc.GetInteger();  
}  
}
```

When inheritance is involved, each contract can define its own constructors. All the constructors from base to derived contracts are executed in sequential order. The constructor from the base contract is executed before the derived contract constructor.

Multilevel inheritance

Multilevel inheritance is very similar to single inheritance; however, instead of just a single parent-child relationship, there are multiple levels of a parent-child relationship.

This is shown in the following diagram. **Contract A** is the parent of **Contract B** and **Contract B** is the parent of **Contract C**:

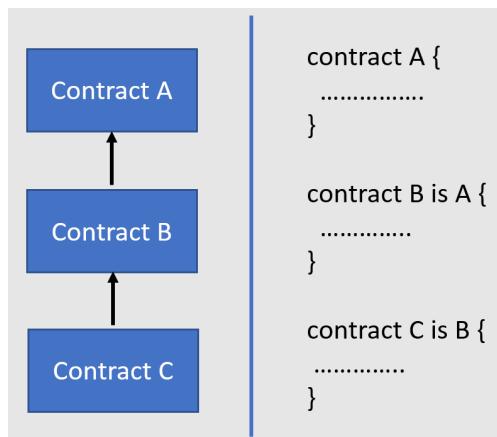


Figure 6.2 – Multilevel inheritance between contracts

Hierarchical inheritance

Hierarchical inheritance is again similar to simple inheritance. Here, however, a single contract acts as a base contract for multiple derived contracts. This is shown in the following diagram. Here, **Contract A** is derived from both **Contract B** and **Contract C**:

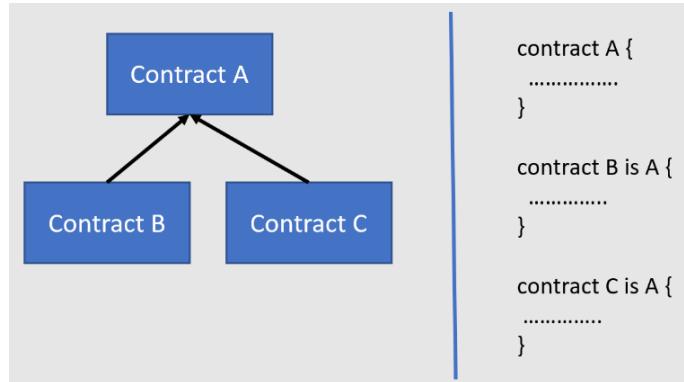


Figure 6.3 – Hierarchical inheritance between contracts

Multiple inheritance

There can be multiple levels of single inheritance. However, there can also be multiple contracts that derive from the same base contract. These derived contracts can be used as base contracts together in further child classes. When contracts inherit from such child contracts together, there is multiple inheritance, as shown in the following diagram:

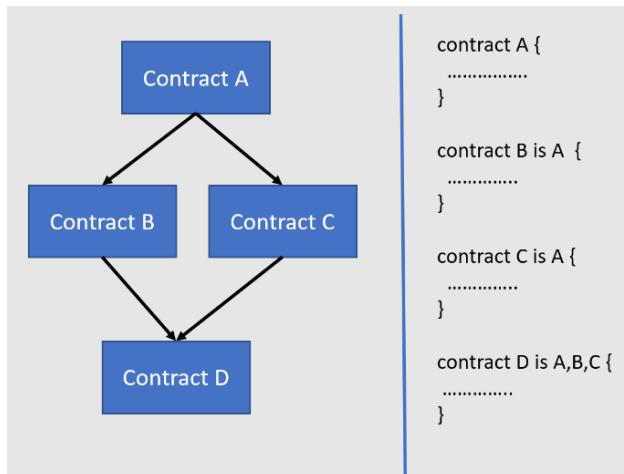


Figure 6.4 – Multiple inheritance between contracts

The following code listing shows an example of multiple inheritance. In this example, `SumContract` acts as a base contract that is derived from the `MultiContract` and `DivideContract` contracts. The `SumContract` contract provides an implementation of the `Sum` function and the `MultiContract` and `DivideContract` contracts provide an implementation of the `Multiply` and `Divide` functions, respectively.

Both MultiContract and DivideContract are inherited in SubContract. The SubContract contract provides an implementation of the Sub function. The Client contract is not part of the parent-child hierarchy and consumes other contracts. The Client contract creates an instance of SubContract and calls the Sum method on it.

Solidity follows the path of Python and uses **C3 linearization**, also known as **Method Resolution Order (MRO)**, to force a specific order in graphs of base contracts. The contracts should follow a specific order while inheriting, starting from the base contract through to the most derived contract. An example of such sequencing is shown next, in which the SubContract contract is derived from SumContract, DivideContract, and MultiContract.

The following code listing shows that MultiContract is an immediate parent contract for the SubContract contract, followed by DivideContract and SumContract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract SumContract {
    function Sum(uint a, uint b) public returns (uint) {
        return a + b;
    }
}

contract MultiContract is SumContract {
    function Multiply(uint a, uint b) public virtual returns
        (uint) {
        return a * b;
    }
}

contract DivideContract is SumContract {
    function Multiply(uint a, uint b) public virtual returns
        (uint) {
        return a / b;
    }
}

contract SubContract is SumContract, MultiContract,
```

```

DivideContract{
    function Sub(uint a, uint b) public returns (uint) {
        return a - b;
    }

    function Multiply(uint a, uint b) public override
        (MultiContract, DivideContract) returns (uint) {
        return a * b;
    }
}

contract client {
    function WorkWithInheritance() public returns (uint) {
        uint a = 20;
        uint b = 10;
        SubContract subt = new SubContract();
        return subt.Sum(a,b);
    }
}

```

It is also possible to invoke a function specific to a contract by using the contract name along with the function name.

Now that we understand composition and inheritance for smart contracts, let's dive into another main pillar of object-oriented programming, known as **encapsulation**, in the next section.

Encapsulation

Encapsulation is one of the most important pillars of OOP. It refers to the concept of declaring state variables that cannot be accessed directly by clients and can only be accessed and modified using functions. This helps in restricting access to variables but, at the same time, allows enough access to the class for taking action on it.

Solidity provides multiple visibility modifiers, such as `external`, `public`, `internal`, and `private`, which affect the visibility of state variables within the contract in which they are defined, inheriting child contracts or outside contracts.

After covering inheritance and encapsulation, the next important object-oriented concept is polymorphism, which is the next topic of discussion.

Polymorphism

Poly means *many* and *morph* means *forms*. Together, the word *polymorphism* means *multiple forms*. It means that contracts in inheritance can be accessed using a common interface. It also means that multiple functions with the same name can be defined and invoked using different objects. There are the following two types of polymorphism:

- Function polymorphism
- Contract polymorphism

Function polymorphism

Function polymorphism refers to declaring multiple functions within the same contract or inheriting contracts with the same name. The functions differ in the parameter data types or the number of parameters. Return types are not taken into consideration for determining valid function signatures for polymorphism. This is also known as **method overloading**.

The following code segment illustrates a contract that contains two functions that have the same name but different data types for incoming parameters. The first function, `getVariableData`, accepts `int8` as its parameter data type, while the next function with the same name accepts `int16` as its parameter data type. It is absolutely fine to have the same function name with a different number of parameters of different data types for incoming parameters, as shown in the following code:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract FunctionPolymorphism
{
    function GetVariableData(int8 data) public pure returns
        (int8) {
        return data;
    }
    function GetVariableData(int16 data) public pure returns
        (int16) {
```

```
    return data;
}
}
```

Defining multiple functions with the same features is the way to achieve function-level polymorphism within a smart contract, and they should differ in the type and number of parameters they accept as inputs and generate outputs. Solidity also supports contract-level polymorphism, which is the topic of the next section.

Contract polymorphism

Contract polymorphism refers to using multiple contract instances interchangeably when the contracts are related to each other via inheritance. In addition, contract polymorphism helps in invoking derived contract functions using a base contract instance.

Let's understand this concept with the help of the following code listing.

A parent contract contains two functions, `SetInteger` and `GetInteger`. A child contract inherits from a parent contract and provides its own implementation of `GetInteger`. The child contract can be created using the `ChildContract` variable data type and it can also be created using the parent contract data type. Polymorphism allows the use of any contract in a parent-child relationship with the base type contract variable. The contract instance decides which function will be invoked—the base or derived contract.

Take a look at the following code snippet:

```
ParentContract pc = new ChildContract();
```

The preceding code creates a child contract and stores the reference in the parent contract type variable. This is how contract polymorphism is implemented in Solidity, as shown in the following code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract ParentContract {
    uint internal simpleInteger;

    function SetInteger(uint _value) public {
        simpleInteger = _value;
```

```
}

function GetInteger() virtual public view returns (uint) {
    return 10;
}

contract ChildContract is ParentContract {
    function GetInteger() override public view returns (uint) {
        return simpleInteger;
    }
}

contract Client {
    ParentContract pc = new ChildContract();
    function WorkWithInheritance() public returns (uint) {
        pc.SetIntegers(100);
        return pc.GetInteger();
    }
}
```

This concludes the discussion on polymorphism. Now, let's move on to another important object-orientation topic, known as **method overriding** and **abstract contracts**.

Method overriding

Method overriding is the process of redefining a function available in the parent contract with the same name and signature in the derived contract. A parent contract contains two functions, `SetInteger` and `GetInteger`. A child contract inherits from the parent contract and implements `GetInteger` by overriding the function.

Now, when a call to the `GetInteger` function is made on the child contract, even while using a parent contract variable, the child contract instance function is invoked. This is because all functions in contracts are virtual and based on a contract instance; the most derived function is invoked.

Abstract contracts

Abstract contracts are contracts that have partial function definitions. You cannot create an instance of an abstract contract. An abstract contract must be inherited by a child contract to utilize its functions.

Abstract contracts help in defining the structure of a contract, and any class inheriting from it must ensure to provide an implementation for them. If the child contract does not provide the implementation for incomplete functions, its instance cannot be created. The function signature is terminated using a semicolon (;). Solidity provides an *abstract* keyword to mark a contract as abstract. This is a relatively new keyword addition and it was not part of prior versions. A contract becomes an abstract contract when it consists of functions without any implementation.

The following code snippet is an implementation of an abstract contract.

The `abstractHelloWorld` contract is an abstract contract as it contains a couple of functions without any definitions. `GetValue` and `SetValue` are function signatures without any implementation. There is another method that returns a constant. The purpose of `AddNumber` is to show that there can be functions within an abstract contract containing an implementation as well.

The `abstractHelloWorld` abstract contract is inherited by the `HelloWorld` contract, which provides an implementation for all the methods. The `client` contract creates an instance of the `HelloWorld` contract using the base contract variable and invokes its functions, as shown in the following code snippet:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract AbstractHelloWorld {
    function GetValue() virtual public view returns (uint);
    function SetValue(uint _value) virtual public;
    function AddNumber(uint _value) virtual public returns
        (uint) {
        return _value;
    }
}

contract HelloWorld is AbstractHelloWorld{
    uint private simpleInteger;
```

```
function GetValue() override public view returns (uint) {
    return simpleInteger;
}
function SetValue(uint _value) override public {
    simpleInteger = _value;
}
function AddNumber(uint _value) override public view
    returns (uint){
    return (simpleInteger + _value);
}
}

contract Client {
AbstractHelloWorld myObj;
constructor() {
myObj = new HelloWorld();
}
function GetIntegerValue() public returns (uint) {
myObj.SetValue(100);
return myObj.AddNumber(200) + 10;
}
}
```

Abstract contracts are an important concept that helps in achieving polymorphism and writing modular code. Abstractions are the first step toward using interfaces. In fact, pure abstract contracts are also known as **interfaces** and that is the topic of the next section.

Interfaces

Interfaces are like abstract contracts, but there are differences. Interfaces cannot contain any definitions; they can only contain function declarations. It means functions in interfaces cannot contain any code. They are also known as **pure abstract contracts**.

An interface can contain only the signature of functions. It also cannot contain any state variables. It cannot inherit from other contracts or contain enums or structures. However, interfaces can inherit other interfaces. The function signatures terminate using the semicolon (;) character. Interfaces are declared using the `interface` keyword followed by an identifier. The following code block shows an implementation of an interface. The `IHelloWorld` interface is defined, containing two function signatures—`GetValue` and `SetValue`. There are no functions containing any implementation. `IHelloWorld` is implemented by the `HelloWorld` contract. Contracts that intent to use this contract would create an instance as it would do normally, as shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

interface IHelloWorld {
    function GetValue() external view returns (uint);
    function SetValue(uint _value) external;
}

contract HelloWorld is IHelloWorld{
    uint private simpleInteger;
    function GetValue() public view returns (uint) {
        return simpleInteger;
    }
    function SetValue(uint _value) public {
        simpleInteger = _value;
    }
}

contract Client {
    function GetSetIntegerValue() public returns (uint) {
        IHelloWorld myObj = new HelloWorld();
        myObj.SetValue(100);
        return myObj.GetValue() + 10;
    }
}
```

Interfaces are a cornerstone feature of object-oriented programming and Solidity supports them completely. In this section on interfaces, we saw that we can create a new contract instance and assign it to a variable of the interface type. Using the interface type, all functions supported by the interface can be invoked. Other methods supported by the target contract are not reachable by the interface type. This section showed an example of using an interface at the code level. The code is available for the target contract and so, it was possible to use the `new` keyword to create an instance of a contract and assign it to the variable type. However, how can we use variables of the interface type for already-deployed contracts when only their address is available? This is explained in the next section.

Advanced interfaces

There are situations in which the contract code is not available to us. The contract is already deployed on a network and its address is available to us. The only way to use such a contract is through low-level address-provided functions (`send`, `transfer`, `staticcall`, `call`, and `delegatecall`). This is because we do not have the definition of that contract and we cannot use the `new` keyword for it. If such a contract implements an interface and that interface definition is available to us, we will be able to use it to reference the target contract through its address itself. Let's see this in action.

First, we will define an interface called `IMaths` that contains a single function called `GetSquare`. The function accepts a single argument of the `uint256` type and returns another `uint256` as its return value. This interface is defined in the `interfaces.sol` file:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

interface IMaths {
    function GetSquare(uint256 value) external returns
        (uint256);
}
```

Next, the `IMaths` interface is inherited by a `Mathematics` contract and this contract implements the `GetSquare` function. The `GetSquare` function squares the argument value and returns it to the caller, as shown in the following code snippet. The `Mathematics` contract is defined in the `mathematics.sol` file and imports the `interfaces.sol` file, along with the `IMaths` interface definition:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "./interfaces.sol";

contract Mathematics is IMaths {
    function GetSquare(uint256 value) external pure returns
        (uint256) {
        return value ** 2;
    }
}
```

The `Mathematics` contract can now be deployed on an Ethereum network and this will generate its address. The address is important as it will be used by the clients of this contract.

The `client` contract comprises a single function called `CallSquare`, which accepts two arguments—the address of the target contract (in this case, it would be the address of the `mathematics` contract) and an `inputValue` of the `uint256` type (this value should be squared). The function returns the result obtained from the `GetSquare` function implemented in the `Mathematics` contract. The `client` contract imports the `IMaths` interface by importing the `interfaces.sol` file. The function uses the provided address of the `Mathematics` contract and, using the interface, creates a reference to the target contract. The `mathematics` contract, due to its implementation of the `IMath` interface, becomes accessible by using the interface definition, as shown in the following code:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "./interfaces.sol";

contract Client {
    function CallSquare(address targetContract, uint256
```

```
    inputValue) public returns (uint256) {
        IMaths targetMathematics = IMaths(targetContract);
        return targetMathematics.GetSquare(inputValue);
    }
}
```

It is important to note that only functions defined within the interface are callable using this method. Other functions defined in the `Mathematics` contract are not accessible using this method.

One of the important features of Solidity is creating reusable functions in the form of a library and making it available to multiple contracts within a network. The library concept helps us achieve reusability and is covered in the next section.

Library

Programming languages provide facilities to write reusable code and use it across multiple projects. Solidity has a similar concept through which code written once in a library can be reused across multiple smart contracts. A library in Solidity is created using the `library` keyword followed by the `library` code within a `{ }` block:

```
library {
}
```

The concept of a library might sound very similar to that of a contract; however, there are differences. The similarity of a library with a contract is that they both consist of functions and they both can be deployed on the Ethereum network. They both generate unique addresses on the Ethereum network. However, a library can declare its own state variables. A library does not manage or maintain any state. It has a set of functions that are available for use as reusable code. So, ideally, they are best suited for implementing logic that is common across contracts without the involvement of state variables.

The code for a simple library implementation is shown next:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

library MyMathLibrary {
    function sum( uint256 a, uint256 b) public returns
        (uint256) {
        return a + b ;
    }
}
```

```
}

function exponential( uint256 a,uint256 b) public returns
    (uint256) {
return a ** b ;
}
}
```

A library can be used in two ways within a contract. It can be used by importing the library into a contract. This is applicable when the library code is available. However, if the library code is not available, then it should already be deployed on the Ethereum network with a valid address. Using the address of a library, it is possible to use the functions of the library.

Importing a library

We need a client contract to use a library. A smart contract with functions calling the library function is implemented in the following code snippet. A library can be imported in the current contract using the `import` keyword. The `import` statements are generally placed before any contract definition. An `import` statement consists of an identifier that can point to the location of the library code and a semicolon for statement termination, as shown here:

```
Import "./mylib.sol";
```

It is important to note that you only need to import a library when the code for the library is in a different file from the contract definition file. There is no need to import a library if the code for both the library and contract is in the same file.

After importing the library, the client contract can make calls to the library function using the `<<contractname>>. <<functionname>>(param1, param2, ..., paramn)` syntax. The contract name should be prefixed before the function name:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "./mylib.sol";

contract LibraryClient {
    function GetExponential(uint256 firstVal, uint256
        secondVal) public returns(uint256) {
```

```
    return MyMathlibrary.exponential(firstVal, secondVal);  
}  
}
```

Execution of the `GetExponential` function in `LibraryClient` will provide the result of `firstVal` squared by `secondVal`.

Another way to use a library is by using a deployed instance and its associated address. This uses a low-level `delegatecall` function, which is shown in the next chapter.

Summary

This brings us to the end of this chapter. It was a heavy chapter that focused primarily on smart contracts, the different ways to create an instance, and all the important object-oriented concepts related to them, including inheritance, polymorphism, abstraction, and encapsulation. Multiple types of inheritance can be implemented in Solidity. Simple, multiple, hierarchical, and multilevel inheritance were discussed, along with the usage and implementation of abstract contracts and interfaces. It should be noted that by using inheritance in Solidity, there is eventually just one contract that is deployed instead of multiple contracts. There is just one address that can be used by any contract with a parent-child hierarchy. You learned how to use object-oriented concepts, such as inheritance, abstraction, and polymorphism, with Solidity and contracts.

The next chapter will focus purely on functions within contracts. Functions are central to writing effective Solidity contracts. They help change the contract state and retrieve them. Without functions, having any meaningful smart contracts is difficult.

Functions have different visibility scopes. Multiple attributes are available that affect their behavior and also help in accepting Ether. Stay tuned for a function ride in the next chapter!

Questions

1. What are the different ways through which a contract instance can be created within a smart contract?
2. Do smart contracts support both function- and contract-level polymorphism?
3. What are the differences between abstract contracts and interfaces?

Further reading

Ethereum Cookbook, by Manoj P R, Packt Publishing: https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781789133998/7/ch07lvl1sec91/creating-upgradable-smart-contracts

7

Solidity Functions, Modifiers, and Fallbacks

Solidity is maturing and providing advanced programming constructs so that users can write better smart contracts. This chapter is dedicated to some of the most important smart contract constructs, such as **functions**, **modifiers**, and **fallbacks**.

Functions are the most important element of a smart contract after state variables. It is functions that help to create transactions and implement custom logic in **Ethereum**. There are various types of functions, which will be discussed in depth in this chapter.

Modifiers are special functions that help in writing more readily available and modular smart contracts. Fallbacks are a concept unique to contract-based programming languages, and they are executed when a function call does not match any existing declared method in a contract. Finally, every function has visibility attached to it that affects its availability to the external caller, other contracts, and contracts in inheritance.

This chapter covers the following topics:

- Function input and output
- Modifiers

- Visibility and scope
- Views, constants, and pure functions
- Address related functions
- Fallback functions
- Receive functions

This is an important chapter that will explore topics that are frequently used with smart contracts. Functions, modifiers, and fallback and receive functions are interrelated, and having a good grasp on them will help you become a successful Solidity developer.

Technical requirements

To follow the instructions in this chapter, you will need the following:

- A Chrome or Firefox browser
- The Remix editor

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter07>.

Function input and output

Functions would not be that interesting if they didn't accept parameters and return values. Functions are made generic with the use of parameters and return values. Parameters can help by changing function execution and providing different execution paths. Solidity allows you to accept multiple parameters within a function; the only condition is that their identifiers should be uniquely named.

We will use the following code listing to understand functions and their different flavors of input parameters and return values:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Parameters {
    function SingleIncomingParameter(int _data) public
        pure returns (int _output) {
            return _data * 2;
```

```

    }
    function MultipleIncomingParameter(int _data, int
        _data2) public pure returns (int _output) {
        return _data * _data2;
    }
    function MultipleOutgoingParameter(int _data) public
        pure returns (int square, int half) {
        square = _data * _data;
        half = _data / 2 ;
    }
    function MultipleOutgoingTuple(int _data) public pure
        returns (int square, int half) {
        (square, half) = (_data * _data, _data / 2 );
    }
}

```

Let's examine the preceding code, as follows:

1. The first function, `singleIncomingParameter`, accepts one parameter named `_data` of the `int` type and returns a single return value that is identified using `_output` of the `int` type. The function signature provides constructs to define both the incoming parameters and return values. The `returns` keyword in the function signature helps define the return types from the function. In the following code snippet, the `return` keyword within the function code automatically maps to the first return type declared in the function signature:

```

function singleIncomingParameter(int _data)
    returns (int _output) {
    return _data * 2;
}

```

2. The second function, `multipleIncomingParameter`, accepts two parameters, `_data` and `_data2`, both of the `int` type, and returns a single return value, identified by using `_output` of the `int` type, as follows:

```

function multipleIncomingParameter(int _data, int
    _data2)
    returns (int _output) {
    return _data * _data2;
}

```

3. The third function, `multipleOutgoingParameter`, accepts one parameter, `_data`, of the `int` type and returns two return values, identified using `square` and `half`, which are both of the `int` type. In the following code snippet, returning multiple parameters is something unique to Solidity and is not found in many programming languages:

```
function multipleOutgoingParameter(int _data)
    returns (int
        square, int half)
{
    square = _data * _data;
    half = _data /2 ;
}
```

4. The fourth function, `multipleOutgoingTuple`, is similar to the third function. However, instead of assigning return values as separate statements and variables, it returns values as a tuple. A **tuple** is a custom data structure consisting of multiple variables in a group, as shown in the following code snippet:

```
function multipleOutgoingTuple(int _data) returns
    (int square, int half)
{
    (square, half) = (_data * _data,_data /2 );
```

It is also possible to declare parameters without any identifier at all. However, this feature does not have much utility, as those parameters cannot be referenced within the function code. Similarly, return values can be declared without any identifier.

Now that we understand functions, it's time to get into the details of modifiers, which have some unique concepts in Solidity compared to other languages.

Modifiers

Modifiers are another concept unique to Solidity. They help in modifying the behavior of a function. Let's try to understand this with the help of an example. The following code does not use modifiers; in this contract, two state variables, two functions, and a constructor are defined.

One of the state variables stores the address of the account deploying the contract. Within the constructor, the `msg.sender` global variable is used to input the account address in the owner state variable. The following functions check whether the caller is the same as the account that deployed the contract. If it is, the function code is executed; otherwise, it ignores the rest of the code.

While this code will work, it can be made better both in terms of readability and manageability. This is where modifiers can help. In this example, the checks are made using the `if` conditional statements. Later, in the next chapter, we will see how to use new Solidity constructs, such as `require` and `assert`, to execute the same checks without the `if` condition. Take a look at the next code listing, which does not use modifiers:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ContractWithoutModifier {
    address owner;
    int public mydata;
    constructor() {
        owner = msg.sender;
    }
    function AssignDoubleValue(int _data) public {
        if(msg.sender == owner) {
            mydata = _data * 2;
        }
    }
    function AssignTenerValue(int _data) public {
        if(msg.sender == owner) {
            mydata = _data * 10;
        }
    }
}
```

Modifiers are special functions that change the behavior of a function. Here, the function code remains the same, but the execution path of a function changes. Modifiers can only be applied to functions. Let's now see how to write the previous contract using modifiers:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ContractWithModifier {
    address owner;
    int public mydata;
    constructor() {
        owner = msg.sender;
    }
    modifier isOwner {
        // require(msg.sender == owner);
        if(msg.sender == owner) {
            _;
        }
    }
    function AssignDoubleValue(int _data) public isOwner {
        mydata = _data * 2;
    }
    function AssignTenerValue(int _data) public isOwner {
        mydata = _data * 10;
    }
}
```

The contract shown here has the same constructs – a constructor, two state variables, and two functions. It also has an additional special function that is defined using the `modifier` keyword. The function code for both the `AssignDoubleValue` and `AssignTenerValue` functions is different, although they have a similar functionality. These functions do not use the `if` condition to check whether the caller of the function is the same as the account that deployed the contract; instead, these functions are decorated with the modifier name in their signature.

Let's now try to understand the modifier construct in Solidity and its usage.

Modifiers are defined using the `modifier` keyword and an identifier. The code for the modifier is placed within curly brackets. The code within a modifier can validate the incoming value and conditionally execute the called function after evaluation. The `_` identifier is of special importance here – its purpose is to replace itself with the function code that is invoked by the caller.

When a caller calls the `AssignDoubleValue` function, which is decorated with the `isOwner` modifier, the modifier takes control of the execution and replaces the `_` identifier with the called function code – that is, `AssignDoubleValue`. Eventually, in EVM, the modifier looks like the following code during runtime:

```
modifier isOwner {
    // require(msg.sender == owner);
    if(msg.sender == owner) {
        mydata = _data * 2;
    }
}
```

The same modifier can be applied to multiple functions, and the `_` identifier can be replaced with the called function code.

This helps in writing cleaner, more readable, and more maintainable code. Developers do not have to keep repeating the same code in every function or check for the incoming value when executing a function.

Each function and variable can have scope assigned to it. Scope determines the visibility of functions and variables to others. In the next section, the concept of scope will be covered.

Visibility scope

As we know, contracts comprise functions and state variables. So, when we declare functions and state variables, the next question that arises is who can access them. Visibility scope helps in determining who can view and access them. Solidity provides four levels of visibility modifiers. They vary in their usage and determine the level of visibility to their callers:

- **Private:** This is the most limited and constrained visibility modifier available in Solidity. *Private* means private to a contract. So, if a function is defined and marked as private within a contract, this function is only visible within the contract and it is not callable or visible from outside the contract, including child contracts.

- **Internal:** This is built on top of private scoping rules, and internal functions are visible within a contract and not from outside. However, it adds another rule that states that functions within any contract inheriting it can also call and have visibility of the internal function.
- **Public:** Public scope is more relaxed in terms of visibility compared to internal and private visibility scoping modifiers. Public visibility for a function means that it is visible within the contract, to any child contracts inheriting the contract and other external contracts, and to individually owned accounts. This means such functions are accessible to everyone within the ecosystem.
- **External:** External scoping refers to visibility that ensures that these functions can only be called externally from other contracts and externally owned accounts. These functions cannot be invoked from within the current contract or derived contracts. The `receive` and `fallback` functions that are covered later in this chapter are mandated to have external scope visibility, as these special functions cannot be invoked explicitly by anyone apart from EVM. Also, the state variables cannot be declared using external visibility

Apart from having scope applied to functions, it is also possible to apply additional Solidity-provided modifiers to function those constraints or limit their capability. There are three such modifiers – `view`, `constant`, and `pure` – and the next section discusses these in detail.

View, constant, and pure functions

Solidity provides special modifiers for functions, such as `view`, `pure`, and `constant`. These are also known as **state mutability** attributes because they define the scope of changes allowed within the Ethereum global state. The purpose of these modifiers is similar to those discussed previously, but there are some small differences. This section will detail the use of these keywords.

Writing smart contract functions helps primarily with the following three activities:

- Updating state variables
- Reading state variables
- Logic execution

The execution of functions and transactions costs gas and is not free of cost. Every transaction needs a specified amount of gas, based on its execution, and callers are responsible for supplying that gas for successful execution. This is true for transactions or for any activity that modifies the global state of Ethereum.

There are functions that are only responsible for reading and returning the values in a state variable, and these are like property getters in other programming languages. They read the current value in a state variable and return values to the caller. These functions do not change the state of Ethereum. However, it is important to know the actions that tend to modify the state. The actions that change the state of smart contracts and Ethereum include the following:

- Writing to state variables
- Emitting events
- Creating other contracts
- Using `selfdestruct`
- Sending ether via `send` and `transfer`
- Calling any function not marked `view` or `pure`
- Using low-level calls
- Using inline assembly that contains certain opcodes

Solidity developers can mark their functions with the `view` modifier to suggest to EVM that this function does not change the Ethereum state, or any activity mentioned before. An example of the `view` function is shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ViewFunction {
    function GetTenerValue(int _data) public view returns
        (int) {
        return _data * 10;
    }
}
```

If you have functions that just return values without any modification of state, they can be marked with the `view` function.

It is also worth noting that the `view` functions are also known as `constant` functions. The `constant` functions were used in previous versions of Solidity.

The `pure` functions are more restrictive in terms of state mutability when compared to the `view` functions; however, their purpose is the same – that is, to restrict state mutability. It is also worth noting that even the `pure` functions are not enforced at the time of writing, but we expect them to be in the future.

The `pure` functions add further restrictions on top of the `view` functions – for example, a `pure` function is not allowed to even read the current state of Ethereum. In short, the `pure` functions disallow reading and writing to Ethereum's global state. The additional activities not allowed according to the documentation include the following:

- Reading from state variables
- Accessing `this.balance` or `<address>.balance`
- Accessing any of the members of `block`, `tx`, and `msg` (with the exception of `msg.sig` and `msg.data`)
- Calling any function not marked `pure`
- Using inline assembly that contains certain opcodes

The previous function has been rewritten as a `pure` function in the following code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

contract PureFunction {
    function GetTenerValue(int _data) public pure returns
        (int) {
        return _data * 10;
    }
}
```

Note how this `pure` function, although performing some calculations, is not accessing any state variable. Next, we will get into the details of methods provided by the `address` data type.

Address-related functions

Chapter 3, Introducing Solidity focused on introducing major data types in Solidity, including the ubiquitous address type. This is probably one of the most used data types in smart contracts. Address types provide multiple methods, which were not discussed in *Chapter 3*, because of their close association with contracts and functions. It is time to introduce all methods and properties supported by the address type.

The address data type provides five functions and a single property. The only property provided by address is the balance property, which provides the balance available in an account (contract or individual) in wei denomination, as shown in the following code snippet:

```
<<account>>.balance ;
```

It is important to note the preceding code should be modified while getting the balance from the contract address, as shown next. `this` is a special keyword in Solidity and refers to the current contract. It can only be used within a contract to refer to the current contract:

```
address(<<contract address>>).balance or  
Address(this).balance.
```

In the preceding couple of code listings, `account` is a valid Ethereum address, and this returns the balance available in terms of wei.

Now, let's take a look at the methods provided by the address data type.

The address send method

The `send` method is used to send ether to a contract or an externally owned account. Take a look at the following code depicting the `send` method:

```
<<account>>.send(amount) ;
```

The `send` function provides 2,300 units of gas as a fixed limit stipend to the target address, and this cannot be superseded. This is especially important when sending Ether to a contract address. To send an amount to an externally owned account, this amount of gas is enough. The `send` function returns a `true` or `false` Boolean type as a return value. Instead of an exception, the `send` function returns `false`, marking the execution as a failed transaction. If `send` is used along with the contract address, it will invoke a `fallback` or `receive` function on the contract. We will investigate `fallback` and `receive` functions in detail in the next section.

Now, let's see an example of the `send` function in action, as shown in the following code snippet:

```
function SimpleTransferToAccount() public {
    msg.sender.send(1);
}
```

In the preceding code, the `send` function sends 1 wei to the caller of the `SimpleSendToAccount` function. We learned about `msg.sender` in *Chapter 5, Expressions and Control Structure*, when dealing with global variables.

The `send` function is a low-level function and should be used with caution, as it can invoke `fallback` or `receive` functions that may recursively call back within the calling contract again and again. There is a pattern known as **Check-Deduct-Transfer (CDF)**, or sometimes **Check-Effects-Interaction (CEI)**, which we look at in the next code listing. This pattern helps in reducing the surface area for reentrancy attacks on the contract. A reentrancy attack makes use of `fallback` and `receive` functions to recursively call `withdraw` functions on contracts. Reentrancy attacks will be shown with complete code in *Chapter 13, Writing Secure Contracts*. The following code listing shows an example of implementing the CEI pattern:

```
mapping (address => uint) balance;
function SimpleSendToAccount(uint amount) public returns (bool)
{
    if(balance[msg.sender] >= amount ) {
        balance[msg.sender] -= amount;
        if (msg.sender.send(amount) == true) {
            return true;
        }
    } else {
        balance[msg.sender] += amount;
        return false;
    }
}
```

In this example, a check is first made to see whether the caller has a sufficient balance to withdraw funds. If it has, it reduces the amount from the existing balance and uses the `send` method to transfer the amount to the target address.

It is worth noting that a lot of sources claim `send` is being deprecated, but I do not think it is. There are specific usages of the `send` function still available, such as sending an amount to multiple accounts. However, a new function, `transfer`, has been introduced to send Ether from one account to another; an even better solution is to ask other contracts and accounts to call for a specific method to withdraw the amount.

The address transfer method

The `transfer` method is similar to the `send` method. It is responsible for sending Ether or wei to an address. However, the difference here is that `transfer` raises an exception in the case of execution failure, instead of returning `false`, and all changes are reverted. Take a look at the `transfer` method in the following code snippet:

```
function SimpleTransferToAccount() public {
    msg.sender.transfer(1);
}
```

The `transfer` method is preferred over the `send` method, as it raises an exception in the event of an error. Those exceptions can be bubbled up in the stack and halt overall code execution.

The address call method

The `call` method has resulted in a lot of confusion among developers. There is a `call` method available via the `web3.eth` object, and there is also the `<<address>>.call` function. These are two different functions that have different purposes.

The `web3.eth.call` method can only make calls to a node it is connected to and is a read-only operation. It is not allowed to change the state of Ethereum. It does not generate a transaction, nor does it consume any gas. It is used to call the `pure`, `constant`, and `view` functions.

On the other hand, a `call` function provided by the `address` data type can call any function available within a contract. There are times when the interface of the contract, more commonly known as the **Application Binary Interface (ABI)**, is not available, and so the only way to invoke a function is to use the `call` method, using the address of the contract. Execution of the `call` method is not checked against the ABI, and thus it has the capability to call any function. There is no compile time check available for these calls, and it returns a tuple consisting of the Boolean status of the function call and any return value from the function.

Important Note

It is worth noting that it is not an ideal practice to call a `contract` function using the `call` method, as no checks and validation are involved.

Every function in a contract is identified at runtime using a 4-byte identifier. This 4-byte identifier is the trimmed-down hash of a function name along with its parameter types. After hashing the function name and parameter types, the first 4 bytes are considered as the function identifier. The `call` function accepts these bytes to call the function as the first parameter and the actual parameter values as subsequent parameters.

A `call` function *without* any function parameter is shown in the following code. Here, `SetBalance` does not take any parameter:

```
myaddr.call(bytes4(sha3("SetBalance()")));
```

A `call` function *with* a function parameter is shown in the following snippet. Here, `SetBalance` takes a single `uint` parameter:

```
myaddr.call(bytes4(sha3("SetBalance(uint256)")), 10);
```

It is also worth noting that the `send` function discussed earlier internally calls the `call` function.

We will use the next contract shown for the purpose of demonstrating usage of the `call` function. This `EtherBox` contract declares a state variable, `balance`, that stores the current available balance. There is a set of `getter` and `setter` functions that update the `balance` state variable with a new value or return the current value in the `balance` variable:

- `SetBalance`: This has a single state variable, and the purpose of this function is to add 10 in every invocation to the existing value of the state variable.
- `GetBalance`: This function is responsible for returning the current value of a state variable:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EtherBox {
    uint balance;
    event logme(string);
    function SetBalance() public {
```

```

        balance = balance + 10;
    }
    function GetBalance() public payable returns(uint) {
        return balance;
    }
}

```

Another contract named `UsingCall` is created to invoke methods on the `EtherBox` contract via the `call` function. Let's take a look at the following functions mentioned in the upcoming code example:

- `SimpleCall`: This function creates an instance of the `EtherBox` contract and converts it to an address. Using this address, the `call` function is used to invoke the `SetBalance` function on the `EtherBox` contract. The code for this is shown here:

```

function SimpleCall() public returns (uint) {
    EtherBox eb = new EtherBox();
    address myaddr = address(eb);
    bytes memory payload = abi.encode
        WithSignature("SetBalance()");
    (bool success, bytes memory returnData) =
        myaddr.call(payload);
    require(success);
    return eb.GetBalance();
}

```

- `SimpleCallWithGas`: This function creates an instance of the `EtherBox` contract and casts it into an address. Using this address, the `call` function is used to invoke the `SetBalance` function on `EtherBox`. The `call` function allows us to send custom units of gas along with the function payload. In the following code snippet, 200,000 units of gas are sent to the `GetBalance` function:

```

function SimpleCallwithGas() public returns (bool) {
    EtherBox eb = new EtherBox();
    address myaddr = address(eb);
    //status = myaddr.call.gas(200000)
    (bytes4(sha256("GetBalance()))));
    bytes memory payload = abi.encodeWith
        Signature("SetBalance()");
}

```

```
        (bool success, bytes memory returnData) =
            myaddr.call{gas: 200000}(payload);
        return success;
    }
```

- **SimpleCallWithValue**: This function creates an instance of the EtherBox contract and casts it into an address. Using this address, the call function is used to invoke the SetBalance function on EtherBox. The call function allows us to send custom units of gas and Ether along with the function payload. In the following code snippet, 200,000 units of gas and 1 Ether are sent to the GetBalance function:

```
function SimpleCallwithGasAndValue() public
returns (bool) {
    EtherBox eb = new EtherBox();
    address myaddr = address(eb);
    //status = myaddr.call.gas(200000).value(1)
    //bytes4(sha256("GetBalance()"));
    bytes memory payload = abi.encodeWith
    Signature("GetBalance()");
    (bool success, bytes memory returnData) =
        myaddr.call{gas: 200000, value: 1 ether}
        (payload);
    return success;
}
```

The entire code consisting of the previously listed function is available in the accompanying source code on GitHub. The overall structure of the contract is shown next:

```
contract UsingCall {
    constructor() payable {

}

function SimpleCall() public returns (uint) {}
function SimpleCallwithGas() public returns (bool) {}

function SimpleCallwithGasAndValue() public
```

```
    returns (bool) {  
}  
}
```

The address callcode method

This function is deprecated and will not be discussed here. This function is no longer available after the 0.5.0 version of Solidity. More information about `callcode` is available at <http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html>.

The address delegatecall method

This function is, again, a low-level function, responsible for calling functions in another contract. It is very similar to the `call` function discussed before; however, `delegatecall` is used along with libraries in Solidity. Libraries in Solidity do not declare state variables and comprise a set of reusable functions. However, these functions can access state variables belonging to the contract using the library. If contract A is using library B, library B does not have its own state variables, but it can use the state variables defined in contract A.

We have already seen the usage of libraries in the previous chapter. The `library` functions can be called directly using their name; however, they can also be called using the `delegatecall` method. This is shown in the following code snippet. Note that the usage is very similar to the `call` method discussed before. Instead of a contract address, the library address needs to be supplied for `delegatecall` to work. `delegatecall` will use the address of library to call the function supplied as part of the payload. The payload is the same `sum` function that we saw in the library section of the previous chapter. It accepts two arguments of the `uint256` type and returns a `uint256` value as well. It is important to note that ethers cannot be sent to library functions using the `delegate` code:

```
function SimpledelegateCallwithGas(address libAddress)  
public returns (bool) {  
bytes memory payload = abi.encodeWithSignature  
("sum(uint256,uint256)", 10,10);  
(bool success, bytes memory returnData) =  
libAddress.delegatecall{gas: 2000000}(payload);  
return success;  
}
```

The address staticcall method

The `address` type provides one more method, called `staticcall`. This function is similar to the `call` and `delegatecall` functions. It is a low-level function for calling functions in current or other contracts:

```
function SimpleStaticCallwithGas(address contractAddress)
    public returns (bool) {
    bytes memory payload = abi.encodeWithSignature
        ("sum(uint256,uint256)", 10,10);
    (bool success, bytes memory returnData) =
        contractAddress.staticcall{gas: 2000000}(payload);
    return success;
}
```

Being a method of the `address` type, it can call methods on contracts with the given address. The main difference between other methods and `staticcall` is that `staticcall` can only be used for the `pure`, `view`, and `constant` functions, which do not change the state variables. It will raise a `revert` opcode if the target function changes any state variable within the contract. It is important to understand the usage of the `call`, `delegatecall`, and `staticcall` functions. As mentioned before, these are low-level functions and should be used only in certain circumstances. These functions should be used when the source code of the target library or contract is not available. These libraries and contracts are deployed and provided to us with their addresses. Using these addresses, it is possible to invoke functions on them by making use of the low-level functions. In short, these functions should be used as a last resort when only an address to the library or contract is made available.

The `address` type is one of the most used types in smart contracts, as it helps in storing ownership details. It also helps in transferring ownership by using its methods, such as `transfer`, `send`, and `call`. There are a couple of special functions available within smart contracts, and they are discussed at length in the next section.

The fallback and receive functions

Both the `fallback` and `receive` functions are special type of functions available in Ethereum. The `fallback` functions were available in previous versions; however, the `receive` function is relatively new. They are special functions because we cannot invoke these functions directly by using their name. These functions do not accept any parameters or return any values. They must have external scope visibility, and they are defined without the `function` keyword. Both the `fallback` and `receive` functions might sound similar in nature; however, their usage and intent are completely different.

Let's understand the `fallback` function first. The `fallback` functions are invoked automatically by the **Ethereum Virtual Machine (EVM)** when it finds a function call with a name that does not exist within the contract. When a function call is made and that function name does not exist within the contract, the `fallback` function is invoked automatically. There can only be one `fallback` function within a contract. It can be virtual and overridden in the case of hierarchical contracts.

As mentioned, a `fallback` function does not have a function identifier or function name. It is always named `fallback` with an external scope visibility. An example of a `fallback` function is as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EtherBox {
    event logme(string);
    fallback() external payable {
        emit logme("fallback called");
    }
}
```

In the previous contract shown, a `fallback` function is defined with the `fallback` keyword, along with an `external` scope visibility and a `payable` modifier. The function is invoked within the EVM, and hence it should be `external`. A `fallback` function can be `payable` or non-`payable`. If we ignore the `payable` modifier, it becomes non-`payable`. The `payable` modifier determines whether the `fallback` function can accept any Ether or not. The `fallback` function shown previously just raises a new `logme` event with the `fallback called` value.

The `fallback` functions are executed under certain conditions. When these conditions are `true`, the `fallback` function is executed. The first rule is that a `fallback` function is executed when a function in a contract is invoked and the invoked function does not exist within the contract. To show this using an example, we will use the previously declared contract named `Etherbox`, consisting of the `fallback` function.

To invoke a function that does not exist within the `Etherbox` contract, we will use help of the `call` function of the `address` type. In this function, an instance of the `Etherbox` contract is created. As we know, the instance is nothing but the address of the deployed contract.

We get the address of the contract by explicitly casting the `instance` value into an address. We generate the payload to invoke a function using the `call` function by using the `abi.encodeWithSignature` function. Note that this function is passed to the `GetBalance1()` value as a parameter. This `GetBalance1` function does not exist within the `Etherbox` contract. In short, we are going to invoke the `GetBalance1` function on the `Etherbox` contract, and this should instead execute the `fallback` function:

```
contract UsingCall {
    function SimpleCallwithGasAndValueWithWrongName () {
        public returns (bool) {
            EtherBox eb = new EtherBox();
            address myaddr = address(eb);
            bytes memory payload = abi.encodeWithSignature
                ("GetBalance1()");
            (bool success, bytes memory returnData) =
                myaddr.call(payload);
            return success;
        }
    }
}
```

Executing the `SimpleCallwithGasAndValueWithWrongName` function in turn should invoke the `fallback` function on the `Etherbox` contract, as shown here:

```
{
    "from": "0xE2D48D49Ea51c83DFAF7f91AB67180980aC45bc2",
    "topic": "0x23f195eb3331002b52c02714937ffcf2645ce7be57859c8d68705ad4254f52d3",
    "event": "logme",
    "args": {
        "0": "fallback called"
    }
}
```

Figure 7.1 – The event output from the fallback function

The `fallback` function can also be invoked when a contract receives any Ether. This usually happens when using the `send` or `transfer` function of the `address` type or when the `SendTransaction` function available in `web3` is used to send Ether to a contract. However, in order to get executed in the event of receiving Ethers, the `fallback` function should be `payable`; otherwise, it will not be able to accept the Ether and will raise an error, as shown in the following code listing:

```
function SendEther() public returns (bool) {
    EtherBox eb = new EtherBox();
    address payable myaddr = payable(eb);
    myaddr.transfer(10000000000000000000);
    return true;
}
```

This function is sending 1 ether in wei denomination to `Etherbox` contract. This implements the `fallback` function, which gets executed, as shown here:

```
{
    "from": "0xFbCA1d693A100774A2443cc1D41bD1a3D5295112",
    "topic": "0x23f195eb3331002b52c02714937ffcf2645ce7be57859c8d68705ad4254f52d3",
    "event": "logme",
    "args": {
        "0": "fallback called"
    }
}
```

Figure 7.2 – The event output from the fallback function

The next important question to be answered is how much gas is needed to execute the `fallback` function. Since it cannot be called explicitly, gas cannot be sent explicitly to this function. Instead, the EVM provides a fixed stipend of 2,300 gas to this function. Any consumption of gas beyond this limit will raise an exception, and the state will be rolled back after consuming all the gas that was sent along with the original function. It is, therefore, important to test your `fallback` function to ensure that it does not consume more than 2,300 units of gas.

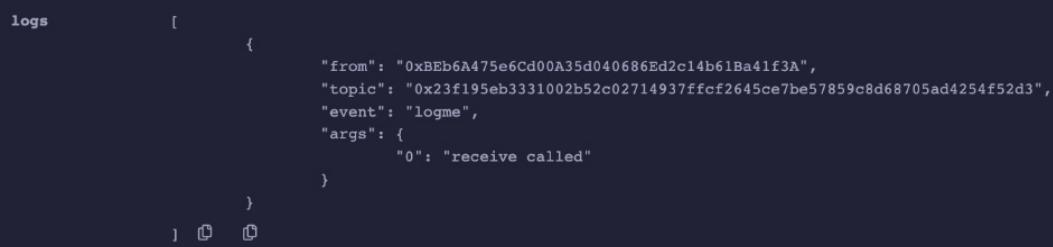
It is also worth noting that the `fallback` functions are one of the top causes of security lapses in smart contracts. It is very important to test these functions from a security perspective before releasing a contract into production.

As mentioned before, the `receive` function is relatively new in Solidity. You might have noticed that the `fallback` function gets executed on multiple conditions. It can become difficult to write code for each of these conditions within a single `fallback` function. It is for this reason that a separate `receive` function was added in Solidity so that a `fallback` function should be executed only if a function signature is not implemented within a contract, and such a function is invoked while the `receive` function is executed automatically while receiving Ethers from external sources. The `receive` function should be payable for receiving Ether and follows similar syntax to that of the `fallback` function.

An example of the `receive` function is shown next:

```
receive() external payable {
    emit logme("receive called");
}
```

The `receive` function, in this case, only emits a `logme` event with custom string data into it:



```
logs [ { "from": "0xEBb6A475e6Cd00A35d040686Ed2c14b61Ba41f3A", "topic": "0x23f195eb3331002b52c02714937ffcf2645ce7be57859c8d68705ad4254f52d3", "event": "logme", "args": { "0": "receive called" } } ]
```

Figure 7.3 – The event output from the `receive` function

A complete example implementing both the `fallback` and `receive` functions can be found in the `FallbackAndReceive.sol` file, available in the source code of the book.

The implementation uses the previously shown contract related to the address `call` function. However, this time, we have implemented a payable `fallback` function and payable `receive` function in the `EtherBox` contract, whose entire purpose is to raise an event. The event is also declared within the function. We will look at events in more depth in the next chapter.

When you execute each of the methods in the `UsingCall` contract, you should notice that the `fallback` function is not invoked for any of the functions, apart from one that does not call a correct function.

Summary

Once again, this was a dense chapter that focused primarily on the usage of functions along with modifiers and the way they help in writing code that improves overall readability and logic flow. Solidity provides special functions, the `fallback` and `receive` functions, within contracts, and they were covered with examples of their usage.

Functions can be constrained using the `pure`, `constant`, and `view` modifiers, which limit the activity possible within a function. State variables and functions can have different visibility scopes such as private, internal, public, and external (although state variables are never external) that constrain their visibility and limits them to only certain types of callers.

The functions related to the `address` type can be intimidating, especially when you consider their multiple variations and their relationship with the `fallback` functions. If you are implementing a `fallback` or `receive` function, remember to pay special attention to testing them, especially from a security point of view. You should also pay special attention when using low-level Solidity functions such as `send`, `call`, and `transfer`, as they invoke the `fallback` function implicitly. Always try using contract functions that use the ABI, as it ensures compile-time checks rather than no checks at all when using low-level functions.

In the next chapter, we will dive deep into the world of events, logging, and exception handling in Solidity. Stay tuned!

Questions

1. What are the different types of scope visibility modifiers available for a function? Is there any difference between state variables and functions with regard to scope visibility?
2. What is the difference between the `fallback` and `receive` functions?

Further reading

- The official Solidity documentation on contracts provides additional information on all the topics covered in this chapter: <https://docs.soliditylang.org/en/v0.8.12/contracts.html>.

8

Exceptions, Events, and Logging

Writing contracts is the fundamental purpose of Solidity. However, writing a contract demands sound error and exception handling skills. **Errors** and **exceptions** are the norms in programming, and Solidity provides ample infrastructure for managing both. Writing robust contracts with proper *error and exception management* is one of the requirements for any functional smart contract. **Events** are another important construct in Solidity. For all the topics that we've discussed so far, we've seen a caller that invokes functions in contracts; however, we have not discussed any mechanism through which a contract notifies its caller and others about changes in its state and otherwise. This is where events come in. Events are a part of event-driven programs where, based on changes within a program, they proactively notify their caller about the changes. The caller is free to use this information or ignore it. Finally, both exceptions and events, to a large extent, use the logging feature provided by the EVM. All these are enterprise features and any serious smart contract implementation should use these features to ensure that it can adjust to unexpected environmental issues and be able to notify administrators about potential runtime issues impacting the execution of the smart contracts.

In this chapter, we will cover the following topics:

- Understanding exception handling in Solidity
- Error handling with `require`
- Error handling with `assert`
- Error handling with `revert`
- Custom errors with `revert`
- `try-catch` exception handling
- Understanding events
- Declaring an event
- Using an event
- Writing to logs

Technical requirements

There are not many technical requirements for this chapter. The following will suffice for this chapter:

- An operating system – Windows, macOS, and Linux would work equally well.
- The Chrome browser for navigation to [remix.org](https://remix.ethereum.org) and writing smart contracts.

All code from this chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter08>.

Exception handling

Errors are a fact of life in the programming world. They are often inadvertently introduced while writing contracts and therefore writing error-free contracts is a desired skill. These errors can occur either at design time or runtime. Solidity is compiled into bytecode as part of compilation and the compiler checks for any syntax errors during this process. Runtime errors, however, are more difficult to catch and generally occur while executing contracts. It is important to test the contract for possible runtime errors, but it is more important to write defensive and robust contracts that take care of both design-time and runtime errors. Some examples of runtime errors are **out-of-gas errors**, **divide-by-zero errors**, **data-type-overflow errors**, and **array-out-of-index errors**.

Until version 4.10 of Solidity, there was a single `throw` statement available for error handling. Developers had to write multiple `if ... else` statements to check the values and `throw` in the case of an error. The `throw` statement consumed all the provided gas and reverted to the original state. This is not an ideal situation for architects and developers as unused gas should be returned to the caller.

From version 4.10 of Solidity, newer error handling constructs were introduced and `throw` was made obsolete. These were the `assert`, `require`, and `revert` functions. In this section, we will look into these error handling mechanisms.

Before we go into any details about the use of Solidity support for exception handling, let's understand what happens in the event of an exception without the use of any exception handling mechanism.

We will write a small smart contract that contains mathematical calculations but without any exception handling code. In this smart contract, we will write a function that will accept an input from the user in the form of an argument and use this argument value for processing mathematical calculations.

The entire code listing is provided here:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract NoExceptionHandlingContract {
    function NoExceptionHandlingFunction(uint256 myNumber)
        public
        payable returns (bool status, uint256) {
        uint256 tempNumber = 200/myNumber;
        if(tempNumber > 10)
            status = true;
        else
            status = false;
        return (status, tempNumber);
    }
}
```

Since the argument value is supplied by a caller, it is difficult to predict its value. The logic written within the function is quite simple. It takes an argument value, divides the input value with 200, and stores the result in a local variable. Later, the function checks whether the result produced a number greater than 10 or less than 10 and accordingly returns a Boolean value – `true` or `false`.

If you try to execute this function from Remix, you will get an exception, as shown here. The exception message is `VM error: revert`:

```
transact to NoExceptionHandlingContract.NoExceptionHandlingFunction errored: VM error: revert.

revert
      The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance.
Debug the transaction to get more information.
```

Figure 8.1 – A revert exception thrown from a contract function without any exception handling
The code listing along with the debug information is shown in the following screenshot:

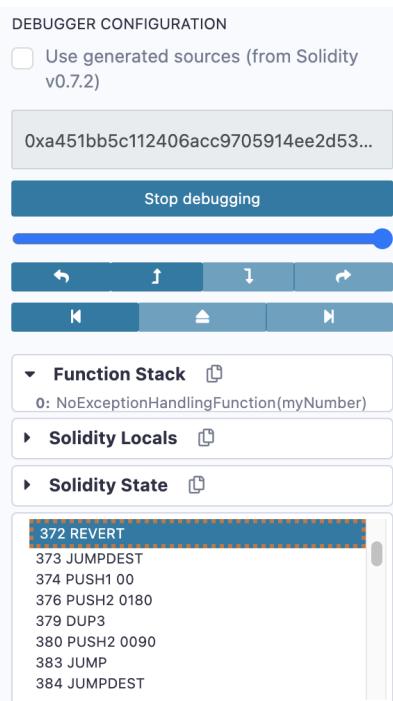


Figure 8.2 – REVERT opcode as seen in Remix Debugger for an exception
Notice that REVERT is visible in the Opcode window.

Require

Recently, Solidity launched the `try-catch` block; however, earlier, there were no exception handling blocks that are normally available in other programming languages. If an exception was encountered, it would bubble up to the caller.

Not using exception handling renders code vulnerable to unexpected events and leads to non-favorable outcomes. Implementing exception handling, in general, is good practice as it leads to writing defensive code against any unexpected event.

It is always better to write robust code using defensive and secure practices. Defensive code means writing code that validates the current global and local state along with incoming argument values before executing the main code logic.

The use of `require` helps in validating the contextual state along with any incoming argument values before executing any logic within the function.

The word `require` means fencing and declaring constraints. Implementing a `require` condition means declaring prerequisites for running the function that must be satisfied before moving forward in code execution.

The `require` function accepts two arguments. The first argument is a conditional statement – a statement that either evaluates to `true` or `false`. If the evaluation of the statement is `false`, an exception is raised and execution is halted. The unused gas is returned to the caller and the state change is reversed. The second argument is optional and accepts a `string` value that is returned to the caller as part of the exception reason. The `require` statement results in the `revert` opcode, which is responsible for reverting the state and returning unused gas. However, it is important to note that `require` does not return already-consumed gas and so it is important that `require` functions, generally, are used at the beginning of the function. It also does not consume or confiscate any ether in the event of failure.

`require` returns an exception of the `Error(string)` type to the caller in the case of failing conditions. The last section of this chapter discusses exception handling using the `try-catch` block and catching the `Error(string)` error type as part of the `catch` clause is covered there.

The `revert` opcode raises an exception that bubbles up to the caller. It also reverts state changes to their original state. The state change concept is important to understand. Within a function, if a modification to global state or storage variables is made prior to the `require` statement, all those changes revert to the original value.

The state change relates only to storage variables as they are persisted in Ethereum global state.

The same contract shown before being rewritten using the `require` function as part of defensive coding practice is shown in the following code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract SimpleRequireContract {
    function SimpleRequireFunction(uint256 myNumber) public
        payable
        returns (bool status, uint256) {
        require( myNumber > 0 );
        uint256 tempNumber = 200/myNumber;
        if(tempNumber > 10)
            status = true;
        else
            status = false;
        return (status, tempNumber);
    }
}
```

This has an additional line of code toward the beginning of the function compared to `NoExceptionHandlingContract` shown before. Note that the value of the incoming argument is checked first using the `require` function and only if it returns `true` is the next set of code statements executed.

The following code illustrates additional examples of the `require` statement. The `RequireValidInt8` function uses a couple of `require` functions. The first `require` function has a conditional statement that validates whether the supplied argument value is greater than or equal to 0. If an exception is raised, execution is halted, the state is reverted, and the caller gets an exception comprising the reason `Number cannot be less than 0`. However, if the argument value supplied is greater than 0, the next `require` function executes. This `require` function checks the upper boundary for the argument value and, depending on the result, either continues further or reverts to the caller with an exception and exception reason:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract RequireUsageContract {
```

```
function RequireValidInt8(uint256 myNumber) public
    returns (uint256) {
    require( myNumber >= 0, "Number cannot be less
        than 0" );
    require( myNumber <= 255, "Number cannot be greater
        than 255" );

    return myNumber;
}

function RequireIsEven(uint256 myNumber) public returns
(bool) {
    require( myNumber % 2 == 0 );

    return true;
}

}
```

The `RequireIsEven` function is similar to the `RequireValidInt8` function. It uses the `require` function to validate whether the supplied value is an even or odd number. If the number is even, the function succeeds; otherwise, it returns an exception without any exception reason.

`require` should be used to validate all incoming arguments. If the arguments are simple signed or unsigned integers, checks for the upper- as well as lower-limit values should be done. Depending on the functional need, checks for all possible outliers that might raise an error should be done. Integer overflows and underflows should also be checked if the incoming arguments are used for calculation. If the argument is of the `address` type, checks for valid addresses that are not null, zero addresses, and so on can be performed.

The next contract shows checking for address values along with integer overflow using the `require` function:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MoreRequireUsageContract {
```

```
uint256 _totalSupply = 1000000000000000;
address _king = 0x5B38Da6a701c568545dCfcB03FcB875
f56beddC4;

function MoreRequireUsageFunction(uint256 myNumber,
address owner) public payable returns (bool status)
{
    require ( _totalSupply + myNumber > _totalSupply );
require( myNumber >= 0 );

    require ( owner != address(0) );
    require ( owner == _king );

    return true;
}

}
```

The previous contract declares two state variables – `_totalSupply` of type `uint256` and `_king` of type `address`. The `_totalSupply` variable is assigned the literal `1000000000000` value and `_king` is assigned a static address.

In the `MoreRequireUsageFunction` function, there are two incoming arguments – again, one of the unsigned integer type and the other being an address. The first two `require` condition checks are related to integers. The first check is related to integer overflow in which it checks that the incoming number, when added to `_totalsupply`, does not overflow, and the second one checks whether the value is greater than 0.

The next set of `require` conditions is related to the `address` type. It checks whether the address is equal to 0 and that the address is the same as the address stored in the `storage` variable.

If any of the `require` conditions fail, the function stops execution; if there is a state change, it is reverted, and additional gas and all supplied ethers are returned to the caller.

The previous example is a great way to write defensive code using the `require` function and the same set of `require` conditions can be repeated in multiple functions as well. There should be a way to reuse a group of `require` conditions together with multiple functions. Function modifiers are an appropriate tool to reduce code duplication and also help in implementing modular code.

We can use function modifiers and implement all the require conditions in them, and eventually use them with appropriate functions. You can see this in the next code listing:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract RequireRefactoredContract {

    uint256 _totalSupply = 1000000000000;
    address _king = 0x5B38Da6a701c568545dCfcB03FcB875
        f56beddC4;

    modifier CheckAddressAndIntegers( uint256 myNumber,
        address owner ) {
        require( myNumber >= 0 );
        require ( _totalSupply + myNumber > _totalSupply );
        require ( owner != address(0) );
        require ( owner == _king );
        _;
    }

    function RequireRefactoredFunction(uint256 myNumber,
        address owner) public CheckAddressAndIntegers
        (myNumber, owner) returns (bool status) {

        return true;
    }
}
```

The contract shown in the preceding code does exactly the same thing as the one shown before it. However, there is a big change from a code refactor point of view. We have introduced a modifier named `CheckAddressAndIntegers` that accepts two arguments and it includes all the `require` conditions within it. We have also associated this modifier with the `RequireRefactoredFunction` function while passing both the arguments to the modifier.

The result of both the previous smart contracts is the same; however, in the latter contract, the modifier can potentially be used with multiple functions and those functions will get the benefits of all the `require` conditions without cluttering their code with repeated statements. The functions can focus on writing their business logic and leave the validation to the modifier. It is much easier to read these functions and also manage them in the future.

The `require` function should be used to validate all arguments and values that are incoming to the function. This means that if another function from another contract or function in the same contract is called, the incoming value should also be checked using the `require` function.

The `require` function should be used to check the current state of variables before they are used. If `require` throws an exception, it should mean that the values passed to the function were not expected by the function and that the caller should modify the value before sending it to a contract.

Assert

`Assert` is similar to `require` in many ways. It also provides a way to write and validate conditions. `Assert` is used for validating and checking the current state before moving to the next line of code. Generally, `require` functions are written at the beginning of the function. `require` can also be used to validate return values from other functions. This means it can be written anywhere within the function. However, `Assert` is used to validate the internal local state of the function. `Assert` is used to ensure the state changes are consistent before and after the function execution. It validates whether the state transition is according to rules and that there are no inconsistencies with regard to state change. While executing a function, the state should undergo expected changes but there might be inconsistencies that can render the state not usable. In such cases, before leaving the function, `Assert` can check whether the state is as desired. If not, it can raise an exception, which will bubble up to the caller.

`Assert` reverts the state to the original state and returns any ethers sent to it. However, it consumes all the *gas* supplied to it. The Opcode generated by `Assert` is also different from the `require` function. It accepts a conditional statement, which should evaluate to either a `true` or `false` value. Based on that, the execution will either move on to the next statement or throw an exception.

`Assert` returns an exception of the `Panic(uint256)` type to the caller in the case of failing conditions. The last section of this chapter discusses exception handling using the `try-catch` block and there, we will cover catching the `Panic(uint256)` error type as part of the catch clause:

```
An example of a smart contract using Assert is shown next.
```

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Order {
    uint256 _balance ;

    constructor() public payable {
        _balance = msg.value;
    }

    function NewOrder(address payable _seller) public
        returns (uint256)  {
        uint256 oldBalance = _balance;
        _balance = _balance - 1000000000000000000;
        _seller.send(1 ether);
        assert (_balance == (oldBalance -
            1000000000000000));
    }
}
```

In the previous example, an assertion is made toward the end of the function to check whether the current balance matches the expected value. If it does, the assertion is successful; otherwise, it raises an exception. Another usage of `Assert` is to check the overflow and underflow of the value for the integer data type. It is always possible to result in an overflow when adding values from two variables. This overflow can be verified using the `Assert` statement.

The following code illustrates another use of the `assert` function:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract OverflowCheckContract {

    function OverflowCheckFunction(uint256 myNumber)
        public
        returns (uint8) {
        require( myNumber >= 0 );
        require( myNumber <= 255 );

        uint8 val = 20;

        assert ( myNumber + val <= 255 );

        return uint8(myNumber + val);
    }

}
```

While `require` should be used for values coming as arguments, `assert` should be used to validate the current state of state variables, local variables, and values returned from other function calls. It is easier to envision `assert` as dealing with runtime exceptions, which are difficult to predict. The `assert` statement should be used when you think that a current state has the potential to become inconsistent.

Revert

Apart from the `assert` and `require` keywords for writing defensive code, Solidity also provides the `revert` function. This is an out-of-the-box function that can be used anywhere from within a function for business functionality-related exceptions. The `revert` function generates the `revert` opcode and it stops the execution of the function while rolling back any state changes to their original state and returning any unused gas and ether to the caller. In fact, the `require` statement conditionally generates the `revert` opcode internally. However, it is important to understand the difference between the `require` and `revert` functions.

Let's understand the execution of the following code in the context of the `revert` function:

```
uint256 tempNumber = 200/myNumber;
```

When the value of `myNumber` is 0, the calculation on the right will result in an exception and this will consume all the supplied gas. This is not an ideal situation. However, it is possible to wrap this statement within the `require` function, as shown next:

```
uint256 tempNumber;
require(tempNumber = 200/myNumber);
```

Now, the condition will be evaluated within the context of the `require` function, and as a result, a managed exception is raised that returns any unused gas and ether. It will also revert any state changes. It is not possible to wrap the `revert` statement within an `if` condition, as shown here:

```
if (tempNumber = 200/myNumber) {
    revert("divide by zero not accepted!!");
}
```

It is not possible because as soon as an `if` condition is evaluated, an exception is raised and `revert` will not get a chance to execute.

When `revert` was launched in version 0.4.10, it was released as a function with two possibilities – an empty `revert` or a `revert` that accepts a string to return custom error messages. With the release of Solidity 0.8.4, a new `revert` statement was launched that works with the `CustomError` object. We will see an example for both the `revert` statement (release 0.8.4) and the `revert` function (release 0.4.10).

The `revert` function provides an option to return a value such as an exception message with a status code with any meaning to the caller such that it can take different actions based on different codes or it can just be an empty call. The code shown next is an example of using `revert` with a string description in smart contracts:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract OrderContract {
    uint256 _balance ;
```

```
constructor() public payable {
    _balance = msg.value;
}

function NewOrder(address payable _seller) public
returns (uint256) {
    uint256 oldBalance = _balance;
    _balance = _balance - 10000000000000000;
    _seller.send(1 ether);
    if (_balance == (oldBalance - 10000000000000000)) {
        revert ("balances do not match");
    }
}
```

The previous code shows a smart contract implementation with a simple `NewOrder` function. This function reduces the balance by 1 ether and while it does so, it also checks for integer underflow using an `if` conditional statement. If it finds that there is integer underflow, it raises an exception using the `revert` function and provides a description as well.

Examples of using `revert` include calling an external function in another contract and getting a different value than expected. In such cases, `revert` can be used. It is used to author code with more complex and deep multiple conditional statements.

Another example of the usage of `revert` is shown in the next smart contract. In the following example, an exception is thrown when the incoming value is checked using the `if` condition; if the `if` condition evaluation results in `false`, it executes the `revert` function. This results in an exception and execution stops, as shown in the following code:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract RevertContract {

    function RevertValidInt8(uint256 myNumber) public
    returns (uint8) {
        if (myNumber < 0 || myNumber > 255) {
            revert ("not a valid int8 value");
    }
}
```

```

    }

    return uint8(myNumber);
}

}

```

In the preceding example, we see that the parameter is checked explicitly using the `if` conditional statement. If it evaluates to `false`, it would raise an exception using `revert` and return a custom message along with it.

Now, let's look into the usage of a `revert` statement that can pass a custom error object to its caller instead of a string description. This is the preferred way to use `revert` in smart contracts over the string descriptions sent as arguments.

Solidity allows us to define new types of errors. This is very similar to the way we define new types of events in Solidity. Solidity has introduced a new `error` keyword that helps in defining a new error type. These are also called custom errors. The syntax for defining a new custom error type is as follows:

```
error <>error object name>>(parameter1,...parameterN);
```

For example, an error type can be defined as shown next:

```
error DividebyZero(uint256 numerator, uint256 denominator)
```

It is important to note that custom error types are not declared within a contract. They can be defined within the file containing the contract but outside the contract. The error type just defined can be used along with the `revert` statement, as shown next:

```
revert DividebyZero(100,0);
```

Let's understand custom errors along with `revert` through a complete example. We can use the same contract used for demonstrating `revert` using string descriptors. The following code listing defines a new custom error type, `InvalidIntegerValue`, which accepts a single `uint256` argument. Within the `RevertValidInt8` function, a conditional check is made for the argument value to be between 0 and 255. If the value is not in this range, it raises an exception using the `revert` statement, passing in an instance of a custom error and values for its arguments:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

error InvalidIntegerValue(uint256 intValue);
```

```
contract RevertContract {
    function RevertValidInt8(uint256 myNumber) public returns
        (uint8) {
        if (myNumber < 0 || myNumber > 255) {
            revert InvalidIntegerValue(myNumber);
        }
        return uint8(myNumber);
    }
}
```

From Remix Editor, passing a value of 100 to the `RevertValidInt8` function is executed successfully; however, passing a value of 256 results in an exception with a rollback of state, unused gas, and ethers. The output of `revert` is shown in the following screenshot:

```
transact to RevertContract.RevertValidInt8 pending ...

transact to RevertContract.RevertValidInt8 errored: VM error: revert.

revert
      The transaction has been reverted to the initial state.
      Error provided by the contract:
      InvalidIntegerValue
      Parameters:
      {
          "intValue": {
              "value": "256"
          }
      }
      Debug the transaction to get more information.
```

Figure 8.3 – Output using a custom error with the revert statement

One important element that was purposely not mentioned before (for the purpose of facilitating your learning) was that from Solidity 0.8.4, either an empty argument or a string description, it generates an error object of the `Error(string)` type internally and sends it back to the caller. This is equivalent to Solidity internally defining an error, as shown here:

```
error Error(string description)
```

Now that we understand all three, `require`, `assert`, and `revert`, comprehensively, it's time to learn a new way of implementing exception handling in Solidity using `try-catch` blocks.

Try-catch in Solidity

Solidity only had limited exception handling capabilities in the form of `require`, `assert`, and `revert` functions prior to version 0.6.0. Solidity introduced the much-needed `try-catch` functionality for implementing exception handling in contracts. The concept of `try-catch` is quite simple. There is a `try` block associated with a function call. This function should be an external function. If the called external function does not produce an error, the `try` block is executed. Execution of code in the `try` block means there have been no exceptions. Immediately following the `try` block is the `catch` block and the code within the `catch` block is executed in the case of `revert` (exception) from the target function. The `try` and `catch` blocks execution are mutually exclusive. Either the `try` block or `catch` block will get executed.

The syntax of `try-catch` is shown next:

```
Try <<function call>> returns (<<return values from
    function call>>) {
    ..code with access to return values
}
catch Error(string memory errorReason) {
}
Catch Panic(uint256 panicode) {
}
Catch (bytes memory lowLevelData) {
}
```

There is a single `try` block along with multiple optional `catch` blocks. There should be at least one `catch` block along with the `try` block. `try` is applied to an external function call that might or might not return any return value. Capturing the return value from the function call is optional; however, if the return value is needed within the `try` block (for raising events with contextual data, computation, business rules, and so on), the return value should be declared.

The target function can execute successfully or generate an error. An error is generated because of exceptions from internal execution, such as divide by zero or stack overflow, or the code raises it due to failing require, assert, or revert conditions. If the target function does not return an error the code in the try block is immediately executed with the option to access the return value from the target function; otherwise, any of the catch blocks are executed depending on the type of exception.

In ordinary cases with no try-catch block, in the case of an error, the execution halts, the unused gas and ethers are returned, and the state is rolled back. Try-catch ensures that the execution does not halt, and it continues with its next logical statement. The try-catch clause consumes the error and takes appropriate actions based on the error type. It is important to note that the scope of try is limited to the external function call associated with it. Errors arising from code in the try block do not come in its ambit and they will halt the execution. The same is the case with the catch block.

One of the major benefits of the try-catch way of exception handling is that in the case of an exception, the execution is not necessarily halted. The execution can continue depending on the implementation logic. It is also possible to have multiple catch blocks for a try block. The most specific catch should be placed before the generic catch blocks.

The most common catch clause is without any error type and arguments. This is the most general and catch-all type of catch clause. This should generally be the last catch clause in the case of multiple catch block implementations; otherwise, it would lead to unreachable code.

In the case of capturing the details of errors, a catch clause with an argument can be used. The only difference from the previous catch clause is the availability of error information within the catch block. This is also a catch-all type of catch clause. There are specific catch clauses are well – error and panic types.

The Error (string memory errorReason) catch clause is executed in response to a revert from the target function. Both of the flavors of revert – revert with string description and empty revert – are caught produces an Error object. Even a failing require function generates the same Error object. It is important to note that custom errors are not caught using this catch clause.

The Panic (uint256 panicode) catch clause is executed in response to a failing assert function or from an internal runtime execution, such as stack-overflow or divide-by-zero errors.

It's time to see an implementation of `try-catch`. We need a contract with a function that would be the target function with the `try` clause. A simple function, `Divide2Nums`, within the `Simpledivision` contract accepts two arguments and divides the first argument with another one. Note that the function is not validating the arguments. This code can result in a divide-by-zero error:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract SimpleDivision {
    function Divide2Nums(uint256 numone, uint256 numtwo)
        public returns (uint256) {
        return numone/numtwo;
    }
}
```

Next, we can focus on our first code listing with the `try-catch` clause. A simple `GetDivision` function within the `TryCatchExample` contract accepts two arguments. The code in this function is wrapped with a `try` clause. The `try` clause is associated with a call to the `Divide2Nums` function in the `SimpleDivision` contract. The `SimpleDivision` contract is deployed in the constructor using the new pattern. The return value from the `Divide2Nums` function is also captured for logging purposes. The general-purpose catch clause with variable capturing error information is implemented. The code for both the `try` and `catch` blocks emits their respective events:

```
contract TryCatchExample {
    SimpleDivision sa;

    event myevent(uint256, string);
    event mybytes(bytes);

    constructor() {
        sa = new SimpleDivision();
    }

    function GetDivision(uint256 a, uint256 b) public
        returns (bool) {
        try sa.Divide2Nums(a,b) returns (uint256 ab) {

```

```
    emit myevent(b, "pure success");

    return true;
}

catch (bytes memory lowLevelData) {
    emit mybytes(lowLevelData);

    return false;
}

}
```

Execution of the `GetDivision` function with values 100 and 5 is executed successfully and the try block is executed. However, the same function execution fails with values 100 and 0 and the catch block is executed. The failure does not halt the execution of the contract, as shown in the following screenshot:

Figure 8.4 – Output showing a general catch-all catch clause getting executed

Say we add an additional `require` statement to the `Divide2Nums` function, as shown here:

```
function Divide2Nums(uint256 numone, uint256 numtwo)
public returns (uint256) {
    require(numtwo > 0, "numtwo is less than or equal
        to zero");
    return numone/numtwo;
}
```

Instead of the general-purpose catch clause implementing the `Error(string memory errorReason)` catch clause, the `Error` catch clause gets executed if the function's second argument is 0 or less than 0:

```
function GetDivision(uint256 a, uint256 b) public returns  
    (bool) {
```

```

try sa.Divide2Nums(a,b) returns (uint256 ab) {
    emit myevent(b, "pure success");
    return true;
}
catch Error(string memory reason) {
    emit myevent(b, reason);
    return false;
}

}

```

The output for this execution with 100 and 0 as arguments is shown next:

```

{
    "uint256 a": "100",
    "uint256 b": "0"
} ↵

{
    "0": "bool: false"
} ↵

[
    {
        "from": "0x38cB7800C3Fddb8dda074C1c650A155154924C73",
        "topic": "0xd53add2ade84cf8bf24f13191e28afac3c84005356a8933e4a50a85129d2774",
        "event": "myevent",
        "args": {
            "0": "0",
            "1": "numtwo is less than or equal to zero"
        }
    }
]

```

Figure 8.5 – Output showing the Error catch clause being executed in response to failing the require function

Similarly, if `Assert` fails, the `Panic(uint256 panicode)` catch clause can be used. It is possible to have multiple catch clauses with the same `try` clause, as shown next:

```

function GetDivision(uint256 a, uint256 b) public
returns (bool) {
try sa.Divide2Nums(a,b) returns (uint256 ab) {
    emit myevent(b, "pure success");
    return true;
}

```

```
        catch Error(string memory reason) {
            emit myevent(b, reason);
            return false;
        }
        catch Panic(uint errorCode) {
            emit myevent(errorCode, "pure failure");
        }
        catch (bytes memory lowLevelData) {
            emit mybytes(lowLevelData);
            return false;
        }
    }
```

try-catch clauses are not only used for calling external functions. They can also be used for deploying contracts using a new pattern and any failure is subject to the same rules, as discussed before.

After understanding the different mechanisms to implement exception handling in Solidity, it is time to understand the concepts of events and logging in the next section.

Events and logging

We have seen the use of events in previous chapters without going into any detail. In this section, however, we will look at events in more depth. Events are well known to event-driven programmers. Events refer to the notification of state change or an act of action in contracts to external entities such that they can react and take related actions.

Events help us to write asynchronous applications. Instead of continuously polling the Ethereum ledger for the existence of a transaction and then blocking with certain information, the same procedure can be implemented using events. This way, the Ethereum platform will inform the client if an event has been raised. This helps when writing modular code and also conserves resources.

It is to be understood that events do not directly associate themselves with any external entities. As we know that function execution in Ethereum happens much later while a block is getting added to the chain, the function cannot have any visibility of external entities. Instead, the events write log entries as logs within the block as part of the `logsBloom` attribute. Events are part of contract inheritance, where a child contract can invoke events. Event data is stored along with block data. The `logsBloom` value is the event data, as shown in the following screenshot:

Figure 8.6 – Ethereum block showing events data in the logsBloom attribute

Declaring events in Solidity is very similar to declaring a function. However, events do not have a body. A simple event can be declared using the `event` keyword followed by an identifier and a list of parameters, as shown in the following code:

```
event LogFunctionFlow(string);
```

In the preceding line of code, `event` is the keyword used for declaring an event followed by its name and a set of parameters. The values supplied to these parameters are written to the logs. In this case, any string value can be sent with the `LogFunctionFlow` event.

Using an event is quite simple. Simply invoke an event using the `emit` keyword followed by the name of the event and pass on the arguments it expects. For the `LogFunctionFlow` event, the invocation looks as follows, which is similar to a function call with parameters. This raises the event with the given text and this text is logged as part of the block:

```
emit LogFunctionFlow("I am within function x");
```

The following code snippet shows an event in use:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EventContract {
    event LogFunctionFlow(string);
    function ValidInt8(uint256 myNumber) public returns
```

```

        (uint8)  {
            emit LogFunctionFlow("within function ValidInt8");

            if (myNumber < 0 || myNumber > 255) {
                revert ("not a valid int8 value");
            }

            emit LogFunctionFlow("Value is within expected
                range !");
            emit LogFunctionFlow("returning value from
                function");
            return uint8(myNumber);
        }
    }
}

```

In the previous code, an event, `LogFunctionFlow`, is declared with a string as its sole parameter. The same event is invoked multiple times from the `ValidInt8` function, providing text information during various stages within the function.

Executing this contract in Remix shows the result, which contains three logs with event information, as shown in the following screenshot:

```
[ { "from": "0xE3Ca443c9fd7AF40A2B5a95d43207E763e56005F", "topic":
"0xb5b850034705238ab6360bcd803e9e3dcraf926c812b20193e2e99a5918d47b0", "event": "LogFunctionFlow", "args": { "0": "within function
ValidInt8" } }, { "from": "0xE3Ca443c9fd7AF40A2B5a95d43207E763e56005F", "topic":
"0xb5b850034705238ab6360bcd803e9e3dcraf926c812b20193e2e99a5918d47b0", "event": "LogFunctionFlow", "args": { "0": "Value is within
expected range !" } }, { "from": "0xE3Ca443c9fd7AF40A2B5a95d43207E763e56005F", "topic":
"0xb5b850034705238ab6360bcd803e9e3dcraf926c812b20193e2e99a5918d47b0", "event": "LogFunctionFlow", "args": { "0": "returning value
from function" } } ]
```

Figure 8.7 – Event data seen as a log attribute in Remix

Events can also be watched from custom applications and decentralized applications using the `web3.js` library. The detailed documentation on `web3.js` is available at <https://web3js.readthedocs.io/en/v1.7.0/>.

Important Note

Events can be filtered using parameter names.

The following two methods allow us to watch for events:

- **Watching individual events:** In this method, using `web3.js`, individual events from contracts can be watched and tracked. When the exact event is fired from a contract, it helps execute a function in the `web3` client. An example of watching an individual event is shown in the following code:

```
var myEvent = instance.ageRead({fromBlock: 25000, toBlock: 'latest'});
myEvent.watch(function(error, result){
  if(error) {
    console.log(error);
  }
  console.log(result.args)
});
```

Figure 8.8 – Reading only the AgeRead event using web3 from block 25000 until the latest block

Here, `ageRead` is the name of the event we are interested in and watching for. We read `fromBlock` number 25000 until the `latest` block. First, a reference to the `ageRead` event is made and a watcher is added to the reference. The watcher takes a `promise` function that is executed whenever the `ageRead` event is fired.

- **Watching all events:** In this method, using `web3`, all events from contracts can be watched and tracked. When an event is fired from a contract, it notifies and helps to execute a function in the `web3` client in response. In this case, the event can be filtered using an event name. An example of watching all events is shown in the following screenshot:

```
var myEvent = instance.allEvents({fromBlock: 24000, toBlock: 'latest'});
myEvent.watch(function(error, result){
  if(error) {
    console.log(error);
  }
  console.log(result)
});
```

Figure 8.9 – Reading all events using web3 from block 24000 until the latest block

Here, we are interested in and watching for any event from a contract. We read `fromBlock` number 25000 until the `latest` block. First, a reference to `allEvents` is made and a watcher is added to the reference. The watcher then takes a `promise` function that is executed whenever an event is fired. The value in the `result` object from the event is shown in the following screenshot:

```
{ address: '0x600c320dd768fb55f03748d4d4028db2caf06a9',
  blockNumber: 24864,
  transactionHash: '0x38ca3d4b40f8e75d27ab3950234d837e96dbdd086178f139c4e675dc6531ee15',
  transactionIndex: 0,
  blockHash: '0x778b9a9a89b4609475dcc52d4c60de44254c24f8356b4886496488f57761ca0c',
  logIndex: 0,
  removed: false,
  event: 'ageRead',
  args: { '' : '33' } }
```

Figure 8.10 – web3 output for an event

Events help us to know about the changes that have taken place within a contract and take reactive measures either to halt the further execution of a contract or execute some external process for further processing. It could also just be a means to get log information about execution.

Summary

In this chapter, we covered exception handling and events. These are important topics in Solidity, especially when writing any serious decentralized applications on the Ethereum platform. Exception handling in Solidity is implemented using three functions: `assert`, `require`, and `revert`. Although they sound similar, they have different purposes, which were explained in this chapter with the help of examples. Events help us to write scalable applications. Instead of continuously polling the platform for data and wasting resources, it's better to write events and then wait for them to execute functions asynchronously. This was also covered in this chapter. You will be able to use the skills learned in this chapter to add exception handling to your smart contracts, taking care of results arising out of unexpected environmental issues. Smart contracts allow you to notify in an asynchronous manner using logs and events when such an untoward incident happens.

In the next chapter, we will focus on using Truffle, one of the most popular development platforms for developing an application on the Ethereum platform. Stay tuned!

Questions

1. What are the keywords available in Solidity for exception handling?
2. What keyword is used for raising an event from a contract function?
3. Name two new error handling features in Solidity.

Further reading

- The Solidity docs have a good introduction to events at <https://docs.soliditylang.org/en/v0.8.9/contracts.html#events>.
- More details about the web3.js library are available at <https://web3js.readthedocs.io/en/v1.7.0/>.
- Information about another library that can help watch for events, known as ethers.js, is available at <https://web3js.readthedocs.io/en/v1.7.0/>.

9

Basics of Truffle and Unit Testing

Programming languages need a rich ecosystem of tools that eases development. Like any application, even blockchain-based decentralized applications should have a minimal **Application Life Cycle Management (ALM)** process. It is important for any application to have a process of building, testing, and deploying continuously. Solidity is a programming language and needs support from other tools to ensure that developers can develop, build, test, and deploy contracts with ease, rather than going through the painful process of deploying and testing them. This improves their productivity and eventually helps bring the application to market faster, better, and cheaper. It is also possible to introduce DevOps for smart contracts with the help of such tools. Truffle is one such development, testing, and deployment utility that can make these activities a breeze.

This chapter covers the following topics:

- Application development life cycle management
- Understanding and installing Truffle
- Contract development with Truffle
- Testing contracts with Truffle
- Interactively working with Truffle

By the end of the chapter, you'll have learned the basics of unit tests, writing unit testing using Solidity, and executing those unit tests against smart contracts. You'll also know how to execute unit tests by creating transactions and validating their results.

Technical requirements

To follow the instructions in this chapter, you will need the following tools:

- Ganache
- Truffle

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter09>.

Application development life cycle management

As mentioned before, every serious application has some development process built around it. Typically, it involves designing, building, testing, and deploying. The contract ALM is no different from any other software or programming development life cycle.

The first step in contract development is to get and finalize requirements about the problem under consideration. Requirements form the starting activity for any decentralized application. They contain descriptions of problems, use cases, and detailed testing strategies.

Architects take functional and technical requirements as their inputs and create application architecture and design. They also document them using notations easily understandable by others. The project development team takes these architecture and design documents and breaks them down into features and sprints. The development team starts working on building contracts and other artifacts based on this documentation. The contracts are frequently deployed to a test environment for testing and to ensure that they are in a working condition, both technically and functionally. The contracts are unit tested to check their functionality in isolation. If there are unit test failures, the entire build and test process should be repeated. In the end, all artifacts are deployed to the production environment.

As you can see, ALM is an involved process and can consume substantial time and productivity on the part of developers. There is a need for tools and automation to help ease this process, and this is where Truffle as a utility shines.

Introducing Truffle

Truffle is an accelerator that helps increase the speed of development, deployment, and testing, and increases developer productivity. It is built specifically for Ethereum-based contract and application development. At the time of writing this book, the latest Truffle version is 5.5.5. It is a Node.js runtime-based framework that can help implement DevOps, continuous integration, continuous delivery, and continuous deployment with ease. It provides a command-line interface through which it can be configured and managed.

Installing Truffle is quite simple. A prerequisite for installing Truffle is Node.js, as it is deployed as a Node.js package.

The latest version of Truffle can be installed by executing the following npm command from the command line:

```
$ npm install -g truffle
```

Here, npm refers to *node package manager*, and the `-g` switch signifies installation at a *global scope*. The following screenshot shows the installation of Truffle on macOS. The command is the same for Linux and Windows distributions as well:

```
added 125 packages, removed 617 packages, changed 1096 packages, and audited 1222 packages in 3m  
91 packages are looking for funding  
  run `npm fund` for details  
49 vulnerabilities (20 moderate, 29 high)  
To address all issues, run:  
  npm audit fix
```

Figure 9.1 – Installation output for Truffle

Running `truffle version` shows the current version and all commands available with Truffle, as shown in the following screenshot:

```
[(base) riteshmodi@riteshm ~ % truffle version  
Truffle v5.5.5 (core: 5.5.5)  
Ganache v^7.0.3  
Solidity v0.5.16 (solc-js)  
Node v16.2.0  
Web3.js v1.5.3
```

Figure 9.2 – Finding Truffle version details

Truffle provides a command-line interface and accepts numerous commands. Simply executing `truffle` on the console provides all available commands, as shown in the following screenshot:

```
[(base) riteshmodi@riteshm contracts % truffle
Truffle v5.5.5 - a development framework for Ethereum

Usage: truffle <command> [options]

Commands:
  build      Execute build pipeline (if configuration present)
  compile    Compile contract source files
  config     Set user-level configuration options
  console    Run a console with contract abstractions and commands available
  create     Helper to create new contracts, migrations and tests
  dashboard  Start Truffle Dashboard to sign development transactions using
             browser wallet
  db         Database interface commands
  debug      Interactively debug any transaction on the blockchain
  deploy     (alias for migrate)
  develop    Open a console with a local development blockchain
  exec       Execute a JS module within this Truffle environment
  help       List all commands or provide information about a specific command
  init       Initialize new and empty Ethereum project
  install   Install a package from the Ethereum Package Registry
  migrate   Run migrations to deploy contracts
  networks  Show addresses for deployed contracts on each network
  obtain    Fetch and cache a specified compiler
  opcode    Print the compiled opcodes for a given contract
  preserve  Save data to decentralized storage platforms like IPFS and Filecoin
  publish   Publish a package to the Ethereum Package Registry
  run       Run a third-party command
  test      Run JavaScript and Solidity tests
  unbox    Download a Truffle Box, a pre-built Truffle project
  version   Show version number and exit
  watch    Watch filesystem for changes and rebuild the project automatically

See more at http://trufflesuite.com/docs
```

Figure 9.3 – Finding all commands available with Truffle

We will not cover all the available commands with Truffle; however, we will cover the `init`, `migrate`, `console`, and `test` commands. These are some of the most important commands in Truffle and every developer uses them eventually. To get more information about each command with parameters and options available to each one of them, another command, `truffle help <<name of the command>>`, can be executed. For example, help with the `migrate` command can be found by executing the `truffle help migrate` command, as shown in the following screenshot:

```
(base) riteshmodi@riteshm contracts % truffle migrate help
Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.
Network up to date.
(base) riteshmodi@riteshm contracts % truffle help migrate
Usage:      truffle migrate [--reset] [--f <number>] [--to <number>]
            [--compile-all] [--compile-none] [--verbose-rpc] [--interactive]
            [--skip-dry-run] [--describe-json] [--dry-run] [--network <name>] [--config <file>] [--quiet]
Description: Run migrations to deploy contracts
Options:
  --reset
    Run all migrations from the beginning, instead of running from the last completed migration.
  --f <number>
    Run contracts from a specific migration. The number refers to the prefix of the migration file.
  --to <number>
    Run contracts to a specific migration. The number refers to the prefix of the migration file.
  --compile-all
    Compile all contracts instead of intelligently choosing which contracts need to be compiled.
  --compile-none
    Do not compile any contracts before migrating.
  --verbose-rpc
    Log communication between Truffle and the Ethereum client.
  --interactive
    Prompt to confirm that the user wants to proceed after the dry run.
  --dry-run
    Only perform a test or 'dry run' migration.
  --skip-dry-run
    Do not run a test or 'dry run' migration.
  --describe-json
    Adds extra verbosity to the status of an ongoing migration
  --network <name>
    Specify the network to use. Network name must exist in the configuration.
  --config <file>
    Specify configuration file to be used. The default is truffle-config.js
  --quiet
    Suppress excess logging output.
```

Figure 9.4 – Getting help information for a Truffle command (migrate)

Now that we've successfully installed Truffle and found all the available commands, along with learning how to use `help`, it's time to use Truffle for the development and testing of contracts. Having a framework like Truffle eases the process of development and testing by automating the execution on demand. The first Truffle command that developers should know about is the `init` command, which helps in creating a Truffle project and generating a scaffold folder and files. Using `init` and other commands with Truffle is the topic of the next section.

Development with Truffle

Using Truffle is quite simple. Truffle provides lots of scaffolding code and configuration by default. Developers need only to reconfigure some of the out-of-the-box configuration options and focus on writing their contracts. Let's take a look at the following steps to bootstrap a new project using Truffle:

1. The first step is to create a project folder that will hold all project- and Truffle-generated artifacts. Alternatively, a project is generated by Truffle if we provide a project name to the `init` command.

2. Navigate to the project folder and run the `init` command. The `init` command refers to the initiation and initialization of Truffle within the folder. It will generate appropriate folders, code files, configuration, and linkage within the folder, as shown in the following screenshot:

```
[(base) riteshmodi@riteshm ~ % mkdir TruffleProject
[(base) riteshmodi@riteshm ~ % cd TruffleProject
[(base) riteshmodi@riteshm TruffleProject % truffle init

Starting init...
=====
> Copying project files to /Users/riteshmodi/TruffleProject

Init successful, sweet!

Try our scaffold commands to get started:
$ truffle create contract YourContractName # scaffold a contract
$ truffle create test YourTestName          # scaffold a test

http://trufflesuite.com/docs

(base) riteshmodi@riteshm TruffleProject %
```

Figure 9.5 – The output from executing the Truffle `init` command

The preceding code results in the following generated folder structure on macOS:

```
[(base) riteshmodi@riteshm TruffleProject % tree
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
└── test
    └── truffle-config.js

3 directories, 3 files
```

Figure 9.6 – Folders and files with scaffolding code generated by Truffle `init`

Let's take a look at the folders shown in the preceding screenshot:

- The `contracts` folder contains a single file named `migrations.sol`. It contains a contract responsible for deploying custom contracts to an Ethereum network. All custom contracts should be stored within this folder.
- The `migrations` folder initially contains a single JavaScript file for executing the contract deployment process. However, more script files can be added for modular deployments. These JavaScript files should be modified to ensure that all custom contracts are visible to Truffle and Truffle can chain and link them in an appropriate order for deployment. It contains these JavaScript files prefixed with a number. The script numbering is important because they get executed in a sequential order starting from 1.

- The `test` folder is initially empty but eventually, all test scripts should be placed within this folder.
- On macOS, there is also a configuration file called `truffle-config.js`. This configuration file contains all the configuration data that impacts the execution of Truffle. It exports a JSON object consisting of all configuration information that can be consumed by Truffle to configure its runtime and environment.

A piece of important configuration information that should be provided here is the network information to which Truffle should connect and deploy contracts.

3. The following code snippet can be used to configure the network configuration. There should be an existing Geth instance running with an RPC endpoint and port enabled; Ganache can also be used instead of Geth for deploying contracts using the JSON-RPC protocol. A network configuration element should be defined to connect to an existing Ethereum network. The network is configured with a name and, similarly, multiple networks can be configured for different environments:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*" // Match any network id
    }
  }
};
```

4. Create a new contract and store it within the `contracts` folder with `first.sol` as the filename and content, as shown in the following snippet:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;
contract First {
  int public mydata;
  function GetDouble(int _data) public returns (int
    _output) {
    mydata = _data * 2;
    return mydata;
  }
}
```

5. Write another contract, as shown in the following snippet, and save it in the same folder as earlier with `second.sol` as the filename:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;
import "./first.sol";

contract Second {
    address firstAddress;
    int public data;

    constructor(address _first) {
        firstAddress = _first;
    }

    function SetData() public {
        First h = First(firstAddress);
        data = h.GetDouble(21);
    }
}
```

Eventually, the `contracts` folder looks as shown in the following screenshot:

```
(base) riteshmodi@riteshm TruffleProject % cd contracts
(base) riteshmodi@riteshm contracts % tree
.
└── Migrations.sol
    ├── first.sol
    └── second.sol

0 directories, 3 files
```

Figure 9.7 – The newly created contracts within the contracts folder

6. Modify the `migrations` folder to add another script file to it. It should be noted that each filename must be incremented by 1 to set the order of deployment of contracts. In our case, the name of the file is `2_Custom.js`. The content of this file is shown next. The first two lines of this file refer to two contracts written earlier. This file exports a function that is invoked by Truffle while deploying. The function first deploys the first contract and then, after successfully deploying the first contract, deploys the second contract, as shown in the following code block:

```
var firstContract = artifacts.require("First");
var secondContract = artifacts.require("Second");

module.exports = function(deployer) {
  deployer.deploy(firstContract).then(
    function() {
      return deployer.deploy(
        secondContract, firstContract.address);
    }
  );
}
```

7. Execute the `compile` command using the `truffle.cmd` command, as shown in the following screenshot. It might give errors and a warning. If there are any errors or warnings, they should be rectified before moving ahead:

```
[(base) riteshmodi@riteshm TruffleProject % truffle compile
Compiling your contracts...
=====
> Compiling ./contracts/second.sol
> Artifacts written to /Users/riteshmodi/TruffleProject/build/contracts
> Compiled successfully using:
  - solc: 0.8.12+commit.f00d7308.Emscripten.clang
```

Figure 9.8 – The result of executing the Truffle compile command

Important Note

It is to be noted that when executing the `truffle` command on Windows, it gives an error related to an undefined module. If this happens, you should execute `truffle.cmd`, instead of just `truffle`, with the command.

8. Now it's time to deploy the compiled contracts. Truffle provides the `migrate` command and it should be used as shown in the following screenshot. It is to be noted that before running the `migrate` command, an instance of Geth or `ganache-cli` should be running. In the case of using Geth mining, the mining process should also be running. The `truffle develop` command autoruns an instance of Ganache if it is not already running. Ganache generates the blocks by mining them automatically on a continuous basis:

```
[truffle(development)> truffle migrate

Compiling your contracts...
=====
> Compiling ./contracts/first.sol
> Compiling ./contracts/second.sol
> Artifacts written to /Users/riteshmodi/TruffleProject/build/contracts
> Compiled successfully using:
  - solc: 0.8.12+commit.f00d7308.Emscripten.clang
Network up to date.
```

Figure 9.9 – The result of executing the Truffle `migrate` command

The preceding screenshot shows that both the migration scripts were executed based on their number ordering. Now, the contracts are deployed and available for consumption. An instance of a contract can be created using its ABI definition and address. The contract address, along with the transaction hash, is available.

There are many more activities and commands available with Truffle; however, they are out of scope for this book. Next, we will move toward understanding unit testing contracts using the Truffle `test` command.

Testing with Truffle

Unit testing refers to a type of testing specific to a software unit and component in isolation. Unit tests help ensure that code in a contract is written according to functional and technical requirements. When each of the smallest components is tested under different scenarios and passes successfully, other important tests, such as integration tests, can be performed to test multiple components.

Unit testing is also known as white-box testing. This is because unit testing involves testing the execution of each code path. In the context of Solidity smart contracts, unit testing would refer to unit testing each function. The primary goal of unit testing is to test a function, applying different scenarios and ensuring that the function is executed according to expectation. This involves knowing the expected value from execution and finding the actual result. Both the expected and actual results can be compared to find out whether the unit tests are passing or failing. There can be multiple unit tests for a corresponding Contract function—each testing a code path and scenario.

As mentioned before, Truffle generates a `test` folder and all `test` files should be placed in this folder. Tests can be written in JavaScript as well as in Solidity. Since this is a book on Solidity, we will focus on writing tests using Solidity.

Tests in Solidity are written by authoring contracts and are saved as a Solidity file. The name of the contract should start with the `Test` prefix and each function within the contract should be prefixed with `test`. Please note the case sensitivity of the `Test` and `test` prefixes for both contracts as well as function names.

The following code snippet shows the code for writing tests within the contract:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/first.sol";

contract TestFirst {
    function testInitialBalanceUsingDeployedContract()
        public {
        First meta = First(DeployedAddresses.First());
        Assert.equal(meta.GetDouble(10), 20, "done");
    }
}
```

There are a few things to note in the `TestFirst` contract. Important Truffle-provided libraries, such as `Assert.sol` and `DeployedAddresses.sol`, are imported so that functions in them can be used.

There can be multiple functions within one contract but for demonstration purposes, a single unit test is written. In practice, there will be multiple tests within the same contract.

The first line in the function creates a reference to the deployed `First` contract and invokes the `GetDouble` function. The return value from this function is compared to the second parameter of the `Assert.equal` function and, if both are the same, then the test succeeds; otherwise, it fails.

The `Assert.equal` function helps compare an actual return value with the expected return value.

It is important to understand that whenever a function within a contract is invoked, it is a transaction that will eventually be written in a block and ledger. In effect, testing a function within a contract also means that you are testing transactions related to your smart contract.

Tests are executed using the `test` command, as shown in the following screenshot:

```
[(base) riteshmodi@riteshm contracts % truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling ./test/TestFirst.sol
> Artifacts written to /var/folders/f5/rp091z3151d8t_fc1xdbcvx80000gn/T/test--8903-vFP16JLRCG0x
> Compiled successfully using:
  - solc: 0.8.12+commit.f00d7308.Emscripten.clang

TestFirst
  ✓ testInitialBalanceUsingDeployedContract (126ms)

1 passing (6s)
```

Figure 9.10 – The result of executing the Truffle test command

Executing unit tests automatically and getting feedback is quite important for developers. Having a framework like Truffle eases the process of unit testing by automating the execution of the tests and providing results. These tests can be executed interactively or from CI/CD pipelines as well. Another important command that developers should hone their skills on and use frequently is `truffle console`, which is the topic of the next section.

Interactively working with Truffle

Truffle provides an interactive console using which developers can interact with the current blockchain environment and contracts. This helps to find the current values in state variables, execute methods within the contract, check the changes in state values, debug scenarios, send Ether, query for balances, and so on in the current environment. Basically, developers can do everything with deployment contracts and accounts that they normally would do on a live blockchain.

Truffle provides the `console` command to enter the console environment. It changes the environment to the network available based on the Truffle configuration. If there is only one network configured, then it becomes the default environment. Otherwise, a network name can be passed along with the command, such as `truffle console -network development`.

To exit the console environment, the `.exit` command can be executed from within the console environment.

Within the console environment, all contracts are available by their names, for example, `First`.

Contract information in JSON format can be queried by executing the `First.toJSON` command.

It can be queried whether a contract has been successfully deployed within the network using the `First.deployed()` command.

An instance of the contract can be acquired using the `deployed` function available with the contract. This function returns a promise and so it should be executed asynchronously, as shown:

```
let firstContract = await First.deployed()
```

The address of the contract can be displayed using the following command:

```
firstContract.address
```

Developers should note that the contract instance provides numerous methods and properties to query the current environment and details. The contract details are available as top-level variables within the console and all associated methods and properties can be found by just executing the contract variable name.

A contract method can be executed asynchronously using the name of the method, as shown:

```
await firstContract.GetDouble(10)
```

This will result in output as shown in *Figure 9.11*. Note that changing the state of the blockchain using a contract method results in an entry within the ledger as part of a block. The method return value is not directly returned to the caller. The caller should query the transaction receipt to eventually query the result of the original transaction. It is also important to note that `view` and `pure` functions along with `public state` variables can be queried without a transaction and such methods can return values to the caller directly:

Figure 9.11 – The transaction receipt information after executing a contract method on the Truffle console

Using a console provides an easy and interactive way to debug and find the current status of the environment, contacts, and accounts. It improves the productivity of developers multiple times and should be used frequently to fix issues before moving from development to testing environments.

Summary

This chapter introduced Truffle as a utility for easing the processes of developing, testing, and deploying Solidity contracts. Instead of developing and executing each step manually, Truffle provides an environment and easy commands for compiling, deploying, and testing contracts. You have now learned how to use Truffle to create and unit test smart contracts. You have also learned how to use Truffle for the interactive development and deployment of smart contracts to Ethereum networks.

The following chapter will be the last chapter of this book and will focus on troubleshooting activities and tools related to Solidity. Debugging is an important aspect of troubleshooting and is an important skill for any contract developer and development. Remix debugging facilities will be discussed along with other mechanisms for debugging contracts.

Questions

1. Name any four important Truffle commands used in this chapter along with their usage.
2. How do you connect a network so that Truffle can interact with it?

Further reading

You can find the documentation for both Truffle and Ganache at <https://trufflesuite.com/docs/index.html>.

10

Debugging Contracts

So far, we have looked at Solidity and Ethereum from a conceptual standpoint, developed and authored Solidity contracts, and tested them. The only thing that was not discussed was troubleshooting contracts. Troubleshooting is an important skill and exercise when dealing with any programming language. It helps in finding issues and solving them efficiently. Troubleshooting is both an art and a science. Developers should learn the art of troubleshooting through experience as well as by exploring details behind the scenes using debugging. This chapter will focus on debugging coding issues related to Solidity contracts.

This chapter covers the following topics:

- Overview of Debugging contracts
- Debugging contracts using Remix and Solidity events
- Using a block explorer

By the end of the chapter, you'll learn troubleshooting and debugging using multiple tools, such as Remix and events. This chapter will show how to execute code line by line, checking the state after every line of code and changing contract code accordingly.

So, let's begin!

Technical requirements

The following tools are required for working along with this chapter:

- A Chrome browser
- An online Remix editor

All code from the chapter are placed on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter10>.

Overview of debugging

Debugging is an important exercise when authoring Solidity smart contracts. Debugging refers to finding issues, bugs, and removing them by changing code. It is very difficult to debug a smart contract if there is inadequate support from tools and utilities. Generally, debugging involves executing each line of code step by step, finding the current state of temporary, local, and global variables, and walking through each instruction while executing contracts.

We can debug Solidity contracts using the following methods:

- Using the Remix editor
- Raising and consuming events
- Using a block explorer

Let's briefly find out more about each one.

The Remix editor

We used the Remix editor to write Solidity contracts in the previous chapters. However, we have not used the debugging utility available in Remix. The Remix Debugger helps us observe the runtime behavior of contract execution and identify issues. The Debugger works in Solidity and the resultant contract bytecode. With the Debugger, the execution can be paused to examine contract code, state variables, local variables, and stack variables, and view the EVM instructions generated by the contract code.

The following block of contract code will be used to demonstrate debugging in the Remix editor:

```
pragma solidity ^0.8.9;

contract DebuggerSampleContract {
    int counter = 10;

    function LoopCounter(int _input) public view returns(int) {
        int returnValue;
        for ( ; _input < counter ; _input++ ) {
            returnValue += _input;
        }

        return returnValue;
    }
}
```

The contract has a single state variable and function. The function loops over the provided input until it reaches the value of `counter` and returns a cumulative sum to the caller.

Deploying and executing the `LoopCounter` function will provide an opportunity to debug this function by clicking on the **Debug** button, as shown in the following screenshot:



Figure 10.1 – Deployed contract information in the Remix editor

Once the contract is deployed, its functions can be invoked from the Remix editor along with passing in any parameter value it expects. This generates another transaction and provides us with an opportunity to debug the method invocation. The contract deployment itself is a transaction and so it can also be debugged in the usual way applicable for a normal transaction. The following screenshot shows the process of executing the `LoopCounter` method of the contract:

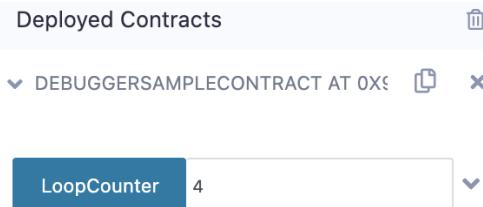


Figure 10.2 – Invoking a contract method from the Remix editor passing in parameter values

Even if there are breakpoints set in Remix within the function, invoking the `LoopCounter` method using the **LoopCounter** button shown before will not stop the execution of the function. There is another method for debugging the method invocation along with breakpoints. To do this, execute the transaction by invoking the `LoopCounter` method. This will generate a transaction hash for the transaction. This transaction hash should be copied and used for debugging purposes. The copied hash should be used in the **DEBUGGER** tab. Paste the hash in the textbox provided and click on the **Start debugging** button as follows:

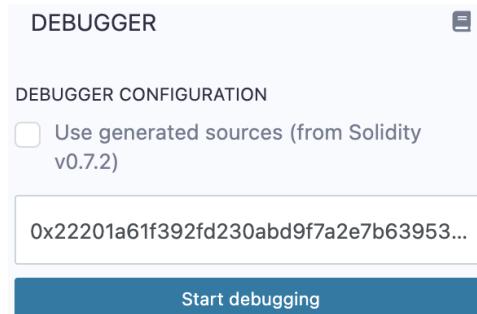


Figure 10.3 – Debugging in the Debugger tab using the transaction hash

This will start the debugging process and additional debugging information and action buttons on the UI will become visible. Here, runtime information about the local variables, state variables, memory variables, call stack, stack, instructions, and call data can be verified for the execution of each code step.

The following screenshot shows a variety of internal information about the contract runtime execution:

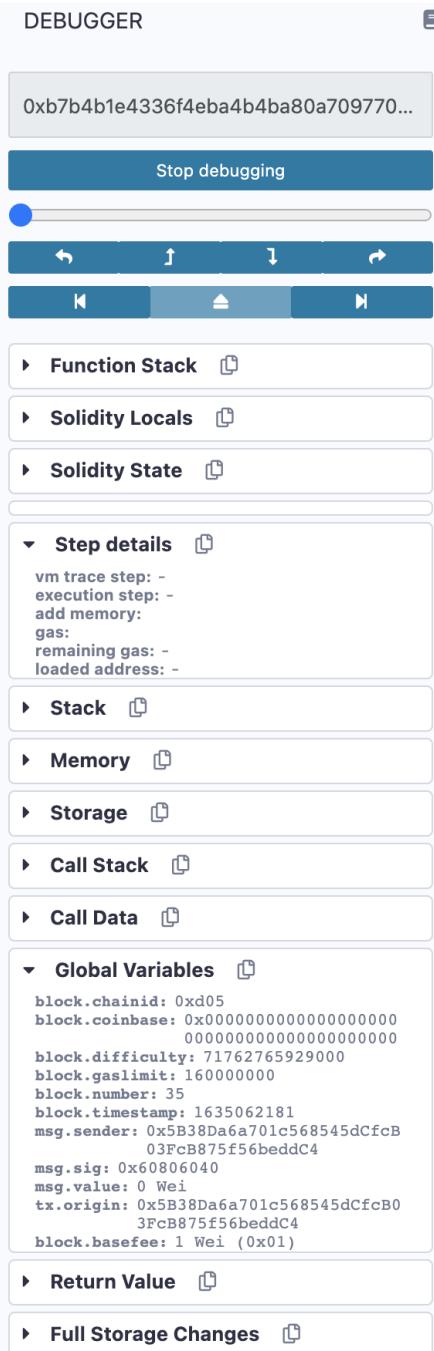


Figure 10.4 – Debug and runtime information for a transaction in the Remix editor

The preceding screenshot shows the bytecode for function execution broken down as follows:

- **Function Stack:** This shows the nested function calls for the method invocation.
- **Solidity Locals:** This section shows information about the parameter, data type, and its value.
- **Solidity State:** This section shows information about state variables, their data type, and current value. Immediately after the state details is a section that shows the bytecode and its step-by-step execution by the EVM.
- **Step details:** This is important for debugging gas usage, consumption, and remaining gas.
- **Stack:** This instruction shows the interim variables needed by the function code.
- **Memory:** This instruction shows the local variables used within the function.
- **Call Stack:** This section shows the value of the nested contracts involved within the transaction. Generally, it contains the address of the invoked contract.
- **Call Data:** This function shows the actual payload the client sends to the contract. The first four bytes refer to the function identifier and the rest contain 32 bytes for each incoming parameter.
- **Global Variables:** This section shows the value of global variables available in each transaction.
- **Return Value:** This section shows the return value in the case and when the instruction set returns a value from the function.

An important aspect of debugging is to stop execution at lines of code that we think needs attention. This could be because we think the code under consideration is causing an issue or is not logically correct. Breakpoints help us suspend the current execution by halting the execution and allow us to inspect the current context, including variable values and the function call stack. Clicking on any line next to the line number sets a breakpoint at that location. Removing a breakpoint is done by clicking on an existing breakpoint. During the execution of a function, when it hits this line, the execution is halted, and the values and execution can be verified from the **Debugger** tab. The following screenshot shows the breakpoint at line number 10:

```

3 pragma solidity ^0.4.0;
4
5 contract DebuggerSampleContract {
6
7     int counter = 10;
8
9     function LoopCounter(int _input) public view returns (int)
10    int returnValue;
11
12    for (; _input < counter; _input++)
13    {
14        returnValue += _input;
15    }
16    return returnValue;
17 }
18 }
```

Figure 10.5 – Breakpoint in the Remix editor for a function in a smart contract

Using Remix, it is also possible to perform *Step over back*, *Step back*, *Step into*, *Step over forward*, *Jump to the previous breakpoint*, *Jump out*, and *Jump to the next breakpoint* operations. It also provides the facility to view information about a particular block or transaction using a block number or transaction hash. It is possible to provide a transaction number in a block instead of a transaction hash, as shown in the following screenshot:

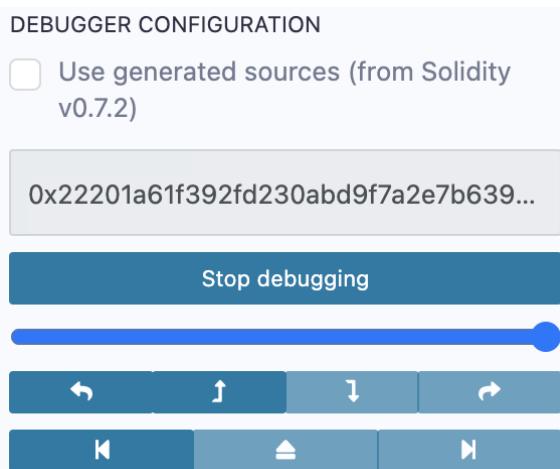


Figure 10.6 – Debugging using debug controls in the Debugger tab

Now that we know the process of debugging Solidity smart contracts using the debugging features available in the Remix editor, let's examine some other ways to perform debugging: using events and a block explorer.

Using events

We saw how to use events in *Chapter 8, Exceptions, Events, and Logging*. Information from events can be automatically read by observing the events and this information can provide data about current transactions, the functions involved, function parameters, current values in state variables, and so on. This information is available from events in real time and assists debugging, not only during development but also in production environments. Contracts should declare events, and functions should invoke these events at appropriate locations with information that provides enough context to whoever is reading these events.

Using a block explorer

A block explorer is an Ethereum browser. They provide reports and information about current blocks and transactions on the Ethereum network. They are a great place to learn more about existing and past data. One example is available at <https://etherscan.io/>, as shown in the following screenshot. Block explorer can help in debugging by making information about blocks, transactions within a block, and other metadata easily accessible to developers. They help in finding the transaction hash, block hash, the sender and recipients involved in a transaction, gas supplied, gas consumed and many other details besides.

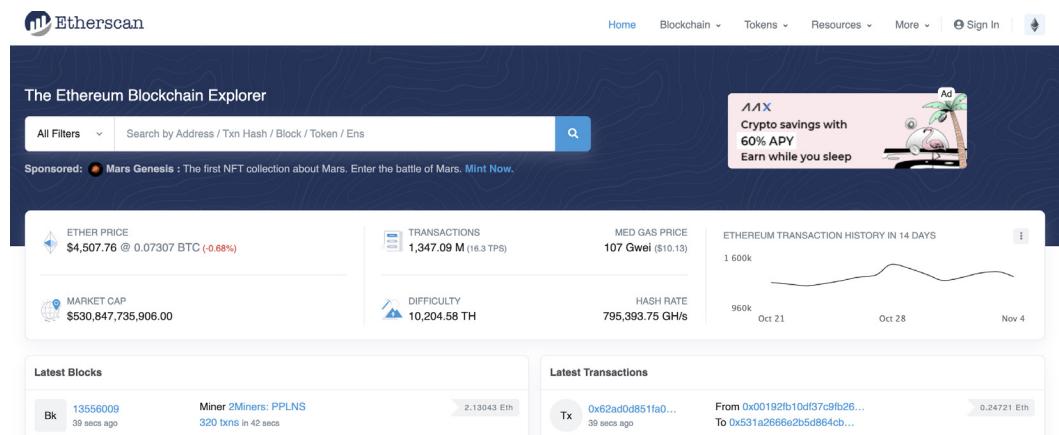


Figure 10.7 – The etherscan.io website provides information about blocks and transactions

It shows transactions involving both accounts and contracts. Clicking on a transaction reveals more details about it, as shown in the following screenshot:

Transaction Details

Sponsored: Coinfluence - Join Bow Wow & other celebrities in the Coinfluence ICO [CFLU Tokens](#)

Overview	Access List	State	Comments
② Transaction Hash:	0x62ad0d851fa02d3d906783479c7cde0d2ef56051d0c2dbb800875f0dff4ec61d		
② Status:	Success		
② Block:	13556009 2 Block Confirmations		
② Timestamp:	① 1 min ago (Nov-05-2021 10:15:50 AM +UTC) ① Confirmed within 30 secs		
② From:	0x00192fb10df37c9fb26829eb2cc623cd1bf599e8 (2Miners: PPLNS)		
② To:	0x531a2666e2b5d864cb69d7906642cf191e3100f0		
② Value:	0.247211515 Ether (\$1,114.16)		
② Transaction Fee:	0.002034107193447 Ether (\$9.17)		
② Gas Price:	0.000000096862247307 Ether (96.862247307 Gwei)		
② Txn Type:	2 (EIP-1559)		
Click to see More			
② Private Note:	To access the Private Note feature, you must be Logged In		

Figure 10.8 – Transaction information and metadata on etherscan.io

At this point, you have an understanding of the details of transactions stored within the Ethereum ledger. Let's take a look at the following few details of the transaction in the preceding screenshot:

- **Transaction Hash:** It has already been explained in *Chapter 1, Introduction to Blockchain, Ethereum, and Smart Contracts*, that every transaction is hashed before storing them in a block.
- **Status:** This detail represents the status of a transaction, whether it is successful or pending.
- **Block:** This detail shows which block number the transaction is stored in.
- **TimeStamp:** This detail shows the timestamp for the transaction.
- **From:** This detail shows who sent the transaction.

- **To:** This detail shows the recipient of the transaction.
- **Value:** This detail shows the amount of Ether transferred.
- **Transaction Fee:** This detail represents the gas limit specified by the user.
- **Gas Price:** This detail shows the gas price determined by the sender.
- **Txn Type:** This attribute was introduced in EIP 2718 and shows the type of transaction.

Clicking on a block shows information about the block and a list of transactions that are part of that block. It shows all the details from the block header, such as the block hash, parent hash, miner account, difficulty level, nonce, and more, as shown in the following screenshot:

Block #13556009

Featured: Review and revoke dApp access to your tokens with our [Token Approvals tool!](#)

Overview	Comments
⑦ Block Height:	13556009
⑦ Timestamp:	⌚ 1 min ago (Nov-05-2021 10:15:50 AM +UTC)
⑦ Transactions:	320 transactions and 95 contract internal transactions in this block
⑦ Mined by:	0x00192fb10df37c9fb26829eb2cc623cd1bf599e8 (2Miners: PPLNS) in 42 secs
⑦ Block Reward:	2.130434497208140237 Ether (2 + 2.750581798609180484 - 2.620147301401040247)
⑦ Uncles Reward:	0
⑦ Difficulty:	10,320,670,491,748,494
⑦ Total Difficulty:	33,789,123,013,234,863,807,981
⑦ Size:	191,791 bytes
⑦ Gas Used:	27,332,421 (91.11%) +82% Gas Target
⑦ Gas Limit:	30,000,000
⑦ Base Fee Per Gas:	0.000000095862247307 Ether (95.862247307 Gwei)
⑦ Burnt Fees:	🔥 2.620147301401040247 Ether
⑦ Extra Data:	EthereumPPLNS/2miners_EU3 (Hex:0x457468657265756d50504c4e532f326d696e6572735f455533)
Click to see more ↓	

Figure 10.9 – Block information and metadata on etherscan.io

The block header has some interesting properties, such as the **Height** detail, which provides the block number in the ledger, the number of transactions within the block (320 in this case), and the number of internal transactions (these are referred to as message calls between contracts). We can also see the hash of the current block header (**Hash**), the hash of the parent block (**Parent Hash**), the hash of the root for the Coinbase or Etherbase account that mined the block (**Mined By**), the difficulty level for the current block, the cumulative difficulty for all blocks up to the current block, the size of the block, the total gas used by all transactions within the block, the maximum limit of gas for the block, the evidence that proof of work has been carried out (**Nonce**), and the reward for mining the block.

Summary

Solidity is a new programming language that is continuously evolving. Solidity contracts can be debugged using the Remix editor. Remix provides a convenient way to author and debug contracts by verifying variables and code execution at every step. It helps us move forward and backward in code execution. It provides breakpoints to break the execution of code. There are other ways to debug contracts as well. These include using block explorers and Solidity events. Although events and block explorers provide limited capabilities for debugging, they are very helpful and facilitate production.

The next chapter will get into low-level programming in Solidity using assembly language. This will allow greater control over gas usage and access to a few features not available with the Solidity language.

Questions

1. What are the different ways to debug smart contracts using the Remix editor?
2. What different contextual information is available in the debug tab in Remix?

Further reading

To get detailed information about debugging using the Remix editor, check out this link:
https://remix-ide.readthedocs.io/en/latest/tutorial_debug.html

Part 3: Advanced Smart Contracts

In this section, we will explore advanced concepts and techniques related to smart contracts, including Solidity design patterns, upgradable smart contracts, and writing secure contracts. This section also shows ways to use assembly programming with Solidity and create ERC20 and ERC721 tokens. A complete implementation of both fungible and non-fungible tokens is part of this section.

This part contains the following chapters:

- *Chapter 11, Assembly Programming*
- *Chapter 12, Upgradable Smart Contracts*
- *Chapter 13, Writing Secure Contracts*
- *Chapter 14, Writing Token Contracts*
- *Chapter 15, Solidity Design Patterns*

11

Assembly Programming

Solidity is a high-level language. A Solidity compiler does not generate assembly code; instead, it generates **Intermediate Code (IL)**. This IL code is understood by the **Ethereum Virtual Machine (EVM)**.

The EVM, on the other hand, does not understand Solidity constructs. It loads the IL code and executes it by interpreting it. This IL code is known as **bytecode** within the Ethereum ecosystem.

All activities, including contract deployment, contract function invocation, and the simple transfer of Ether between accounts, lead to bytecode execution by the EVM. Bytecode comprises instructions in hexadecimal format. The instructions or bytecode built by **Solidity compiler (solc)** comprise a series of opcodes along with their input values in hexadecimal format.

Opcodes are human-readable; however, they have equivalent hexadecimal representation, which is what we see as bytecode. It's easier to think of opcodes as functions that accept arguments and return values. Assembly code in Solidity eventually generates a sequence of opcodes that are executed by the EVM as bytecode.

In this chapter, we will cover the following topics:

- An introduction to Solidity and its advantages
- Getting started with Assembly programming in Solidity
- Scopes and blocks
- Returning values from assembly blocks
- Using memory slots from assembly
- Using storage slots from assembly
- Calling contract functions
- Determining contract or externally owned account addresses

Technical requirements

The following tools are required for working along with this chapter:

- A Chrome browser
- An online Remix editor

The code from this chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter11>.

An introduction to Solidity and its advantages

Assembly programming in Solidity is low-level programming in which we can use opcodes and work with them directly. There are many advantages to using assembly language while writing contracts. The major advantages include the following:

- **Added capability:** There are a few things that can only be done using assembly programming in Solidity. Some of these capabilities are not available in Solidity grammar itself – for example, determining whether an address is a contract address can be ascertained in assembly but not in Solidity. It should be noted that Solidity developers are trying to minimize this gap in the newer versions of the language.
- **Optimization of gas usage:** We can optimize code by writing assembly code because it has fewer instructions compared to Solidity compiler-generated code.
- **Having full control:** Writing assembly language directly gives us more control over generated bytecode compared to the compiler-generated boilerplate code.

Before getting our hands dirty with assembly programming, it is important to understand that the EVM is a stack-based virtual machine. This means that all instructions are pushed on the stack, and they are popped one after another in the **Last-In-First-Out (LIFO)** order. The opcode instructions are also pushed on the stack; the virtual machine pops and executes the opcodes from the top and keeps doing it till there is nothing to execute.

There are two different ways or styles to write assembly code in Solidity – functional and non-functional. They both eventually generate the same bytecode; however, a developer may be comfortable using one over the other. This book focuses on the functional style of assembly coding because it is more intuitive to write and understand. It uses functions and inner functions to push the opcodes onto the stack along with their arguments.

A typical example of functional assembly coding is shown next. Here, the `mload` opcode loads the value stored at the `0x80` memory location, adds 3 to it, and finally, stores it back at the `0x80` memory location using the `mstore` function:

```
mstore(0x80, add(mload(0x80), 3))
```

Note that in the previous code, the flow of execution happens from right to left and is very intuitive, which allows us to understand the intent of the code. The same code written in a non-functional style is shown next:

```
3 0x80 mload add 0x80 mstore
```

All the smart contracts written so far in the book generate the standalone assembly code by the Solidity compiler. However, it is also possible to mix both Solidity statements and Assembly code in the same smart contract. This is also known as **inline assembly**.

Now, we are ready to take our first step into assembly programming. The next section introduces a basic example of Assembly programming using Solidity.

Getting started with Assembly programming

Assembly code can be mixed with Solidity code using the `assembly` keyword, followed by brackets defining a block. There can be as many assembly blocks as needed within a Solidity function. The `SimpleAssembly` contract shown in the following code block shows the usage of Solidity code intertwined with multiple assembly blocks:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract SimpleAssembly {
```

```
function AssemblyUsage() public pure returns (uint256) {  
  
    uint256 i = 10;  
  
    assembly {  
        i := 100  
    }  
  
    assembly {  
        i := 200  
    }  
  
    return i;  
}  
}
```

In the previous code, there are two assembly blocks, with each block assigning a constant value to the `i` function variable. Executing this function will result in 200 being returned as a value.

There are a few things to note about the assembly code:

- There are no semicolons to terminate a statement. The newline acts as a statement terminator.
- Assignment of a value to a variable includes a colon followed by the equals operator (`:=`).
- Assembly blocks can access variables defined outside the assembly block using their names.
- Assembly blocks can read and write values to variables defined in the parent scope. The first assembly block updates the value of the `i` variable to 100, and the last block updates it to 200.

Assembly code allows us to declare new variables using the `let` keyword. `let` initializes the variable and can optionally assign value to it. However, there are no data types in assembly while declaring variables. In assembly, the variables get 32 bytes allocated as memory size.

The next code listing creates multiple variables, with the first three getting assigned literal values of different types. `stringVal` gets a string value, `uintVal` gets an integer value, and `byteVal` gets a byte value in hexadecimal format. Another variable, `newvariable`, is declared in assembly code, and it gets assigned dynamically a generated value from the `add` function. The value is then assigned to the `function` variable. The value returned from the function is 40. The scope of the variables defined within the assembly block is limited to the block itself. The variables go out of scope as soon as the execution leaves the block. To persist the value (not the variable), it should be returned from the assembly block (which will be shown later in the *Returning values* section):

```
contract AssemblyVariablesAndSimpleFunctions {

    function AssemblyUsage() public pure returns (uint256) {
        uint256 i;
        assembly {
            // this is a comment
            /*
                this is a
                multiline comment
            */
            let stringVal := "ritesh modi"
            let uintVal := 100
            let byteVal := 0x100

            let newvariable := add(10,30)
            i := newvariable
        }
        return i;
    }
}
```

Similar to the `add` function, there are multiple other functions defined by Solidity. Some of these include `sub` for subtraction, `mul` for multiplication, `div` for division, `mod` for modulo, `exp` for exponential, `lt` for less than, `gt` for greater than, and `eq` for equals. The complete list is available at <https://docs.soliditylang.org/en/latest/assembly.html#opcodes>.

The assembly block also supports both the `//` and `/* */` operators for comments. `//` is used for single-line comments and `/* */` is used for multi-line comments, as shown in the previous code block.

Another building block of assembly programming is the scope and block constructs that define the visibility of variables within code. The next section discusses the concept of scopes and blocks in the context of assembly programming.

Scopes and blocks

Assembly blocks can, in turn, have nested blocks. A **block** is a scope, and any variables declared within a scope get deallocated as soon as the execution leaves the block. Each block defines a local scope.

Variables declared within nested blocks are not visible outside of the block. They get deallocated after the block execution completes. The `innerValue` variable declared within the inner block cannot be used outside the block. It is visible only inside the block in which it is declared. The same goes for the `outerValue` variable, which is not accessible outside the assembly block but is available for reading inside its nested blocks.

Moreover, variables declared in the parent scope cannot be redefined in the inner scope or block. The `outerValue` is already declared in the parent scope, and trying to redefine it using the `let` keyword within an inner block raises a `variable name is already taken` compile-time error.

Also, variables declared in one scope cannot be accessed from another sibling scope. These concepts are all displayed in the following code block:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract AssemblyScopes {
    function getValues() public pure returns (uint256
        retval) {
        assembly {
            let outerValue := 10
            {
                let innerValue := 20
                {
                    innerValue := 30
                }
            }
        }
    }
}
```

```

        }
        {
        // Variable name already taken in this scope
        //let outerValue := 40

        //cannot use innerValue as it is not visible
        //innerValue := 50
        }
        // retval := innerValue
        retval := outerValue
    }
}

}

```

Blocks and scope are important to understand the availability of a variable in a block, and returning values from block and functions are equally important. Next, let's get into understanding different ways of returning values from assembly blocks and functions.

Returning values

Assembly blocks can return values to their parent scope. Since variables get deallocated after the block execution, returning values from an assembly block can become quite important. There are a couple of ways to return values from the assembly block.

One way to return values from a function that, in turn, gets a value from assembly code is to use a variable name in the function signature. The same variable should be assigned a value from the assembly code. The next function shown declares a return type using `retval` as its name. In assembly code, the `retval` variable is assigned a constant value, and this gets automatically returned as part of the function's return value. Note that the `return` keyword or opcode is not used in this case:

```

function usingReturnVariable() public pure returns (uint256
    retval) {
    assembly {
        let temp := 10
        retval := temp
    }
}

```

Solidity also provides an opcode for returning values. Instead of using the `return` variable name, the `return` opcode can be used to return values from assembly code as well as from a function. It accepts two arguments – the starting memory location and the length in bytes for returning the value. The assembly code initializes and declares a `temp` variable with a constant integer value. It stores the value as part of memory storage using the `mstore` opcode. Don't worry about this opcode; it is covered in detail in the next section. It is enough to know that this opcode stores a value at a given memory location. The assembly code then uses the `return` opcode that accepts the memory location and the length to read for returning the value. In this case, we are returning 32 bytes as data stored at the `0x0` location:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ReturningValues {

    function usingReturnOpcode() public pure returns
        (uint256) {
        assembly {
            let temp := 10
            mstore(0x0, temp)
            return(0x0, 32)
        }
    }
}
```

Returning values from assembly code is an important concept to understand, as it is used extensively when using memory and storage data. The next section will show the usage of memory within assembly code.

Working with memory slots

Solidity provides opcodes for working with memory variables in assembly code. We already know that there are three types of variables – `storage`, `memory`, and `calldata`. Assembly language can work with all three.

Assembly provides the `mload` and `mstore` opcodes for loading or storing values in memory variables. The `mload` opcode helps to load or read the memory location and the returning data stored therein. The `mstore` opcode helps to store data in memory. The usage of both `mload` and `mstore` is shown using a smart contract in the following code block:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MemoryAssembly {

    function AssemblyUsage() public pure returns (uint256)
    {
        assembly {
            let addresult := add(100,200)

            mstore(0x0, addresult)

            let y := mload(0x0)

            mstore(add(0x0, 0x20), add(y,200))

            //return(0x0, 32)
            return(0x20, 32)
        }
    }
}
```

There is quite a lot going on in this seemingly simple code. The function only has assembly code, and it starts by declaring a new variable, `addresult`, with a value assigned using an `add` function. Constant values are supplied to the `add` function. The value in the `addresult` variable is stored in memory available to the function using the `mstore` opcode. `mstore` accepts two arguments – the location to write the value and the actual value. In this case, the value is stored at the `0` memory location.

Every function invocation is allocated memory slots (each slot is 32 bytes in size) for the storage of function variables and their inner working. Each memory storage is completely blank when a function starts its execution. The first four slots are reserved for the function's inner working, and we will see later how to not use the reserved memory slots. In the previous code block, we used the first slot to demonstrate the usage of memory slots; however, this should be practiced in real scenarios.

Immediately after storing the value in the 0 memory slot, the value is read from it using the `mload` opcode and stored in a `y` variable. `mload` accepts the memory location to read as its argument. In the next step, a constant value of 200 is added to the `y` variable, and it is stored in the next memory slot by adding 32 bytes to the `0x0` memory location. The `return` opcode returns 32 bytes, starting from the `0x20` memory to the caller.

The preceding function can be rewritten to use the `0x40` memory location instead of the reserved location. `0x40` contains information about the next available free memory, and using it for reading and writing custom memory-associated variables is always safe. This is shown in the following code block. The only difference from previous code is that the values are stored in memory, starting with the `0x40` location:

```
function AssemblyUsage() public pure returns (uint256) {
    assembly {
        let addresult := add(100,200)

        mstore(0x40, addresult)

        let y := mload(0x40)

        mstore(add(0x40, 0x20), add(y,200))

        //return(0x0, 32)
        //return(0x60, 32)
        return(0x0, 32)
    }
}
```

Storage slots are a permanent form of storage in Solidity, and assembly programming can be used to manage them. In fact, they are used extensively while writing upgradable smart contracts. Next, we will look into working with storage slots using assembly programming.

Working with storage slots

Similar to memory variables, Solidity also provides opcodes for working with state variables. The state variables' related opcodes are `sload` and `sstore`. Again, similar to memory functions, `sload` reads the value stored in the storage slot and returns the value. It accepts the storage slot location as its only argument. `sstore` updates the value at a given storage slot. It accepts the storage slot as its first argument and the value to be stored as its second argument. The storage slot is created if it already does not exist. The usage of both `sload` and `sstore` is shown using a smart contract in the following code block:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract StorageAssembly {
    uint256 StateVariable;

    function AssemblyUsage() public returns (uint256 newstatevariable, uint256 newderivedvariable) {
        assembly {
            sstore(0x0, 100)
            newstatevariable := sload(0x0)
            sstore(0x1, 200)
            newderivedvariable := sload(0x1)
        }
    }

    function GetStateVariable() public view returns (uint256) {
        return StateVariable;
    }

    function GetNewDerivedVariable() public view returns (uint256 slot1) {
        assembly {
            slot1 := sload(0x1)
        }
    }
}
```

```
function UpdateStateVariable(uint256 intValue) public
    returns (uint newValue) {
    assembly{
        newValue := add(intValue, sload(0))
        sstore(0, newValue)
    }
}
```

Unlike memory slots, storage slots do not need mathematical calculations for finding the next free memory location. Instead, each storage slot is accessed sequentially using a number. The first storage slot is accessed as the 0 slot, the next one as the 1 slot, and so on.

The `StorageAssembly` contract has four functions. The main function using assembly code is `AssemblyUsage`, whereas the other functions are helper functions for reading the current value stored in the state variable before and after assembly code has been executed. The `AssemblyUsage` function starts by storing the constant value 100 on the first storage slot using the `sstore` opcode. Next, it loads the value stored at `0x0` using the `sload` opcode and stores it in a state variable (which is already declared and is positioned at the 0 slot). Later, the code stores another constant value, 200, in the 1 storage slot using the `sstore` opcode. Note that this slot was not defined as a part of the state variables. Therefore, a new state variable is initialized and assigned the value 200. Finally, the 1 slot is read using the `sload` opcode into `newderivedvariable`. Both `newstatevariable` and `newderivedvariable` are returned from the function. Executing this function will show tuple values (100 and 200) being returned.

We can validate whether the `statevariable` value underwent a change to 100 by calling the `GetStateVariable` function. It returns 100 from the function. Interestingly, we can also validate whether an assigned value exists at the 1 slot by executing the `GetNewDerivedVariable` function. This function uses assembly to load the value in the 1 storage slot and returns. It indeed shows 200 as a value. This variable cannot be used apart from assembly because no name has been assigned to it.

The last function, `UpdateStateVariable`, updates the value in the 1 slot by loading it and adding the value supplied as an argument to this function. This function returns the updated value. Working with state and memory variables in assembly using direct memory helps in optimizing the code for performance. Another important feature available in assembly code is to call functions in existing contracts and to determine whether an address is a contract address. This is covered in the next section.

Calling contract functions

Assembly code also allows interaction with other contracts. One of the main use cases is to invoke functions on other contracts and get return value from them. Solidity provides the `call` opcode that can call functions in a contract. The signature of the `call` opcode is shown here:

```
call(g, a, v, in, insize, out, outsize)
```

Here, `g` stands for the amount of gas being sent with the call, `a` stands for the address of the target contract, `v` stands for the amount of Ether being sent in wei denomination, `in` stands for the starting memory location containing the data to be sent to the EVM (which comprises the method signature and its parameter values), `insize` is the size of data being sent in the hexadecimal format, `out` stands for the starting memory location that should store the return data from the call, and `outsize` is the size of return data in hexadecimal format.

We will use the `call` function to invoke a function in another contract, `TargetContract`. This is a simple contract used in the next code block. It consists of a single function, `GetAddition`, and accepts three integer arguments. The functions add the three arguments and return the answer to the caller:

```
contract TargetContract {  
    function GetAddition(uint256 firstVal, uint256  
        secondVal,uint256 thirdVal) public pure  
    returns (uint256){  
        return firstVal + secondVal + thirdVal;  
    }  
}
```

Next, we need a contract that will invoke the `GetAddition` function in `TargetContract` from assembly code using the `call` opcode. This contract is `CallingContract`, and it creates an instance of `TargetContract` using its constructor and stores it in the state variable.

The `InvokeTarget` function contains the solidity code. It first generates the hash and trims it to the first four bytes from the function signature. Note that the function signature consists of the name of the function along with its parameter data types. It then fills the memory slots in a contiguous fashion using the free pointer available at the `0x40` memory location. The first four bytes are the function signature, the next 32 bytes are used to store the `firstVal` value, and so on for all the variables. In total, the size of memory to be sent is $(4 + 32 + 32 + 32 =) 100$ bytes, which is equivalent to `064` in hexadecimal. The target function should store the return value at the `0x40` memory location, and the length of the return value is 32 bytes – that is, `0x20` in hexadecimal format.

It then calls the `call` opcode, passing in 100,000 wei as gas, the address of the `targetContract`, no Ether, the address of the memory location containing the function signature and its parameter values, the size of data being sent, the address of the memory location for the target function to store the return value, and the size of the return value. Finally, the address that stores the return value is loaded using the `mload` opcode and returned from the function:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract CallingContract {
    TargetContract targetContract;

    constructor() {
        targetContract = new TargetContract();
    }

    function InvokeTarget(uint256 firstVal, uint256
        secondVal, uint256 thirdVal) public returns (uint256
        retval) {
        address addr = address(targetContract);
        bytes4 functionSignature = bytes4(keccak256
            ("GetAddition(uint256,uint256,uint256)"));

        assembly {
            // Solidity code for generating the function signature
            // and filling memory slots
            // ...
            // Call to targetContract
            // ...
            // Load return value
            // ...
        }
    }
}
```

```
let freePointer := mload(0x40)
    mstore(freePointer,functionSignature)
    mstore(add(freePointer,0x04),firstVal)
    mstore(add(freePointer,0x24),secondVal)
    mstore(add(freePointer,0x44),thirdVal)

    let success := call(
        100000,
        addr,
        0,
        freePointer,
        0x64,
        freePointer,
        0x20)

    retval := mload(freePointer)

}

}

}
```

Another important feature that is only available in assembly programming is to determine whether an address is a contract address or belongs to an externally owned account. The following section explains the usage of assembly to determine the address type.

Determining contract addresses

Now that we understand the process of calling a contract function using Solidity assembly programming, another important coding requirement is to find out whether a given address belongs to an externally owned account or a contract account. This is especially useful when writing ERC20 and ERC721 token contracts.

To demonstrate how to determine whether an address is a contract address or an externally owned account, the same TargetContract contract shown in the previous section will be used. The TargetContract contract should be available on the Ethereum network and can be accessed using its address. The address of the TargetContract address can be supplied as an argument to the CheckIfContract function shown next. This function uses the assembly `extcodesize` opcode to determine the type of address supplied to it. It is a contract account if the length of the value returned by `extcodesize` opcode is greater than zero; otherwise, it belongs to a user account:

```
function CheckIfContract(address contractAddress) public
view returns (bool) {
    uint length;
    assembly {
        length := extcodesize(contractAddress)
    }
    if (length > 0) {
        return true;
    }

    return false;
}
```

The `CheckIfContract` function accepts an address as an argument and returns whether the supplied address is a contract address or belongs to an externally owned account. It uses the `extcodesize` assembly opcode and passes the address as its argument. This opcode returns the length as its output. If the length is greater than 0, then it is a contract address; otherwise, it belongs to an externally owned account. The same can be seen in the preceding code listing.

With this, we come to the end of learning about assembly programming in Solidity. Solidity itself is maturing with newer features; however, a few things can only be done in assembly code, and having the skills to write it in Solidity is essential for an advanced developer.

Summary

This chapter was relatively more complex compared to previous chapters. It dealt primarily with assembly programming within Solidity. Assembly is a comprehensive language, and all of its aspects cannot be covered in a single chapter.

This chapter covered some important assembly programming-related concepts, such as declaring variables, blocks, scope, and returning values from assembly blocks and from a function itself. Some advanced concepts, such as working with memory and storage data, were also covered in the chapter.

Some concepts, such as looping and conditional statements, were not covered within this chapter. If you are interested in learning about them, read the official Solidity documentation for more information. Next, it's time to learn how to write maintainable and modular smart contracts in Solidity using upgradable contracts, which we will discuss at length in the next chapter.

Questions

1. What are the opcodes associated with reading and updating storage slots?
2. What is the opcode used to ascertain whether an address is a contract address or belongs to an externally owned account?
3. At which location is the memory-free pointer available?

Further reading

- All opcodes in solidity are available in the Solidity documentation: <https://docs.soliditylang.org/en/v0.5.3/assembly.html#opcodes>.
- Solidity documentation also provides details about memory layouts and other internal details: <https://docs.soliditylang.org/en/v0.5.3/miscellaneous.html>.
- Solidity documentation related to assembly programming: <https://docs.soliditylang.org/en/v0.5.3/assembly.html>.

12

Upgradable Smart Contracts

Writing upgradable contracts is an essential design pattern for contracts supported by blockchain. It is necessary to be familiar with and implement this pattern in relation to any contracts that developers think may need to be changed in the future. In this chapter, we will look at various ways to write upgradable smart contracts. We will learn what proxy contracts are, how to implement them for function upgradability using object-oriented concepts such as inheritance and composition, and finally learn about the patterns related to upgradable storage within smart contracts. Related concepts, such as dependency injection, will also be covered as part of the chapter.

We will cover the following topics:

- Learning what constitutes upgradability
- Understanding dependency injection
- Reviewing problematic smart contracts
- Implementing simple solutions with inheritance
- Implementing simple solutions with composition
- Implementing advanced solutions using proxy contracts
- Writing upgradable contracts with upgradable storage

By the end of this chapter, you will have the skills to write upgradable smart contracts using a variety of mechanisms. These mechanisms include upgrading the functionality of the smart contracts as well as storage requirements. Both contract functionality and storage can be upgraded using patterns described in this chapter.

Technical requirements

To follow the instructions in this chapter, you will need the following:

- A Chrome or Firefox browser
- Access to the Remix website

You can find the code files for this chapter on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter12>.

Learning what constitutes upgradability

We all download and use various software. We also consume software as a service that is hosted by a provider. In any case, as a developer, we bind ourselves to a particular version of the software/service. For example, when you download Geth to run an Ethereum node, you download a specific version of Geth and use the features provided by that version. There could be multiple versions available prior to the downloaded version and they might have many similar, or, at the same time, not so similar, features. What this means is that every time there is a new version of any software/service, there are generally significant changes and improvements over the prior version.

These significant changes and improvements could be in the form of the introduction of new features, existing feature enhancement, bug fixes, issue resolution, or a variety of other reasons typically related to software, such as base platform upgrades and fixes.

We know software keeps evolving and releasing multiple versions of it is a fact of life. Smart contracts are also pieces of code and executable programs on the Ethereum VM. It means it should be quite natural for them to evolve into newer versions over a period of time either due to enhancements or fixes.

Let's also understand the meaning of "contracts" before we discuss upgradability further in the context of smart contracts.

A contract is a written/verbal agreement between multiple parties. The agreement lays down the rules and regulations along with the duties and responsibilities of each party, which they should adhere to. Any breach of such rules and duties generally levies penalties on the party in breach of the rules of the contract. These penalties are also a part of the rules specified within the agreement. A contract, once signed by each party, cannot be nullified, terminated, or modified unless each party agrees to it, and such clauses are also generally mentioned within the agreement. It means a contract cannot be modified at the will of a single or a subset of parties signed to it. This is the nature of a contract and the same nature has cascaded to smart contracts as well.

So, here comes the surprise. Smart contracts, once deployed, cannot be modified. Replacing smart contracts with updated code is not allowed. This means a smart contract cannot be upgraded with newer features and bug fixes after it has been deployed.

When a smart contract is deployed, an address for the smart contract is generated. In many ways, the address is similar to an address associated with an individually owned account. It can hold ethers and can also have a state associated with it. The address generated is used by the client to invoke functions within the contract.

Even if we do not change any code within the smart contract and deploy the same contract again, a new address of 20 bytes is generated. The smart contract deployed will be a new instance and will have no relation whatsoever to any other addresses that were generated from the same code. This means that every time we deploy a smart contract, a new instance and address are created that bear no resemblance to other deployments of the same smart contract.

Let's understand this concept of smart contracts with the help of an example. Assume that we have developed a smart contract and deployed it on the Ethereum blockchain. Once it is deployed, it starts getting consumed by its client. The smart contract will optionally store an ether balance and store the contract state along with it. Now, after a few months, you find a bug in the smart contract and want to fix the bug. When you fix the bug, you also want to see the bug fix on the Ethereum blockchain and would like to redeploy the smart contract on it. However, if you deploy the updated smart contract, it will generate a new address with a new clean state and ether balance. So, now there are two instances of the same smart contract, one holding the state and ether balance, while the new one does not have any state and balance. Moreover, you want your clients to use the new instance of the smart contract rather than the old one.

This is the nature of smart contracts and cannot be modified. In the beginning, developers were not even aware of this concept and they deployed their smart contract only to realize later that they can no longer use it because of issues in them. They had to resort to workarounds by migrating the state and ether balance from one instance to another instance of the same smart contract. This approach comes with its own set of issues and it also tends to lose certain metadata such as the date and time.

This chapter is about understanding and implementing patterns to develop smart contracts that can be upgraded in the future using the existing state from the old instance of the smart contract.

This is a good time to understand a related pattern called dependency injection, which is used heavily to write upgradable smart contracts.

Understanding dependency injection

When we have dependencies between smart contracts, the dependent smart contract would need an instance of the independent smart contract to invoke its function. There are two choices in such cases: either the smart contract itself creates a new instance of the contract, or it might already expect the smart contract instance to be available. If the dependent smart contract creates a new instance, it will have an instance address directly available using the new keyword, and if it expects the instance to be already available, then the address of the contract instance must be supplied to the dependent smart contract.

Supplying the address of the independent smart contract instance is also known as injection or, in other words, the dependencies of a smart contract are injected at runtime rather than the dependencies getting created at design time. Using the new keyword is a design-time static creation of a smart contract.

The dependencies or independent smart contract instance addresses can be provided to a smart contract in two ways as listed. It should be noted that these are not mutually exclusive and a smart contract will typically implement both the ways to implement dependency injection.

- During deployment time
- After deployment

Let's discuss each method in detail.

Providing instance addresses during contract deployment

During the deployment of a smart contract, the smart contract constructor can accept parameters related to the independent smart contract instance addresses. Whoever is deploying the smart contract must supply the values of all independent smart contract instance addresses. This is also known as constructor-based dependency injection.

This is shown using the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract IndependentContract {
    mapping (address => uint256) balances;

    constructor (uint256 amount) public {
        balances[msg.sender] = amount;
    }
}

contract DependentSmartContract {
    IndependentContract indeContract;

    constructor(address indc) public {
        indeContract = IndependentContract(indc);
    }
}
```

One of the issues that may arise is the need to update the contract address of `IndependentContract` within `DependentSmartContract` following deployment. This can be resolved by adding another function that accepts the contract address and updates the existing contract address with the newly supplied contract address.

Providing instance addresses following contract deployment

The possibility may arise that the address of the smart contract supplied to the constructor during deployment needs to be updated. The smart contract can implement specific functions whose job is to accept the address of the independent smart contract instances and update its references accordingly.

This is shown using the following example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0

contract IndependentContract {
    mapping (address => uint256) balances;

    constructor (uint256 amount) public {
        balances[msg.sender] = amount;
    }

}

contract DependentSmartContract {
    IndependentContract indeContract;

    constructor(address bankc) public {
        indeContract = IndependentContract(bankc);
    }

    function changeContractAddress(address _address) public {
        indeContract = IndependentContract(_address);
    }
}
```

With this, we can conclude the identification and implementation of the different ways in which a smart contract can consume another smart contract and supply the address of the independent contract to the dependent contract using a constructor and properties.

Reviewing problematic smart contracts

Before we write upgradable smart contracts, let's write a simple smart contract that we would like to update in the future as part of enhancements. Using this smart contract, we will understand the problems encountered when we want to upgrade it. The smart contract is called a Bank contract, with a couple of function implementations to debit and/or credit an account. It maintains a global state for each account along with its balance in a mapping data type. Since the mapping has public visibility, a `getter` function is intrinsically generated to access the account balance. There is also a constructor that would initially assign a specified balance to the bank's address. Since the bank is deploying this contract, the value of `msg.sender` in the constructor will have the value of the bank's address:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0

contract Bank {

    mapping (address => uint256) public balances;

    constructor( uint256 amount) public {
        balances[msg.sender] = amount;
    }

    function Debit(address accountAddress, uint256 amount)
        public
        returns (bool) {
        balances[accountAddress] = balances[accountAddress] -
            amount;

        return true;
    }

    function Credit(address accountAddress,uint256 amount)
        public
        returns (bool) {
        balances[accountAddress] = balances[accountAddress] +
            amount;
    }
}
```

```
    return true;
}

}
```

There is another contract named `BankClient` that interacts with the `Bank` contract. It initiates the transactions and ensures that both debits and credits happen within the transaction. An important element to notice in this contract is that the `BankClient` contract does not create an instance of the `Bank` smart contract at all. Instead, it expects that the `Bank` smart contract is already deployed and has an active instance available. The address of the `Bank` contract should be supplied to the `BankClient` constructor while deploying it. The `BankClient` constructor ensures that it stores this instance address in its state variable such that all functions within the smart contract can access the `Bank` instance. If both the `Bank` and `BankClient` contracts are in different `.sol` files, there is a need to explicitly reference `Bank.sol` by including an `import` statement at the beginning of the `.sol` file implementing `BankClient` or any other `import ./Bank.sol` contract file. Here, all the code is assumed to be in the same file:

```
contract BankClient {

    Bank bankContract;

    constructor(address bankc) public {
        bankContract = Bank(bankc);
    }

    function NewTransaction( address to, uint256 amount) public
    returns (bool) {
        bankContract.Debit(msg.sender, amount);
        bankContract.Credit(to, amount);
        return true;
    }
}
```

The steps involved in using these two contracts are as follows:

1. Deploy the `Bank` contract to the Ethereum network. A remixed JavaScript VM can be used for testing purposes. This will generate a new address for the contract. An initial value should be provided as a beginning bank balance during deployment.

2. Deploy the BankClient contract and, during deployment, pass the value of the Bank address generated in the previous step to it as a parameter.
3. Use the NewTransaction function of the BankClient contract.

This is a great solution and works well for the bank until the time comes to modify the Bank smart contract for bug fixes and enhancements.

If a Bank contract is modified for any reason, it should be deployed first before the new changes can be consumed by the BankClient contract, and we know that the deployment of a contract generates a new address with no connection to its previous other instance.

The next section shows a simple solution in which we will use inheritance to solve the upgradability challenge with regard to smart contracts.

Implementing simple solutions with inheritance

We will continue to use the previous Bank and BankClient smart contracts and keep improving them to incorporate upgradability therein.

This solution abstracts the storage or state variables from the smart contract and places them in a different smart contract. Having state and functionality in two separate smart contracts assists in reusing existing smart contracts using the dependency injection pattern that we learned about in the *Understanding dependency injection* section.

In the next example, a new smart contract named BankStorage is listed. This contract just has a declaration for the storage variables. They can additionally have functions whose purpose is to provide read-write functionality for the storage variables. The previous Bank contract has been refactored by removing the state variable to the BankStorage contract.

Next, we create the Bank contract; however, this time we inherit from the BankStorage contract. Inheriting the BankStorage contract helps in accessing the state variable directly from the Bank smart contract. When we deploy the Bank contract, the BankStorage contract is also deployed alongside it.

The remainder of the code is similar to what was shown in the previous section.

The steps involved in using these two contracts are as follows:

1. Deploy the Bank contract to the Ethereum network. A remixed JavaScript VM can be used for testing purposes. This will generate a new address for the contract. An initial value should be provided as a beginning bank balance during deployment. This will also deploy the BankStorage contract intrinsically.
2. Deploy the BankClient contract and, during deployment, pass the value of the Bank address generated in the previous step to it as a parameter.
3. Use the NewTransaction function of the BankClient contract.

Following deployment of the contracts, the BankClient contract will use the Bank contract, which, in turn, will use the BankStorage contract to store and access the storage variables. The next code listing shows all three contracts:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0

contract BankStorage {
    mapping (address => uint256) public balances;

}

contract Bank is BankStorage {

    constructor (uint256 amount) {
        balances[msg.sender] = amount;
    }

    function Debit(address accountAddress, uint256 amount)
        public
        returns (bool) {
        balances[accountAddress] = balances[accountAddress] -
            amount;

        return true;
    }
}
```

```
function Credit(address accountAddress,uint256 amount)
public
    returns (bool) {
    balances[accountAddress] = balances[accountAddress] +
        amount;
    return true;
}

}
```

Note that the `BankClient` contract shown next requires an address during deployment. It also provides a `NewBankAddress` function that can be used for re-assigning a new bank address to it. We will use this feature to inject a new address into `BankClient` after authoring a new smart contract. It should be noted that it does not make a difference whether a new contract is authored or an existing contract modified. In both cases, the contract has to be deployed and the address generated should be passed to the `BankClient` contract:

```
contract BankClient {
    Bank bankContract;

    constructor(address bankc) public {
        bankContract = Bank(bankc);
    }

    function NewBankAddress(address bankc) {
        bankContract = Bank(bankc);
    }

    function NewTransaction( address to, uint256 amount) public
        returns (bool) {
        bankContract.Debit(msg.sender, amount);
        bankContract.Credit(to, amount);
        return true;
    }
}
```

Next, we will author a NewBank smart contract again, inheriting from BankStorage. We can also make a new function that will accept and store the BankStorage contract address within the BankClient contract.

The NewBank smart contract is very similar to the previous Bank contract, but with a few minor enhancements, such as checking the integer overflow and underflow:

```
contract NewBank is BankStorage {  
  
    constructor (uint256 amount) {  
        balances[msg.sender] = amount;  
    }  
  
    function Debit(address accountAddress, uint256 amount)  
    public  
    returns (bool) {  
        require(balances[accountAddress] + amount >= amount);  
        balances[accountAddress] = balances[accountAddress] -  
            amount;  
  
        return true;  
    }  
  
    function Credit(address accountAddress,uint256 amount)  
    public  
    returns (bool) {  
        require(balances[accountAddress] - amount <= amount);  
        balances[accountAddress] = balances[accountAddress] +  
            amount;  
        return true;  
    }  
}
```

The steps involved in using the new Bank contracts have been listed here. It is assumed that the old Bank contract is already deployed:

1. Deploy the NewBank contract to the Ethereum network. A remixed JavaScript VM can be used for testing purposes. This will generate a new address for the contract. An initial value should be provided as a beginning bank balance during deployment.
2. Call the NewBankAddress function in the BankClient contract and pass the address of the NewBank smart contract as a parameter. From this point on, all calls to the NewTransaction function will use the new smart contract instance. It will also get all previously stored values in the state storage variable.
3. Use the NewTransaction function of the BankClient contract.

Let's now learn how to achieve the same solution using the composition technique advocated by object-oriented programming.

Implementing simple solutions with composition

The previous section showed how to create a simple upgradable solution using inheritance. The same can be achieved using composition as well. **Composition** is a well-stable object-orientation concept in which objects are created by composing other objects. The other objects can live independently or can be completely dependent on the parent object.

The same example from the previous section has been used here, but refactored to use composition instead of inheritance. In this example, the BankStorage contract has the storage variable for storing global data along with a couple of getter and setter functions for reading and writing the state variables, as shown here:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0

contract BankStorage {
    mapping (address => uint256) public balances;

    function SetBalance(address addr, uint256 amount) public
        returns
    (bool) {
```

```
        balances[addr] = amount;
    }

    function GetBalance(address addr) public returns (uint256)
{
    return balances[addr];
}

}
```

There is also the Bank contract, but this time it does not inherit from the BankStorage contract. Instead, it expects the address of BankAddress to be supplied to it during deployment:

```
contract Bank{

    BankStorage store;

    constructor (uint256 amount, address bankstorage) {
        store = BankStorage(bankstorage);
        store.SetBalance(msg.sender, amount);
    }

    function SetBankStorage(address bankstorage) public {
        store = BankStorage(bankstorage);
    }

    function Debit(address accountAddress, uint256 amount)
        public
        returns (bool) {
        store.SetBalance(accountAddress, (store.GetBalance
            (accountAddress) - amount));
    }

    return true;
}
```

```
}

function Credit(address accountAddress,uint256 amount)
public
returns (bool) {
    store.SetBalance(accountAddress, store.GetBalance
        (accountAddress) + amount);
    return true;
}

}
```

The client contract does not undergo any changes compared to the previous example and is shown in the following code listing:

```
contract BankClient {

    Bank bankContract;

    constructor(address bankc) {
        bankContract = Bank(bankc);
    }

    function NewBankAddress(address bankc) public {
        bankContract = Bank(bankc);
    }

    function NewTransaction( address to, uint256 amount) public
    returns (bool) {
        bankContract.Debit(msg.sender, amount);
        bankContract.Credit(to, amount);
        return true;
    }
}
```

The steps involved in using these contracts are listed here:

1. Deploy the `BankStorage` contract to the Ethereum network. This will generate a new address for the contract. Make a note of this address as it will be required in the next step.
2. Deploy the `Bank` contract to the Ethereum network. A remixed JavaScript VM can be used for testing purposes. This will generate a new address for the contract. An initial value should be provided as a beginning bank balance during deployment and the address of the `BankStorage` contract should also be provided.
3. Deploy the `BankClient` contract and, during deployment, pass the value of the `Bank` address generated in the previous step to it as a parameter.
4. Use the `NewTransaction` function of the `BankClient` contract.

Now, the `Bank` contract can be updated precisely as shown previously. The `NewBank` contract can be deployed and the `BankClient` contract can be provided with the address using the `NewBankAddress` function:

```
contract NewBank {  
  
    BankStorage store;  
  
    constructor (uint256 amount, address bankstorage) public {  
        store = BankStorage(bankstorage);  
        store.SetBalance(msg.sender, amount);  
    }  
  
    function SetBankStorage(address bankstorage) public {  
        store = BankStorage(bankstorage);  
    }  
  
    function Debit(address accountAddress, uint256 amount)  
    public  
    returns (bool) {  
        uint256 tempBalance = store.GetBalance(accountAddress)  
        -  
        amount;  
    }  
}
```

```
        require(tempBalance >= amount);
        store.SetBalance(accountAddress, tempBalance);

        return true;
    }

    function Credit(address accountAddress,uint256 amount)
public
    returns (bool) {
    uint256 tempBalance = store.GetBalance(accountAddress)
    + amount;
    require(tempBalance >= amount);
    store.SetBalance(accountAddress, tempBalance);
    return true;
}

}
```

This concludes the implementation of upgradable smart contracts using the composition approach. The composition approach is a preferred pattern compared to inheritance, although both are equally applicable from the implementation perspective.

Apart from creating and implementing simple solutions, we can also have advanced solutions, which we will see next.

Implementing advanced solutions using proxy contracts

The word proxy means *on behalf of*. Proxy is one of the established patterns in the design pattern world. Generally, functions are directly invoked in a smart contract. This induces tight coupling between the caller and the contract called. To induce upgradability and manageability, this direct link between contracts should be removed and an additional abstraction called proxy contracts introduced instead. The proxy contract stays in between the caller and called contract and ensures that a two-way request-response operation can be handled between them.

In a proxy contract implementation, the caller does not invoke functions directly on the target contract. Instead, the caller knows about the proxy contract and invokes functions on the proxy contract. It is the job of the proxy contract to take the request from the caller contract, optionally modify the request, and invoke the actual function from the main contract implementing the behavior. The proxy contract does not contain any functional or business logic. It just contains the logic of converting a request and passing that request to another contract by invoking a function on it and returning the values from the target contract back to the caller.

Proxy contracts are intermediary contracts, and they should have the capability to invoke any function within a target contract. This can be achieved using assembly code in Solidity.

The next example uses an interface named `IBanking` with a couple of method declarations. The `MyStorage` contract is simply implemented to declare storage variables with functions to read/write to them:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0

interface IBanking {
    function NewBankingCustomer(uint256 customerid, address
        custaddress ) external;
    function GetBankingCustomer(uint256 customerid ) external
        returns (address);
}

contract mystorage {
    mapping (uint256 => address) public customers;

    function GetCustomer(uint256 customerid) internal returns
        (address) {
        return customers[customerid];
    }

    function SetCustomer(uint256 customerid, address
```

```

        custaddress)
    internal returns (bool) {
        customers[customerid] = custaddress;
        return true;
    }
}

```

Another contract, `ManageCustomer`, is derived from both the `IBanking` interface and the `MyStorage` contract and implements the functions provided by the interface. The implemented functions invoke functions in the `MyStorage` contract to access state variables:

```

contract ManageCustomer is IBanking, mystorage {

function NewBankingCustomer(uint256 customerid, address
    custaddress ) override external {
    SetCustomer(customerid, custaddress);
}

function GetBankingCustomer(uint256 customerid ) override
external
returns
(address) {
    address addr = GetCustomer(customerid);
    return addr;
}

}

```

The `Proxy` contract is the contract responsible for invoking functions on the `ManageCustomer` contract. To invoke functions on this contract, it needs its address, which it accepts using the `AssignCustomer` function (dependency injection). It does not implement the same functions as that of `ManageCustomer`, but rather uses Solidity assembly code within its fallback function to invoke the functions.

As we all know, the fallback function is executed when a function invocation is made without any corresponding matching signature. All function invocations on proxy contracts eventually execute the fallback function. Within the fallback function, the low-level `calldata` value is extracted using the `msg.data` global variable, and using the assembly's `call` function, the actual function is invoked on the target contract, passing in any parameters sent by the client contract. The return value is captured within the assembly code and returned to the client by the proxy contract:

```
contract ProxyContract {  
  
    address maincontract;  
  
    function AssignContract(address addr) public {  
        maincontract = addr;  
    }  
  
    fallback() external payable {  
  
        address mc = maincontract;  
  
        assembly {  
            let startAddress := mload(0x40)  
            calldatcopy(startAddress, 0, calldatasize())  
            let result := call(100000, mc, 0, aa, calldatasize,  
            0, 0 )  
            let sz:= returndatasize()  
            returndatcopy(startAddress, 0, sz)  
  
            if eq(result, 0) {  
                revert(0, 0)  
            }  
  
            return(startAddress, sz)  
        }  
    }  
}
```

The `ProxyContract` contract shown previously does not have any function implementation apart from a fallback function and a function that accepts the address of the target contract. When the fallback function executes, it uses the address of the target contract to invoke the functions using the assembly language. It takes care of any values returned by the target contract and makes sure to return them to the caller. The client contract for the upgradable contract is shown next:

```
contract Client {  
    address maincontract;  
  
    function AssignContract(address addr) public {  
        maincontract = addr;  
    }  
  
    function NewBankingCustomer(uint256 customerid, address  
        custaddress) public returns(bool) {  
        //ProxyContract pc = ProxyContract(maincontract);  
        maincontract.call(msg.data);  
  
        return true;  
    }  
  
    function GetBankingCustomer(uint256 customerid) public  
    returns(bool) {  
        //ProxyContract pc = ProxyContract(maincontract);  
        maincontract.call(msg.data);  
        return true;  
    }  
}
```

In the future, when the `ManageCustomer` contract implements newer functions or deploys newer updated contracts, the address of the new instance can be passed to the proxy contract and it will continue to invoke the functions in the new contract.

This is one of the preferred ways of implementing upgradable patterns within Solidity for smart contracts.

Writing upgradable contracts with upgradable storage

All examples and techniques shown so far in this chapter were dealing with future changes in contract functions. The changes to the functions are generally more than the storage variables. However, the contract variable may require changes. In such cases, there is an additional design pattern that should be implemented alongside the others that were shown earlier in this chapter. This pattern helps in creating upgradable storage variables.

Storage variables are stored in persistent storage, with dynamic types such as arrays and mappings treated differently from native types such as Booleans and integers. Native types are stored in sequence one after another, while arrays and mappings are stored at different locations. Instead of placing variables consecutively in sequence, we can determine the location of these variables dynamically and store our data there. We can create some placeholder variables to start with and store newer data in the future in these placeholder variables.

In the next example, we have some storage variables, each of 32 bytes. These variables are assigned a static value determined by hashing a string value.

A function named `SetEmployeeData` that accepts employee-related data for the first two storage variables. The third storage variable is a placeholder variable, and it might get used in the future.

`employeeId` and `EmployeeName` are assigned to the first two state variables whose location is being determined dynamically. This function uses the Solidity assembly feature to store the values in the storage variable using the `sstore` command. In fact, the code is storing the employee data within Ethereum global state that is determined by the hash value of the state variables. :

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0

contract UpgradableStorage {

    bytes32 internal positionOne = keccak256('random');
    bytes32 internal positionTwo = keccak256('random1');
    bytes32 internal positionThree = keccak256('random2');

    function SetEmployeeData(uint256 employeeId, string memory
        employeeName, string memory placeholder) public {
```

```
        bytes32 localPositionOne = positionOne;
        bytes32 localPositionTwo = positionTwo;
        bytes32 localPositionThree = positionThree;

        assembly {
            sstore(localPositionOne, employeeId)
            sstore(localPositionTwo, mload(employeeName))
            sstore(localPositionThree, mload(holder))

            sstore(localPositionThree,
            mload(add(holder, 0x20)))
        }

    }

function GetEmployeeData() public returns (uint256
employeeId,
    string memory employeeName) {
    bytes32 localPositionOne = positionOne;
    bytes32 localPositionTwo = positionTwo;
    bytes32 localPositionThree = positionThree;

    assembly {
        employeeId := sload(localPositionOne)

        employeeName:= mload(0x40)
        mstore(employeeName, sload(localPositionTwo))
        mstore(add(employeeName, 0x20),
        sload(localPositionThree))
        mstore(0x40, add(employeeName, 0x40))
    }

}
```

Writing smart contracts that can evolve both from a storage and functionality perspective is a quintessential pattern for writing robust and production-ready smart contracts and every developer should think about implementing them in their contracts.

Summary

This chapter was a deep dive into writing smart contracts that can upgrade and evolve in the future. Dependency injection is an important concept for writing upgradable smart contracts, and this was also explained at the beginning of the chapter. Almost every contract will change at some point in time and will need to be upgraded. Writing upgradable contracts is generally a non-functional requirement for any mission-critical contract and a must-have skill for any serious smart contract author. In this chapter, we saw multiple ways of writing upgradable smart contracts, from simple implementations to complex approaches involving assembly code.

In the next chapter, we will get into the details of writing secure smart contracts. It is one of the most important aspects of writing quality smart contracts and plays an important role in taking them to production.

Questions

1. What are the different ways through which a contract address can be supplied to another contract and why would you do that?
2. Is it possible to implement upgradable smart contracts with upgradable storage?
3. Why do we need upgradable smart contracts in the first place?

Further reading

- *Ethereum Cookbook*, by Manoj P. R., Packt Publishing (https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781789133998/7/ch07lvl1sec91/creating-upgradable-smart-contracts)

13

Writing Secure Contracts

Security is an important aspect of any software solution, and smart contracts are no different. In fact, security is more important for smart contracts, since they are custodians of assets and value. Any security lapse can result in hackers siphoning assets belonging to others. There have been multiple cases where multi-million dollars' worth of assets were transferred by hackers to their own accounts. Smart contracts are one of the easiest targets to hack within the Ethereum ecosystem. This chapter will show you some of the ways to solve general security issues within smart contracts. We will also enumerate security best practices for them.

In this chapter, we will cover the following topics:

- The importance of security in smart contracts
- Improvements in Solidity for solving underflow and overflow hacks
- Solving reentrancy hacks in Solidity
- Security best practices from an audit and implementation perspective

By the end of the chapter, you'll know some of the best practices that developers should implement to code-secure smart contracts. We will also demonstrate a couple of the most common security issues with smart contracts, such as overflow, underflow, and reentrancy attacks.

Technical requirements

The following tools are required for working along with this chapter:

- A Chrome browser
- An online Remix editor

The code for the chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter13>.

SafeMath and under/overflow attacks

Solidity in previous versions (prior to 0.8) has been subject to integer overflow and underflow attacks. Before exploring these attacks, it is important to understand what overflow and underflow mean in terms of Solidity.

Integer overflow is a mechanism that occurs when you assign an integer value more than the variable can accept. In such cases, the value assigned is different and calculated by starting over with the minimum value, supported by the data type. For example, `uint8` in Solidity can accept values ranging from 0 to 255. Assigning a value of 256 to `uint8` would assign a value of 1 to the variable. Similarly, assigning 257 to `uint8` would assign a value of 2 to the variable.

Integer underflow is similar to overflow. The difference is that the value assignment happens at the lower boundary for an acceptable value for a datatype. Assigning a value of -1 to `uint8` would recycle its value to 255. Similarly, assigning -2 to `uint8` would assign it a value of 254.

Previously, while doing calculations within smart contracts, it was important to check for underflow and overflow to ensure that the calculation logic and value assigned to variables were correct. OpenZeppelin provides a `SafeMath` library that performs the checks related to underflow and overflow before assigning new values to the variables. OpenZeppelin is a framework that provides multiple reusable standard contracts. These contracts can be imported into our contracts and utilized. `SafeMath` is one such contract that performs integer overflow and underflow checks for major mathematical operations such as addition, subtraction, multiplication, and division.

However, with the release of Solidity 0.8, the EVM itself can now perform the checks and generate errors while generating the bytecode. This makes the use of OpenZeppelin's `SafeMath` redundant; however, there is no harm in using it (although it might increase gas consumption and cost).

The next code block shows the usage of compiler-provided underflow and overflow features. The code has two state variables of the `uint8` type – `upperBoundary` and `lowerBoundary`. The `upperBoundary` state variable is assigned a value of 255 and `lowerBoundary` is assigned a value of 0 in the contract constructor:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract UnderOverFlowContract {

    uint8 public upperBoundary;
    uint8 public lowerBoundary;

    constructor() {
        upperBoundary = 255;
        lowerBoundary = 0;
    }

    function AddtoUpperBoundary(uint8 value) public
        view returns (uint8) {
        return upperBoundary + value;
    }

    function ReducefromLowerBoundary(uint8 value)
        public view returns (uint8) {
        return lowerBoundary - value;
    }

}
```

There are two functions, `AddtoUpperBoundary` and `ReducefromLowerBoundary`, both of which accept a single argument of the `uint8` type and return another `uint8` value.

The `AddtoUpperBoundary` function adds the argument value to the `upperBoundary` state variable and returns the new value. The `ReducefromLowerBoundary` function subtracts the argument value from the `lowerBoundary` state variable and returns the new value. Executing both the functions by supplying a value of 1 to their argument results in an error during execution, as shown in the following figure:

```
CALL  [call]  from: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db  to: UnderOverFlowContract.AddtoUpperBoundary(uint8)
      data: 0x6d9...00001
call to UnderOverFlowContract.ReducefromLowerBoundary

call to UnderOverFlowContract.ReducefromLowerBoundary errored: VM error: revert.

revert
      The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance.
Debug the transaction to get more information.

CALL  [call]  from: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db  to: UnderOverFlowContract.ReducefromLowerBoundary(uint8)
      data: 0xc7f...00001
```

Figure 13.1 – The error result from the function execution, leading to integer overflow

Solidity is maturing, and its developers are adding more secure features in newer versions. Overflow and underflow were issues that every developer had to deal with by writing additional code or using OpenZeppelin code. Now, with the newer version, no action is required for this security aspect in Solidity code.

Next, we will look into a powerful attack scenario on smart contracts known as a **reentrancy attack**.

Reentrancy attack

It is important to protect digital assets stored in smart contracts. Smart contracts are responsible for the transfer of these assets to their owner on demand. However, an underdeveloped smart contract with security bugs can allow a hacker to siphon off all assets using a reentrancy attack. It can have serious consequences because it has the capability to whisk away all funds from the contract.

A reentrancy attack happens when a smart contract implements a function that transfers an asset to an address belonging to a third party. In such cases, a hacker writes a malicious smart contract that acts as one of the users of the smart contract. The malicious smart contract then calls the methods that initiate the transfer of assets; however, it traps the response and makes a recursive callback for the withdrawal of assets. The recursion will continue as long these funds are within the contract.

A reentrancy attack happens because of a lack of proper smart contract coding from a security perspective. It is important to note that such attacks can be prevented by writing robust and secure smart contracts; we will see how to write such contracts later.

One of the tenets of smart contract development is to treat any contract-to-contract interaction as untrusted. This is because a contract contains code, and code can be malicious. So, anytime a contract is talking to another contract, appropriate measures must be taken so that we can eliminate attacks like these.

Having gained the conceptual knowledge behind a reentrancy attack, the next section will implement a scenario to show a reentrancy attack on a non-secure smart contract.

The EtherPot contract

To demonstrate reentrancy attacks, we need a smart contract that accepts, stores, and transfers assets (Ethers in this case). We'll name this smart contract `EtherPot`. The code for `EtherPot` is shown next. Please note that this is not a secure contract and has security bugs.

The contract comprises three functions apart from the `constructor` function. It has a single-balance state variable of the `mapping` type. The mapping maps addresses to their Ether (stored as `wei`) balances stored within the contract. The `constructor` is payable in case the owner wants to deploy the contract with the initial Ether balance (although it is not compulsory).

The `AddEther` function is tagged as payable because any address wanting to deposit Ethers with `EtherPot` has to call this function. Within this function, the mapping is updated with the address and the Ether amount is sent in `wei`. Each contract has an inbuilt Ether balance maintained and can be accessed using the following:

```
address(this).balance
```

The `GetBalance` function returns the currently stored Ether balance with the contract to the caller. This function should ideally only be callable by the owner of the contract. However, currently, it can be called by everyone:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EtherPot {
    mapping(address => uint256) balances;
    constructor() payable {
```

```
}

function AddEther() public payable returns (bool) {
    balances[msg.sender] = balances[msg.sender] +
        msg.value;
    return true;
}

function Withdraw() public returns (bool) {
    require(balances[msg.sender] > 0);
    payable(msg.sender).call{value:
        1000000000000000000}("");
    delete balances[msg.sender];
    return true;
}

function GetBalance() public returns (uint256) {
    return address(this).balance;
}
}
```

The culprit function in the `EtherPot` contract is the `Withdraw` function. This function checks whether the caller has any funds available with the contract and, if so, transfers one Ether to the caller using the `call` method. After transferring the Ether, it removes any balance against the address within the mapping by deleting it.

This contract assumes that its callers are always going to be externally owned accounts. It does not take into consideration that a caller can be another contract as well. This is the fundamental flaw within the contract implementation. Another flaw is that it is using low-level code (a `call` function) to transfer Ether to the caller, although there are better alternatives.

We need another contract to implement an attack on the `EtherPot` contract. This "hacker" contract will deposit Ether with the `EtherPot` contract and then surmount an attack to siphon all Ether available with it.

The Hacker contract

A hacker writes a simple and small smart contract that interacts with the EtherPot contract by becoming one of its users. The Hacker contract consists of similar functions to those available in the EtherPot contract, and it mimics an externally owned account – that is, it can send Ethers, withdraw them, and check the account balance. Additionally, it also has a special `receive()` `payable` function. This is a special function because it cannot be called on demand. It is executed automatically whenever the contract receives Ether. To receive Ether, it should be tagged as `payable`:

```
contract Hacker {
    address payable etherpot;

    constructor(address payable _etherpot) payable {
        etherpot = _etherpot;
    }

    function SendEther() public {
        EtherPot(etherpot).AddEther{value:
            10000000000000000000}();
    }

    function WithdrawEther() public {
        EtherPot(etherpot).Withdraw();
    }

    function GetBalance() public view returns (uint256) {
        return address(this).balance;
    }

    receive() payable external {
        EtherPot(etherpot).Withdraw();
    }
}
```

Note that the hacker is calling the `Withdraw` function from within the `receive` function. This single line of code leads to recursion between two smart contracts. The hacker initiates the withdrawal process by calling the `WithdrawEther` function, which initiates the recursion process.

Within EtherPot's Withdraw function, a transfer of Ether is performed, which automatically invokes the receive function on the Hacker contract. The receive function, in turn, again calls EtherPot's Withdraw function. This process will continue until there is no Ether left within the EtherPot contract.

As you can see, having loopholes in smart contracts that enable a reentrancy attack is far from ideal, and the smart contract needs to be made secure. The next section provides solutions to the reentrancy attack.

Solutions to the reentrancy problem

We can adopt quite a few security best practices to resolve the reentrancy problem. In fact, following some of these security best practices not only solves the reentrancy issue but also many other issues that cannot be envisaged while writing contracts. As part of the solution, we need a target contract and a Hacker contract. The Hacker contract will remain the same from the previous code listing; however, the EtherPot contract will undergo changes because it now implements secure coding principles.

Some of the solutions for a reentrancy attack are mentioned next. These solutions can be implemented together to develop more secure contracts:

- **Check for contract accounts:** This is an optional solution that you can implement to solve reentrancy attacks. In this case, the contract will not allow other contracts to interact with it. If the requirement is that the other contracts should not interact with the current control, the contract accounts can be determined, and the contract can revert the call. However, this practice should not be implemented if a contract has both the contract account and the externally owned account as its users. This practice would use the Solidity `extcodesize` assembly opcode to find the length. If the length is greater than zero, it means it is a contract account. The code for this solution is commented out in the next code listing. Without this code, we will not be able to test the contract. This concept was explained in detail in *Chapter 11, Assembly Programming*.
- **Adopt the Checks-Effects-Interactions paradigm:** We have already discussed this pattern in the previous chapter; however, it is an important pattern from a security perspective, and hence, we cannot skip it here. The previous chapter showed a complete implementation of this pattern, and here is the summary of what was discussed earlier.

The Checks-Effects-Interactions pattern is the three distinct stages in a sequence within a function that changes the contract state and transfers tokens and Ether to other accounts. All incoming argument validation for correctness is executed as part of the checks stage:

- I. The checks stage also includes validating the current state of the contract. By ensuring that the context and environment are in a conducive stage, the checks stage ensures that nothing goes wrong from the state and incoming arguments perspective. It should stop execution if any of the checks fail.
 - II. Then comes the effects stage. At this stage, based on the logic implemented, all state changes are executed – state variables are updated with current values, state transitions are made, and local variables are updated with necessary calculations. At this stage, it is still possible to revert the changes if something untoward happens at the interact stage. Till now, no Ether has been transferred, and no external communication has happened with other third-party addresses and accounts.
 - III. At the last stage, during the interaction, all external communications and Ether transfers happen. This pattern ensures that any interaction happens only after all the state changes. So, even if someone tries to implement a reentrancy attack on the contract, it will not be successful.
- **Validate both the `tx.origin` and `msg.sender` addresses:** This practice helps to establish the fact that the caller is an externally owned account. If both `tx.origin` and `msg.sender` refer to the same address, it means there is no chain of function calls between contracts, and a direct user is calling the function. This is ensured because a contract can never initiate a transaction, and only an externally owned account can start a transaction.
 - **Use better alternates to call a function:** Multiple functions are available to transfer Ethers from one contract to another. If we allow contracts to interact with our contract, there is always a possibility of a reentrancy issue. To avoid this, we should prefer the safer methods of transferring Ether from the contract, such as using the `transfer` function over the `send` and `call` functions. The `transfer` function provides a fixed stipend of 2,300 amount of gas, and any `fallback` or `receive` function will not be able to initiate a recursive call using that amount of gas. It is because gas does not provide adequate feedback, and hence the `transfer` function is a better option to transfer Ethers.

- **Halt the contract:** In spite of implementing all secure coding practices, it is still quite possible that an attack might happen. A smart contract should be able to halt or freeze all operations by implementing a haltable pattern in the event of an attack. This will minimize the loss due to the attack. Haltable patterns are discussed in detail in *Chapter 15, Solidity Design Patterns*.
- **Change the call direction:** This practice is less about implementation and more about the design of the smart contract. In this practice, instead of transferring Ethers using addresses, the contract – after determining that a target is a contract – can initiate a function call and send appropriate Ethers along with the call. This will ensure that no auto-code execution such as fallback or receive functions are executed, and the contract will be in full control over the transfer of Ethers.
- OpenZeppelin provides an out-of-the-box solution related to the reentrancy issue in the smart contract. It provides a library that helps in accelerating the coding of reentrancy checks. The details for this are available at <https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard>.

In the next code listing, the focus is only on the Withdraw function as there are no changes to other parts of the code from the previous listing. The Withdraw function implements the different options available to resolve the reentrancy attack:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EtherPot {
    mapping(address => uint256) balances;
    constructor() payable{
    }

    function AddEther() public payable returns (bool) {
        balances[msg.sender] = balances[msg.sender] + msg.value;
```

```
    return true;
}

function Withdraw() public returns (bool) {
    address sender = msg.sender;
    require(tx.origin == msg.sender);
    require(balances[msg.sender] > 0);
    // use the commented code below in case you dont want
    // contract address to interact with the contract
    /*uint length;
    assembly {
        length := extcodesize(sender)
    }
    if (length > 0) {
        revert("contract addresses are not allowed!");
    } */
    balances[msg.sender] = 0;
    payable(msg.sender).transfer(1000000000000000000000000);
    return true;
}

function GetBalance() public returns (uint256) {
    return address(this).balance;
}
```

Now, this is not an exhaustive list of smart contract security practices. These are some of the important practices that we can follow to reduce the attacks on the contract. There are many other secure practices that we can implement within smart contracts, and they are discussed in the next section.

Security best practices

Some of the important security best practices, both from an audit as well as an implementation perspective, are listed in this section. They should be applied to smart contracts as and when they become an application:

- Each function within a smart contract should have the following:
 - Only the required number of parameters with the most restrictive data type. There should not be extra parameters.
 - The return type from a function should again be appropriately constrained with the right data type.
 - An argument input validation as the first set of actions or steps within a function.
 - Proper exception handling using `requires/revert` functions for input validation and `try-catch` blocks for making external calls to other smart contracts.
 - The correct and appropriate scope, and visibility assigned. If a function can only be called from a contract, do not mark it as public. Similarly, if a function can only be called from an external request, it should be marked as external.
 - Adopt the Checks-Effects-Interactions pattern for any transactions within a function, especially the transfer of tokens or Ether from the contract. This pattern was explained in this book in earlier chapters.
 - Avoid looping over mappings and arrays. It is more costly, time-consuming, and can lead to stack overflow exceptions.
 - Check for gas usage and try to minimize consumption.
 - Apply the principle of least privilege. This principle states that we should allow adequate permissions for a service or user to be able to perform their responsibilities. There should not be any additional permissions beyond the requirement.
 - Reduce or remove unnecessary code to reduce the attack surface area.
 - Validate the use of modifiers in case of repetitive code among multiple functions.
 - Check the mechanism for the transfer of Ethers to third-party addresses. The `transfer` function is better than `send`. The `send` function is better than `call`.

- Security best practices related to contracts include the following:
 - Implement haltable contracts. There can be an attack anytime on a contract, and it should be possible to halt the contract execution immediately to reduce the damage caused.
 - Avoid using low-level functions such as `call`, `callcode`, and others to invoke functions on other contracts or to transfer Ethers to addresses.
 - Avoid `tx.origin` and favor `msg.sender` in contracts. `msg.sender` refers to an immediate caller while `tx.origin` refers to the `origin` caller in chain. `tx.origin` is always an externally owned account while the `msg.sender` value can be a contract account or an externally owned account. Checking for `msg.sender` can provide insights about who is the immediate caller, and then you can take action based on the information.
 - At times, it is beneficial to check whether the account related to `tx.origin` is the same as `msg.sender` by using a simple `require` statement such as the following:

```
require(tx.origin == msg.sender)
```

- Raise events from every function that performs a state change on the contract and provide enough information while emitting events.

This is not a complete list but a substantial one of some secure practices you can follow to develop a smart contract. There are additional security-related practices available at <https://swcregistry.io/>. Some of the security-related tools for smart contracts are available at <https://consensys.net/diligence/tools/>.

Summary

Security is one of the most influential factors when developing a smart contract. No organization can afford a vulnerable smart contract. Organizations are now spending substantial funds on getting their smart contracts audited by security specialists and firms. Multiple tools that automate code analysis and linting smart contracts from a security perspective are available to fix security bugs. Security is important and cannot be ignored.

In this chapter, we saw how Solidity is maturing in fixing some of the underlying issues related to integer overflow and underflow and how it is incorporating safe math within the EVM itself.

Next, we saw how to implement non-secure contracts and how hackers can mount reentrancy attacks on such contracts. This led to a discussion on some of the security best practices that should be implemented by every Solidity developer to ensure that deployed contracts are secure with a minimum attack surface area.

The next chapter will focus on creating different types of tokens within the Ethereum network, going in depth on creating ERC20 and ERC721 tokens.

Questions

1. What are the differences between `tx.origin` and `msg.sender`?
2. How does recursion between contracts happen as part of a reentrancy attack?
3. What is the Checks-Effects-Interactions pattern?

Further reading

- For more information about security considerations for Solidity, check out <https://docs.soliditylang.org/en/v0.8.11/security-considerations.html>.
- For more information about security recommendations for smart contracts written in Solidity, check out <https://docs.soliditylang.org/en/v0.8.11/security-considerations.html#recommendations>.

14

Writing Token Contracts

Ethereum is one of the most important blockchain platforms around. One of the major reasons for its popularity is that Ethereum natively supports a token – Ether. It's well known that Ether is the native currency of Ethereum. Ethereum involves Ether in various denominations for every operation that changes its state, even when invoking simple contract operations such as raising an event or storing some value within a ledger. In short, cryptocurrency is built into Ethereum.

Despite the fact that Ethereum has its own currency, it allows developers to create a new currency in the form of tokens. This chapter is all about tokens, creating new tokens, and especially tokens based on ERC20 and ERC721 standards. **ERC** stands for **Ethereum Requests for Comment** and is the closest thing Ethereum has to a standard ratification process.

This chapter is focused on building tokens and will cover the following topics:

- Creating a new ERC20 token
- Creating a new ERC721 token
- Using ERC165
- Using ERC223

By the end of the chapter, you'll be equipped to create your own ERC20- and ERC721-based tokens.

Technical requirements

The following tools are required for working along with this chapter:

- A Chrome browser
- An online Remix editor

You can find the GitHub link to the chapter at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter14>.

Introducing tokens

Before getting into building our own tokens, it is important to understand the fundamental concept of tokens.

Tokens are everywhere. Imagine going to a funfair full of activities, rides, and sports. Upon entry, you get given some tokens that can then be exchanged for multiple rides or activities. The token is the currency of the fair. The fair stalls recognize tokens and do not accept any other currency, even though you get these tokens in exchange for some other currency to start with.

Another example of the usage of tokens is in casinos. You can buy tokens after entering the casino and then continue to play on different tables using the same tokens. Again, these tokens are the currency for the casino, and the tables do not accept any other currency.

So, what do we understand about tokens from the previous two examples? Tokens are a generic concept and can represent anything. A token is meant for gambling within a casino and, at the same time, it can be used for enjoying rides at a fair. Now, if we apply the same thought process in the Ethereum world, tokens are the cryptocurrency that can be used on the Ethereum network.

There are other blockchains supporting other cryptocurrencies, and they conceptually mean the same thing; however, they may be created differently. Each **Decentralized Application Organization (DAO)** can develop its own token, with specific use cases based on business needs.

ERC20 tokens were the first and, by far, the most prevalent tokens within the Ethereum network. They will be discussed in the next section.

ERC20 Tokens

ERCs are standards published by the Ethereum foundation. ERC is a set of documents that list the rules and standards for a particular scenario and use case. There are multiple ERCs, each referred to using a number. For example, ERC20 is a document that lists down the rules and standards for defining a **fungible token** on the Ethereum network. Similarly, ERC721 is another document that lists rules and standards for defining a **non-fungible token** on the Ethereum network. Some of the well-known ERCs are ERC20, ERC165, ERC721, ERC1155, and ERC223. There are many more ERC standards; however, we are going to concentrate on ERC20 and ERC721 in this chapter.

Another important question that begs to be answered is why tokens need a standard. It is evident by now that a token has value just like any other currency. Using tokens, you can purchase services and assets and exchange them with others, just like any other currency. Moreover, tokens are interoperable with Ether along with other assets and services. Such a valuable concept needs standards that everyone can adhere to so that the network can maintain complete fidelity when tokens are exchanged.

In the absence of such a standard, every developer would be free to develop tokens in a custom way that might not be interoperable or usable with Ether or other tokens. This will reduce the usability and effectiveness of cryptocurrencies within the Ethereum ecosystem.

Another important concept to understand before developing an ERC20 token is fungible tokens. The literal meaning of fungible is *mutually interchangeable* or a token replaceable by another one. Fungible tokens are identical to each other, and it is not possible to distinguish one from another.

For example, having two Ethers in a wallet would mean two identical Ethers. There is no way to provide an identity to the first Ether one and another identity to the second Ether. They are similar in all aspects. This concept is similar to dollar coins in circulation – two coins are similar in every aspect. They have the same value, and using one over another makes no difference in value. They cannot be distinguished because they do not have any identity.

It is important to understand the ERC20 standard before getting into its implementation, which we will discuss in the next section.

ERC20 standard

The ERC20 standard provides rules that, when implemented, render a smart contract as an ERC20 compliant contract. These standards are as follows:

- A token can be sent or transferred from one account to another. The accounts can be externally owned or contract accounts.
- It should be possible to query the contract to get the token balance for any account. Just like a currency, the contract should provide the facility for anyone to query the amount of token held against their address.
- It should also be possible to query the contract to get the total supply of tokens in circulation for all accounts.
- And lastly, a token can be authorized by an account to be spent by another account. This is also known as spent on behalf of an account. The authorized account should not be able to spend more tokens than have been authorized by the authorizer.

You should implement these rules within a smart contract of the hierarchy of smart contracts (in case of inheritance) to adhere to and comply with ERC20 standards. The ERC20 standard also provides the function signatures for implementing these rules. Every smart contract that wants to comply with ERC20 must implement the following functions. The function signatures are shown in the next section.

ERC20 functionality

ERC20 tokens should implement minimal functionality to be used as a medium of exchange and establish ownership. Any contract that implements the ERC20 token should pay special attention to providing Solidity functions that help in establishing token ownership as well as transferring ownership from one account to another. The contract functions that help achieve these two aspects of ERC20 are as follows:

- `totalSupply`: An ERC20 contract should implement the `totalSupply` function that returns the total supply of tokens available as an `int256` data type. This function does not change any state of the contract:

```
function totalSupply() public view returns (uint256)
```

- **balanceOf:** An ERC20 contract should implement the `balanceOf` function. The function should accept an address as an argument. The function should return the balance of the address provided to the caller by querying the state variables. This function does not change any state of the contract:

```
function balanceOf(address _owner) public view returns  
(uint256 balance)
```

- **transfer:** An ERC20 contract should implement the `transfer` function. The function should accept an address as an argument. This address is the beneficiary address, and tokens will be transferred to it. The number of tokens to be transferred is supplied as the `_value` argument. This function needs to be called by an account whose token balance is maintained within the contract and whose address is implicitly passed to the contract as part of a global variable (`msg.sender`). The function returns *success* or *failure* using the Boolean data type. The function changes the state of the contract by updating its state variables:

```
function transfer(address _to, uint256 _value) public  
returns (bool success)
```

- **transferFrom:** An ERC20 contract should implement the `transferFrom` function. This function should accept an address as a `_from` argument. The tokens from this address will be transferred to another address. The beneficiary address is provided as an argument named `_to`, and the `_value` argument represents the number of tokens to be transferred. This function is invoked by an account that has been authorized by another account to act on its behalf. The account address invoking the function is implicitly passed to the contract as part of the global variable (`msg.sender`). The function returns *success* or *failure* using the Boolean data type. The function changes the state of the contract by updating its state variables:

```
function transferFrom(address _from, address _to,  
uint256 _value) public returns (bool success)
```

- **approve:** An ERC20 contract should implement the `approve` function that accepts an address that is getting authorized to spend or transfer tokens on behalf of another account. This function should be invoked by a caller account that wants to authorize another account to spend tokens on its behalf, and the number of tokens is represented by the `_value` parameter. The account address invoking the function is implicitly passed to the contract as part of the global variable (`msg.sender`). The function returns success or failure using the Boolean data type. The function changes the state of the contract by updating its state variables:

```
function approve(address _spender, uint256 _value)
public returns (bool success)
```

- **allowance:** A contract should implement the `allowance` function that accepts two addresses – the address of the original owner who had authorized another account to spend on their behalf, and an address that is getting authorized to spend tokens on behalf of its owner. This function does not change any state of the contract. It simply queries the contract variables and returns an appropriate value based on the given address as an unsigned integer to the caller. This function can be called by anyone, provided both the addresses (owner and spender) are known:

```
function allowance(address _owner, address _spender)
public view returns (uint256 remaining)
```

Obviously, in addition to these functions, you can implement other functions within the smart contract that will still comply with ERC20 standards. However, it is important that these functions are available within the contract.

The ERC20 standard provides a couple of events that can be used within a token contract, which we will discuss next.

ERC20 events

Optionally, there are a couple of events that can also be implemented along with the aforementioned functions. It is a good practice to implement them within the contract, but they are not mandatory. These events are as follows:

- `event Transfer(address indexed _from, address indexed _to, uint256 _value)`
- `event Approval(address indexed _owner, address indexed _spender, uint256 _value)`

As we know, events should be fired to denote the state changes happening within a smart contract. These events should be fired from the functions discussed in the prior section. The `transfer` event should be emitted from the `transfer`, the `transferFrom` function, and the `approval` event should be emitted from the `approve` function. Both the events expect addresses between the two parties – that is, from an account and to an account – along with the number of tokens transmitted or approved.

Now, it's time to implement a smart contract that will also represent our sample token, known as `WorldToken`. This token will be compliant with the ERC20 standard:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract WorldToken {

    function name() public view returns (string) { }
    function symbol() public view returns (string) { }
    function decimals() public view returns (uint8) { }
    function totalSupply() public view returns (uint256) { }
    function balanceOf(address _owner) public view returns
        (uint256 balance) { }
    function transfer(address _to, uint256 _value) public
        returns (bool success) { }
    function transferFrom(address _from, address _to, uint256
        _value) public returns (bool success) { }
    function approve(address _spender, uint256 _value) public
        returns (bool success) { }
    function allowance(address _owner, address _spender)
        public view returns (uint256 remaining) { }

    event Transfer(address indexed _from, address indexed
        _to, uint256 _value)
    event Approval(address indexed _owner, address indexed
        _spender, uint256 _value)
}

}
```

Initially, all tokens in the `WorldToken` contract are assigned to the owner of the contract. The owner of the contract is the account that deploys the contract on the Ethereum network. The owner account provides the number of tokens to be generated as input during the deployment of the contract. The count of these tokens also becomes the total supply of tokens.

At the time of deployment, the owner of the contract will supply the name of the token and its symbol as arguments. All these activities should happen during the deployment of the contract, and thus the contract constructor should contain and execute these bits of code. The constructor code for the contract is shown next.

The constructor accepts a single parameter – the total supply or count of tokens. It updates the `_totalSupply` state variable with this value and updates the owner state variable with the deployer's account address, using the `msg.sender` global variable. It also updates the balances' mapping and assigns the entire token supply to the owner of the contract. It then assigns a value to both the name and symbol state variables. The code for the constructor is shown next:

```
constructor(uint256 totalSupply) {
    _totalSupply = totalSupply;
    owner = payable(msg.sender);

    balances[msg.sender] = _totalSupply;

    name = "WorldToken";
    symbol = "WOT";
}
```

There are quite a few state variables within this token contract. The name and symbol state variables are self-explanatory. The total supply state variable contains the current supply of tokens across all accounts. If the total supply of tokens needs to be increased (minting) or decreased, then these variables should be updated.

The address of the owner of the contract is maintained within the `owner` state variable. This is declared using the `payable` modifier because eventually, when the token is sold, the Ether (received as part of the sale proceeds) should be transferred to this account. If this address is not qualified with the `payable` qualifier, Solidity will not allow the transfer of Ether to this account from the contract.

Finally, there are a couple of mapping variables – balances and allowed. The balances mapping stores the number of tokens available to each address. The allowed variable is a nested mapping used to authorize third-party accounts to spend on behalf of other accounts.

The next code listing declares the name and symbol state variables. It also declares the balances mapping along with a nested mapping for storing the relationship between an owner and its authorized spender:

```
string public name;
string public symbol;

mapping(address => uint256) balances;
mapping(address => mapping (address => uint256)) allowed;

uint256 _totalSupply;
address payable owner;
```

The totalSupply function is one of the simplest functions within the contract. It returns the total supply of tokens within the contract. The implemented function returns the current value stored within the _totalSupply state variable.

The next code listing shows the totalSupply function returning the current total number of tokens in circulation:

```
function totalSupply() public view returns (uint256) {
    return _totalSupply;
}
```

The balanceOf function accepts an address as an argument and uses it to find and return the current token balance available within the balances mapping. It is important to note that anybody can call this function by supplying an address to get the balance value. It is not only the owner of the tokens who can call this function. This function is open for anyone to call:

```
function balanceOf(address _owner) public view returns
    (uint256 balance) {
    return balances[_owner];
}
```

By now, we are able to assign a name and symbol for our NFT and are also able to find the current supply of NFTs, along with the individual owner's balance. Next, we need to understand the mechanism of the exchange or transfer of tokens between accounts. The `transfer` function helps in the transfer of tokens from one account to another.

The `transfer` function accepts a receiver's address and the number of tokens to transfer from its owner's account. Note that the owner's account is not a part of the function signature. The owner's address is implicitly passed and available as a global `msg.sender` variable. Every time a contract function is invoked, the `msg.sender` global variable is filled with the caller's address value and made available to function code. This ensures that only the owner of a token can transfer their tokens, and there is no means through which others can transfer their tokens.

The implementation for the `transfer` function is shown in the next code listing. It accepts two arguments – `_to` and `_value` – and transfers the ownership of tokens from one account to another. This happens by reducing the count from the `balances` mapping for the owner account and increasing the same amount for the new owner account:

```
function transfer(address _to, uint256 _value) public
    returns (bool success) {
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
}
```

The `transfer` function checks whether the sender has enough balance to transfer the tokens to the receiver's account using the `require` function. If there is not enough balance, an exception will be raised, and token transfer will not occur.

However, if there is enough balance, the sender's balance reduces after the transfer, while the receiver's balance increases. Eventually, the `transfer` event is raised, denoting a successful transfer of a token, and a `true` value is returned from the function.

The `approve`, `allowance`, and `transferFrom` functions are closely related to each other. These functions work together to implement the transfer of token ownership from a current owner to a new owner by an authorized spender.

The flow for authorizing and allowing a third party to spend on behalf of an account takes place in this manner:

1. The original account holder of tokens approves another account, authorizing it to spend on their behalf, alongside the amount they can spend. Only the token owner can perform this activity. The approved tokens are also known as an **allowance**.
2. Anybody can query the token contract by supplying the address of both the token owner and spender address to find the available balance for spending by the spender. The spender need not spend its entire allowance in one transaction. They can transfer tokens on behalf of the token owner in a piece-meal manner. Therefore, it helps to query the contract and check the current balance that the third-party account can spend.
3. The transaction of tokens – by a third-party account on behalf of the token owner account – happens using the `transferFrom` function.

From a token contract perspective, each of these three steps maps respectively to the `approve`, `allowance`, and `transferFrom` functions. The only state variables involved in these three functions are the `balances` and `allowed` mappings. The `allowed` mapping is a mapping of mapping – that is, it is a nested mapping. A nested mapping is required to establish a relationship between three variables – the token owner, the spender, and the token ID.

The key for outer mapping is the address of the token owner, the key for inner mapping is the address of the spender, and the value in inner mapping is the count of tokens.

The `approve` function is again quite simple. It updates the `allowed` mapping by adding new tokens to an existing number of tokens, if any; otherwise, it just adds another entry within the mapping, with the first two keys as account addresses. It then emits the `Approval` event and returns `success` from the function.

The implementation of the `approve` function is shown in the following code listing. The code authorizes an address to spend the mentioned number of tokens on its behalf:

```
function approve(address _spender, uint256 _value) public
    returns (bool success) {
    allowed[msg.sender][_spender] = allowed[msg.sender]
        [_spender] + _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

Again, the `allowance` function just returns the current balance available within the allowed mapping for a given combination of the original token owner address and the spender account address.

The implementation of the `allowance` function is shown in the following code listing. The code simply returns the number of tokens that a spender has available as a balance:

```
function allowance(address _owner, address _spender)
    public view returns (uint256 remaining) {
    return allowed[_owner][_spender];
}
```

The `transferFrom` function is very similar to the `transfer` function. It also does exactly the same activities as the `transfer` function. The difference is that `transferFrom` gets the sender's account address as a part of its arguments. The `transferFrom` function is invoked by the authorized third-party address. The function should be supplied by both the owner's and receiver's addresses. The function reduces the tokens in the owner's balance while increasing by the same amount in the receiver's balance. It also reduces the balance that a spender can spend on behalf of the owner.

The implementation of the `transferFrom` function is shown in the following code listing. The code is executed by a spender and transfers ownership of a set of tokens represented by the `_value` argument from their current owner (`_from address`) to a new owner (`_to address`):

```
function transferFrom(address _from, address _to, uint256
    _value) public returns (bool success) {
    require(_value <= balances[_from]);
    require(_value <= allowed[_from][msg.sender]);
    balances[_from] = balances[_from] - _value;
    allowed[_from][msg.sender] = allowed[_from]
        [msg.sender] - _value;
    balances[_to] = balances[_to] + _value;
    emit Transfer(_from, _to, _value);
    return true;
}
```

The implementation of all the functions ensures that `WorldToken` is an ERC20-compliant contract. Now, how can an account address buy `WorldToken` and become the owner of the tokens? The contract can sell tokens to accounts in multiple ways. The two main ways are to implement an additional payable function (`buyTokens`) that can be invoked explicitly to buy tokens in exchange for Ethers. Another method is to implement a payable `Receive` function.

We already discussed the `receive` functions in detail in *Chapter 8, Exceptions, Events, and Logging*. It is important to note that the logic of selling tokens from a contract in exchange for Ethers is the same for both `receive` and `custom` functions.

A `custom` function is used to sell tokens in exchange for Ethers and is shown in the following code block. It is important to note that this function is not a part of the ERC20 standard:

```
function buyTokens() public payable returns (bool
success) {
    require (msg.value > 0);
    uint256 num_of_tokens = ((msg.value/
    1000000000000000000) *10);
    balances[msg.sender] = balances[msg.sender] +
    num_of_tokens;
    balances[owner] = balances[owner] - num_of_tokens;
    owner.transfer(msg.value);
    return true;
}
```

The first aspect that a developer or contract owner needs to decide is the number of tokens to exchange for Ether. In this case, 10 World Tokens will be sold for each Ether. This calculation is shown in the second line of the `buyTokens` function. The `msg.value` global variable consists of the Ether value in terms of Wei. This is the amount sent by an address (maybe using MetaMask). This value is divided by 1000,000,000,000,000,000 (1 followed by 18 zeros) to convert the wei amount into Ether. For example, 200,000,000,000,000 will be converted into 2 Ethers, and multiplying by 10 ensures that 20 WorldTokens will be exchanged for 2 Ethers sent by the address.

Once the number of tokens is ascertained, the rest is a simple reduction from the contract owner's address and the addition of the same to the caller's address. Note that all tokens initially belong to the contract owner, and only they can transfer them when someone buys a token from the contract.

The received Ethers are transferred to the contract owner's address using the `transfer` function provided by the `address` data type (note that this `transfer` function is not the same `transfer` function implemented within the `WorldToken` contract).

The `receive` function has the same code as that of `buyToken`, as shown in the following code block:

```
receive() public payable {
    require (msg.value > 0);
    uint256 num_of_tokens = ((msg.value/
        1000000000000000000) *10);
    balances[msg.sender] = balances[msg.sender] +
        num_of_tokens;
    balances[owner] = balances[owner] - num_of_tokens;
    owner.transfer(msg.value);
}
```

The main difference between the `buyToken` and the `receive` functions is that `buyToken` should be invoked explicitly by a caller. They should know that such a function exists within the contract. The `receive` function is invoked by a simple transfer of Ethers. The caller need not know about its existence. They can send the Ethers using any wallet, and the `receive` function will be invoked automatically. It is a good practice to implement both methods of selling tokens so that there are no lost opportunities.

The account addresses that bought tokens from the contract using the `buyTokens` function can transfer their tokens to another address using the `transfer` function. Alternatively, they can authorize other account addresses to manage them on their behalf but cannot sell their tokens using this contract. They have to either transfer their tokens to a crypto exchange or execute a wallet-to-wallet transfer.

We have covered the details of Fungible and ERC20 tokens so far with a complete implementation. Now, it's time to get into the details of another type of token, known as the **Non-Fungible Token (NFT)**. NFTs will be discussed in the next section.

Non-fungible tokens

One of the most prominent innovations on the Ethereum network has undoubtedly been the concept of **NFTs**. We saw the process and implementation for creating fungible tokens using the ERC20 specification, and now, we will get into the details of creating our very own NFT.

ERC20s are fungible tokens, meaning they are interchangeable. You cannot distinguish one token from another token. All Ethers are the same and provide the same value. What matters is how many tokens are in the wallet and not their identity.

Non-fungible tokens, on the other hand, are not interchangeable. Each NFT has an identity associated with it and each token has a different value, even though they are issued by the same platform.

Each NFT is unique, with different features, characteristics, and values, although they all might belong to the same class of token. For example, CryptoPunks is the issue for all NFTs belonging to it. However, all NFTs issued by CryptoPunks as a class have different values and characteristics.

Imagine you are holding multiple variants of a credit card from a bank. Each card has an identity, features, and characteristics and provides a different value than other cards. A card cannot be interchanged for another card. This is analogous to the concept of NFT. Each credit card can be a single NFT, and they all belong to the same issue – in this case, a bank.

This section of the chapter is about building an NFT using Solidity. NFTs are based on the ERC721 specification and interface. The ERC721 interface provides all the methods needed for the appropriate functioning of an NFT. It helps in minting or generating a new NFT, establishing and transferring the ownership of a token, burning or destroying the token, querying for existing owners of the token, and authorizing other accounts to manage the token on behalf of the original owner.

Another important behavior of an NFT is that it is not divisible. If you remember, ERC20 tokens and even Ether is divisible, and less than one token can be used for transfer and other purposes. However, this is not possible with NFTs. An NFT as a token is managed without getting divided further; it will lose its functionality and features if it is allowed to be broken down into smaller quantities.

Due to their uniqueness and indivisibility, NFTs can be used to represent anything, including land, digital art, valuable documents, images, and rare items. These assets can be stored as NFTs on a blockchain and traded with traceability and complete ownership.

With this background information on NFTs, it's time to get into the details of ERC721 and its implementation from the next section onward.

ERC721

As mentioned earlier, ERC721 defines a standard and is available at <https://eips.ethereum.org/EIPS/eip-721>. This Solidity interface provides details and defines functions and events that a derived contract should implement to develop an NFT.

It is to be noted that the specification only provides function signatures, and implementation is left to the developer of the NFT smart contract. The developer is free to implement the necessary functions that best fit their requirements. Moreover, the derived contract can contain additional functions, state variables, and every other contract artifact, besides the functions defined by the ERC721 specification. However, it is mandatory to implement all the functions in the derived contract to be able to deploy the NFT contract on the Ethereum network.

The ERC721 specification is shown in the following code block. It comprises three events and nine functions. These functions will be explained with an implementation later in this chapter:

```
interface IERC721 {  
  
    event Transfer(address indexed from, address indexed to,  
        uint256 indexed tokenId);  
    event Approval(address indexed owner, address indexed  
        approved, uint256 indexed tokenId);  
    event ApprovalForAll(address indexed owner, address  
        indexed operator, bool approved);  
  
    function balanceOf(address owner) external view returns  
        (uint256 balance);  
    function ownerOf(uint256 tokenId) external view returns  
        (address owner);  
    function safeTransferFrom(address from, address to,  
        uint256 tokenId, bytes calldata data) external;  
    function safeTransferFrom(address from, address to, uint256  
        tokenId) external;  
    function transferFrom(address from, address to, uint256  
        tokenId) external;  
    function approve(address to, uint256 tokenId) external;  
    function setApprovalForAll(address operator, bool  
        _approved) external;
```

```

function getApproved(uint256 tokenId) external view
    returns (address operator);
function isApprovedForAll(address owner, address
    operator) external view returns (bool);
}

```

There are quite a few functions within this interface, and a complete ERC721 implementation should have them implemented along with other helper functions. With this knowledge about the interface, we can drill down into its implementation.

The ERC721 implementation

The ERC721 Solidity implementation defines the NFT contract and implements the ERC721 and ERC165 interfaces. It also defines all the relevant state variables to store and manage the NFTs, along with their ownership, interface function implementations, and additional internal and external functions such as minting, burning, and using ERC721-defined events in the function to let consumers know about a state change within a contract.

The definition of the contract is shown in the following code block. It implements both the IERC165 and IERC721 interfaces defined earlier. The contract defines a `CryptoNFT` NFT:

```

contract CryptoNFT is IERC165, IERC721 {
}

```

The contract declares multiple state variables to manage the state of its NFTs. It declares variables for both the name and symbol of the NFT. The value for these variables is assigned during contract deployment within the `constructor` constructor.

The constructor code is shown next. It accepts two arguments – one for each name and symbol for the NFT. It assigns the arguments to their respective state variables:

```

string private _name;
string private _symbol;

constructor(string memory name_, string memory symbol_) {
    _name = name_;
    _symbol = symbol_;
}

```

Note that both these variables have a private scope, and they can only be assigned values using the constructor. However, they can be queried anytime with the help of `getter` functions – `name` and `symbol`. Both `name` and `symbol` are public functions and return the currently stored values in the `_name` and `_symbol` state variables, as shown in the following code listing:

```
function name() public view returns (string memory) {
    return _name;
}

function symbol() public view returns (string memory) {
    return _symbol;
}
```

The next set of variables helps in establishing the ownership of an NFT. They also help in transferring ownership from one account to another, as shown here:

```
mapping(address => uint256) internal ownedTokens;
mapping(uint256 => address) internal tokenOwner;
mapping(address => uint256) internal ownedTokenCount;
```

Three mappings are required to establish the ownership of an NFT and to transfer the ownership from one account to another. They are simple mappings – `ownedTokens` is a mapping from an address or owner to `tokenId`, `tokenOwner` is a mapping from `tokenId` to an address, and `ownedTokenCount` is a mapping of an address or owner to the number of tokens owned by the address.

The `ownedToken` mapping is used to store the ownership of tokens. It is frequently used to look up and find `tokenId` based on an owner's address. Similarly, the `tokenOwner` mapping is used to store the ownership of tokens that can be used to find the owner's address using `tokenId`.

The question might arise why two mappings are required instead of one. The reason is to optimize and reduce gas usage. It is possible to maintain ownership and transfer ownership using `ownedTokens`; however, finding an owner using `tokenId` will result in the function looping through the entire `ownedTokens` mapping. This will be an expensive operation, and it will slow down the execution. The `ownedTokenCount` mapping helps to find the number of tokens owned by an address.

The code for both the `ownerOf` and `balanceOf` functions is shown in the following listing. These functions return the value stored in state variables based on `tokenId` and the owner's address:

```
function ownerOf(uint256 tokenId) public override view
    returns (address owner) {
    return tokenOwner[tokenId];
}

function balanceOf(address owner) external override view
    returns (uint256 balance) {
    return ownedTokensCount[owner];
}
```

The ERC721 interface provides two functions – `ownerOf` and `balanceOf` – that directly interact with and return the current values stored in the `tokenOwner` and `ownedTokenCount` mappings. The `ownerOf` function accepts `tokenId`, queries the `tokenOwner` mapping using the same, and returns the address or owner of the token.

Similarly, the `balanceOf` function accepts an address, queries the `ownedTokenCount` mapping, and returns the number of tokens owned by the address. These functions primarily help find the current state and ownership of the tokens. These functions do not bring any changes in ownership of the tokens.

The next important function that you should implement in an NFT contract is the `mint` function, which is responsible for generating new NFTs. Without generating new tokens, the contract would almost be useless, as no one would be interacting with the contract. The `mint` function accepts an address that represents the owner of the token to be generated along with `tokenId`.

Generally, this function is called from a frontend portal listing all NFTs. Once a user selects to buy a token, the address of the user along with the selected `tokenId` would be supplied to the `mint` function. The `mint` function will validate the address and the token ID. After the validation is successful, the function increments the count of tokens owned by the supplied address by adding it to the `ownedTokenCount` mapping. It also adds a new entry within the `tokenOwner` and `ownedTokens` mappings. This establishes that the supplied address becomes the owner of `tokenId`.

Finally, the `transfer` event is raised, with values supplied to the function. This is one of the possible implementations of the `mint` function. There can be other implementations, depending on requirements.

For example, `tokenId` can be a string or a URL, and then the token should also have additional metadata associated with it. Also, this function should be external so that it can only be invoked by externally owned accounts.

The code for the `mint` function is shown next, and as mentioned before, this is just one way to implement this function. It can be implemented using different logic as well:

```
function mint(address to, uint256 tokenId) external {
    require ( to != address(0) , "address cannot be default
        null address");
    require (tokenId > 0, "token id cannot be zero or
        less!");
    ownedTokensCount [to] += 1;
    tokenOwner [tokenId] = to;
    ownedTokens [to] = tokenId;

    emit Transfer(msg.sender, to, tokenId);
}
```

If a contract has a function to generate an NFT, it should also be possible to destroy or burn the token. This is exactly what the `burn` function does within the contract. Again, this function is not part of the specification, and developers are free to implement it as necessary. This function should be external so that it can only be invoked by externally owned accounts.

The `burn` function accepts `tokenId`. Generally, this function is called from a frontend application listing all NFTs. The function checks for a valid address as an argument and whether `tokenId` is greater than zero. Since the address is not supplied, the address for a token is fetched using the `ownerOf` function shown earlier.

After validation success, the function decrements the tokens owned by the address by subtracting one from the `ownedTokenCount` mapping. It also deletes the existing entry within the `tokenOwner` and `ownedTokens` mappings by using the `delete` keyword. `delete` updates the value for the address and token to the data type default value. This establishes the fact that the token is burned and no longer owned by an address.

Finally, again, the `transfer` event is raised, with argument values supplied to the function. The next code listing shows the implementation of the `burn` function. This is also one of the possible implementations for the `burn` function. There can be other implementations, depending on requirements:

```
function burn(uint256 tokenId) public {
    require (tokenId > 0, "token id cannot be zero or
less!");
    address from = ownerOf(tokenId);
    tokenApprovals[tokenId] = address(0);
    ownedTokensCount[from] = ownedTokensCount[from] - 1;
    delete tokenOwner[tokenId];
    delete ownedTokens[from];
    emit Transfer(from, address(0), tokenId);
}
```

There are a couple of other EIP specifications that should also be implemented in the derived NFT smart contract to make it more robust and usable. These include **EIP-165** and **EIP-223**.

The next variable helps authorize another account to operate on a token owned by someone else. This variable has a nested mapping, with outer mapping maps addressing an inner mapping. The outer address is the address of the token's owner. The inner mapping also has an address mapping to a Boolean value. This inner address is the address of the spender/operator authorized to manage the token on behalf of the original owner. The `true` Boolean value allows it to be managed by this address and vice-versa.

Note that this operator address is just like a co-owner of the token and can do everything an owner can do:

```
mapping(address => mapping(address => bool)) internal
operatorApprovals;
```

To manage this variable to access and update it, the `setApprovalForAll` and `isApprovedForAll` functions are provided by the ERC721 specification, as shown next:

```
function setApprovalForAll(address operator, bool
approved) external override {
require (operator != address(0) , "operator address
cannot be default null address");
```

```
operatorApprovals[msg.sender][operator] = approved;
emit ApprovalForAll(msg.sender, operator, approved);
}

function isApprovedForAll(address owner, address
operator) public override view returns (bool) {
return operatorApprovals[owner][operator];
}
```

A contract should implement the `setApprovalForAll` function that accepts an address as an argument that will be authorized to spend or transfer tokens on behalf of the owner. This function should only be invoked by the owner's account.

The owner can call the function for both, allowing as well as disallowing a third-party account to manage their tokens, by sending `true` or `false` values for the `approved` parameter. The owner address invoking the function is implicitly passed to the contract as part of the global variable (`msg.sender`). The function adds an entry or updates an existing entry into the `operatorApprovals` mapping.

Note that this function sets authorization approval for all tokens owned by the owner. If it is required to provide authorization approvals for a single token, then an alternative implementation is required for this function.

A contract should implement the `isApprovedForAll` function and it accepts two addresses – the address of the original owner who had authorized another account to spend on their behalf, and an address that is getting authorized to spend tokens on behalf of the original owner. This function does not change any state of the contract. It simply queries the `operatorApprovals` state variables and returns `true` or `false`. This function can be called by anyone, provided both the addresses (the owner's and the spender's) are known.

One of the functions of the token is that an owner of it can authorize a spender account to manage its tokens. To perform this activity, we need another function within the contract along with state variables to manage the relationship between the two accounts. Note that this is separate from the concepts of the operator approvals we saw before. **Operators** are similar to co-owners; however, approved accounts can simply transfer tokens from one account to another on behalf of an owner.

This `tokenApprovals` mapping stores a map between `tokenId` and its spender's address. The `approve` function is responsible for adding an address as an approver against `tokenId`. The function accepts the spender's address and the token ID. It validates the spender's and the owner's addresses and checks whether the token ID actually belongs to the owner. This function should be called by an owner or an operator who is approved by the owner, and this address is captured using the `msg.sender` global variable. If all validations are successful, the spender address is added to the `tokenApproval` mapping against the given token ID.

The implementation for the `approve` function along with the state variable it has an impact on is shown next. The `approve` function updates the `tokenApprovals` mapping, so they are shown together:

```
mapping(uint256 => address) internal tokenApprovals;

function approve(address spender, uint256 tokenId)
    external override {
    require (tokenId > 0, "token id cannot be zero or
        less!");
    require ( spender != address(0) , "to address cannot be
        default null address");
    address owner = tokenOwner[tokenId];
    require( msg.sender == owner || operatorApprovals
        [owner][msg.sender], "not owner nor approved for
        all" );
    tokenApprovals[tokenId] = spender;
    emit Approval(owner, spender, tokenId);
}

function getApproved(uint256 tokenId) public override
    view returns (address spender) {
    return tokenApprovals[tokenId];
}
```

The `getApproved` function is responsible for querying the `tokenApproval` mapping and returning the address already approved for a token ID. If there is a requirement for multiple approvers of a token ID, an alternative implementation is required.

Before getting into transfer-related functions within the ERC721 interface, it is important to understand another specification known as ERC223.

EIP223

EIP223 and ERC223 are important specifications for detailing certain aspects of contract-to-contract communication. We have already seen in the previous chapters numerous ways of a contract deploying other contracts, using them by invoking functions. This is nothing new conceptually, so how does EIP223 help in contract-to-contract communication?

ERC223 is especially important for contracts that represent tokens – either ERC20 or ERC721. We all know that contracts can accept Ethers and one contract can send Ether to another contract, either with a `payable` function invocation or by transferring them directly using a contract address. If a contract address is used to send Ethers to a contract, a `payable receive` function is invoked automatically that accepts the Ethers. If there is no `payable receive` or `fallback` function, simply sending Ether using the contract address will fail.

Now, imagine the same situation for custom tokens represented by ERC20 and ERC721. ERC20 and ERC721 tokens are generally minted, transferred, and burned for externally owned accounts. They are not meant for contract addresses. Contract addresses should not own these tokens in their current condition, since they would not be able to use or transfer the tokens to any other account. Eventually, these tokens will be lost forever. Therefore, it is necessary that the ERC20 and ERC721 token contracts should identify the current caller address as either a contract account or an externally owned account.

If it finds the current caller address to be a contract account, it declines to mint, transfer, or generate the token. This way, the token contract can ensure that the tokens are not lost forever. However, if that caller's address is a contract and is capable of handling the tokens, then the token contract should be able to transfer, mint, or burn tokens on its behalf. This exact scenario is handled by the ERC223 specification.

The ERC223 specification provides a mechanism to identify an address as a contract address and then allows the transfer of tokens to the contract address, which is capable of accepting the ERC20 and ERC721 tokens. Note that ERC223 does not provide any interface; instead, it provides a mechanism to implement the contract-to-contract communication when tokens are involved.

For demonstrating the implementation of ERC223, we will need a target contract that can receive tokens from the ERC721 contract. This is a simple contract named `ERC223Recipient` with an event and a `tokenReceived` function. This function accepts the `from` argument for the address of the sender (which will be the ERC721 contract address), the `_value` argument contains the token ID, and metadata is contained in the `_data` argument. This function is called when the ERC721 tokens are transferred to this contract, alongside supplied values for its parameters. The function simply emits an event in its current implementation to show that it actually gets invoked; however, it can store the token information in its state variables or take any action it deems necessary.

The `ERC223Recipient` contract implementation is shown next, with both the `Event` and `tokenReceived` functions:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract ERC223Recipient {

    event TokenReceived(address, uint, bytes);
    function tokenReceived(address _from, uint _value, bytes
        memory _data) public
    {
        emit TokenReceived(_from, _value, _data);
    }
}
```

We will have to deploy this contract and note the contract address. We will also have to pass this contract address to the `transfer` function to transfer the ERC721 token to this contract. We will discuss the `transfer` functions later in the chapter. The contract can be deployed using Remix.

We will need a reference to the definition of the `ERC223Recipient` contract so that it can be called from the `ERC721` contract. If both the contracts – that is, `ERC223Recipient` and `ERC721` – are defined in the same file, then no extra step is required to call the `tokenReceived` function. However, if they are in different files, then the file containing the `ERC223Recipient` contract needs to be imported into the `ERC721` contract file, as shown here:

```
import './ERC223Recipient.sol';
```

Now that we understand the concept of the `ERC223` specification and have both the `ERC223Recipient` definition and contract address, let's see how we can use them along with the `transfer` functions for the `ERC721` contract.

There are three variations of `transfer` functions available in the `ERC721` contract, compared to two in the `ERC20` contract. There are two variations (function overloading) of the `safeTransferFrom` function and a simple `transferFrom` function. The `safeTransferFrom` function, as the name suggests, transfers tokens to the destination address. However, it implements the features of `ERC223` and checks whether the destination address is a contract address. If it finds the destination address to be a contract address, it references the existing contract instance using its address and invokes the `tokenReceived` function, passing relevant values to its parameters.

The two overloaded `safeTransferFrom` functions differ from each other based on the number of function arguments. One of the `safeTransferFrom` functions accepts an optional argument, and it can contain any metadata that the `ERC721` contract wants to pass to the destination contract.

Assembly code is used to discover whether the destination address is a contract address. Solidity provides a low-level assembly function, `extcodesize`. It accepts an `address` argument and provides the address length as its `return` value. If the length is greater than 0, it means the supplied address belongs to an address; otherwise, it belongs to an externally owned account. It creates a reference to the existing `contract` instance using the address and invokes the `tokenReceived` function. Eventually, all the `safetransfer` functions internally call the `transferFrom` function.

The implementation for both the overloaded `safeTransfromFrom` functions is shown next. The first function simply delegates the call to the next one with an empty `_data` value:

```
function safeTransferFrom(address from,address to,uint256  
tokenId) external override {  
    safeTransferFrom(from,to,tokenId,"");  
}
```

```
function safeTransferFrom(address from, address to,
    uint256 tokenId, bytes memory data) public override{
    uint length;
    assembly {
        length := extcodesize(to)
    }
    transferFrom(from, to, tokenId);
    if(length > 0) {
        ERC223Recipient(to).tokenReceived(from, tokenId,
            data);
    }
}

function transferFrom(address from,address to,uint256
    tokenId) public override {
    require ( from != address(0) , "from address cannot be
        default null address");
    require ( to != address(0) , "to address cannot be
        default null address");
    require (tokenId > 0, "token ID cannot be zero or
        less!");
    require (ownedTokensCount[from] > 0, "from address
        should own a token!");
    require (tokenOwner[tokenId] == from, "from address
        should own a token!");
    ownedTokensCount[from] = ownedTokensCount[from] - 1;
    ownedTokensCount[to] = ownedTokensCount[to] + 1;

    tokenOwner[tokenId] = address(0);
    tokenApprovals[tokenId] = address(0);
    tokenOwner[tokenId] = to;
    ownedTokens[to] = tokenId;

    emit Transfer(from, to, tokenId);
}
```

The `transferFrom` function is where the magic happens. It is responsible for the transfer of all the tokens. `transferFrom` validates the incoming parameter values and checks whether both the from and to addresses are well formed. It also checks whether the token ID is greater than zero and that the token actually belongs to the owner. Plus, it checks whether the `ownedTokenCount` mapping has information about the owner and the token ID.

After successful validation, the function reduces the count from the sender's balance and increments the count of tokens for the destination address. It nullifies both the `tokenApprovals` and `tokenOwner` mappings for the token ID and then adds a new mapping in the `tokenOwner` and `ownedTokens` maps for the destination address. Finally, the `transfer` event is raised, with values supplied to the function.

ERC721 is dependent on a few other optional standards. Since these are also standards, they are also named based on ERC numbers. One such standard is ERC165, which helps in discovering interfaces implemented within a contract.

ERC165

ERC165 is an optional specification for ERC721 implementation. Its main purpose is to publish the metadata of interfaces and associated functions. This metadata can be used by the caller to determine whether they want to interact with the contract and invoke its functions.

ERC165 provides an interface comprising a single `supportInterface` function. This function accepts `interfaceId` and shows whether or not the contract implements the interface represented by `interfaceId` by returning `true` or `false`.

`interfaceId` comprises 4 bytes of data (`bytes4` is its data type), containing an interface identifier. This `interfaceId` type is the result obtained after performing XOR (exclusive OR) on all its hash function signatures. To understand the process of getting `interfaceId`, first, we take the signature of each function available within an interface. This should just include the name of the function along with the parameter data types (this should not include the name of the parameters and return types).

Generate the hash of the resultant function signature using the `keccak256` hash function. The resultant hash is greater than 4 bytes in length and is trimmed to extract only the first 4 bytes.

This process is executed for all the functions of the interface.

Finally, the resultant value of each interface function is binary-XOR'ed together to obtain the interface ID. For example, the calculation of `interfaceId` of the IERC165 interface is shown in the following code block:

```
bytes4(keccak256('supportsInterface(bytes4)'))
```

When an interface has multiple functions, the interface ID is calculated as shown (note the XOR operator):

```
bytes4(keccak256('function1(uint256)')) ^  
bytes4(keccak256('function2(bytes4, bool)')) ^  
bytes4(keccak256('function3(bytes4, uint256)'))
```

Instead of using all of these steps and the `bytes4` and `keccak256` functions, Solidity provides an easy way to find the interface identifier using the `type` function:

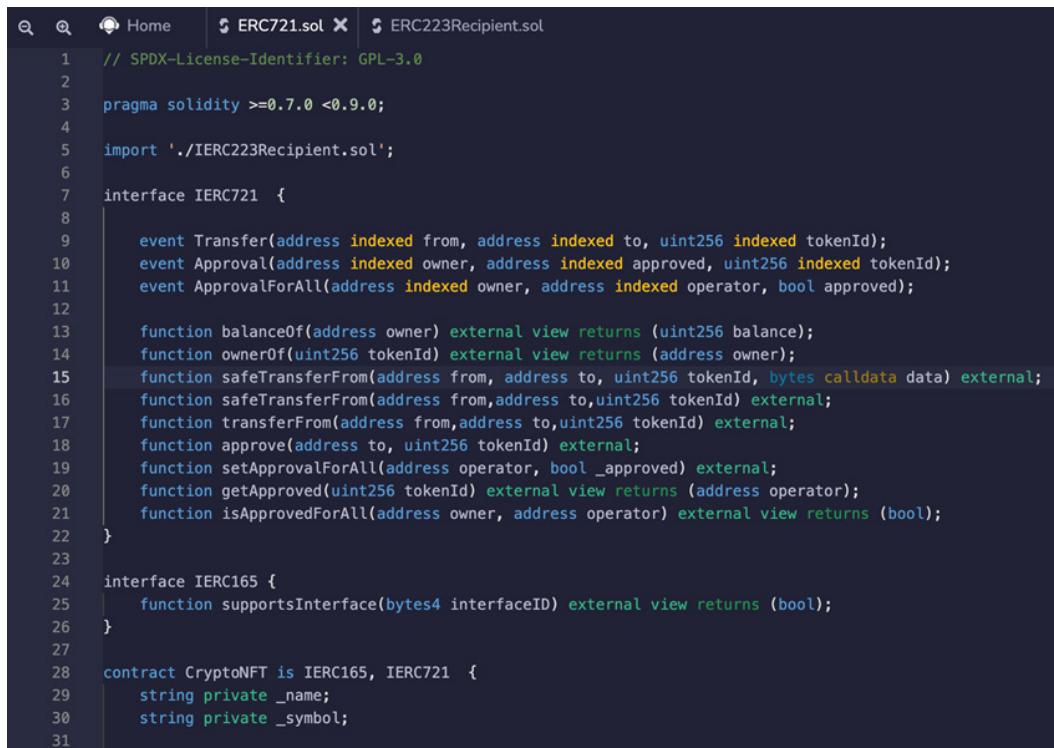
```
interface IERC165 {  
    function supportsInterface(bytes4 interfaceId) external  
        view returns (bool);  
}
```

Instead of using the `bytes4` and `keccak256` functions, Solidity provides an easy way to find the interface identifier using the `type` function. The `type` function has an `interfaceId` getter property that returns the interface identifier for a given interface.

The ERC721 contract should inherit from the IERC165 interface and implement the `supportsInterface` function, as shown in the following code block. Since the ERC721 contract implements both the ERC721 and ERC165 interfaces, the `supportsInterface` function checks whether the incoming `interfaceId` matches either the ERC721 or IERC165 interface ID. Depending on the result of the comparison, success or failure should also be returned from the function:

```
function supportsInterface(bytes4 interfaceId) external  
    override pure returns (bool) {  
  
    return interfaceId == type(IERC721).interfaceId ||  
        interfaceId == type(IERC165).interfaceId;  
}
```

The complete code for the chapter is available with accompanied source code and can be executed directly in the Remix editor, as shown in the following screenshot:

A screenshot of the Remix Ethereum IDE interface. The top navigation bar shows tabs for Home, ERC721.sol (which is the active tab), and ERC223Recipient.sol. The code editor contains two files: ERC721.sol and ERC223Recipient.sol. The ERC721.sol file is the primary contract being edited, while ERC223Recipient.sol is listed as a dependency. The code itself is a Solidity smart contract for a CryptoNFT token, implementing both the IERC721 and IERC165 interfaces. It includes functions for token transfer, approval, and metadata retrieval, along with a constructor to initialize the name and symbol.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
import './IERC223Recipient.sol';
interface IERC721 {
    event Transfer(address indexed from, address indexed to, uint256 indexed tokenId);
    event Approval(address indexed owner, address indexed approved, uint256 indexed tokenId);
    event ApprovalForAll(address indexed owner, address indexed operator, bool approved);
    function balanceOf(address owner) external view returns (uint256 balance);
    function ownerOf(uint256 tokenId) external view returns (address owner);
    function safeTransferFrom(address from, address to, uint256 tokenId, bytes calldata data) external;
    function safeTransferFrom(address from, address to, uint256 tokenId) external;
    function transferFrom(address from, address to, uint256 tokenId) external;
    function approve(address to, uint256 tokenId) external;
    function setApprovalForAll(address operator, bool _approved) external;
    function getApproved(uint256 tokenId) external view returns (address operator);
    function isApprovedForAll(address owner, address operator) external view returns (bool);
}
interface IERC165 {
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
contract CryptoNFT is IERC165, IERC721 {
    string private _name;
    string private _symbol;
```

Figure 14.1 – The ERC721 and ERC223 recipient contracts in Remix

The steps to use the contracts are as follows:

1. Deploy the `ERC223Recipient` contract using Remix. Note the contract address generated by this action. The contract address is needed in later steps.
2. Deploy the `CryptoNFT` contract by passing string values for both the name and the symbol.
3. Mint a few NFTs and then transfer them to the contract address you noted in the first step, using the `safeTransferFrom` function that does not have a `data` parameter.

The result should show two events getting raises – Transfer from the ERC721 contract, and TokenReceived from the ERC223Recipient contract, as shown in the following screenshot:

```
logs [
  {
    "from": "0x38cB7800C3Fddb8dda074C1c650A155154924C73",
    "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
    "event": "Transfer",
    "args": {
      "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
      "1": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBafA1Fd3",
      "2": "2",
      "from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
      "to": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBafA1Fd3",
      "tokenId": "2"
    }
  },
  {
    "from": "0x0498B7c793D7432Cd9dB27fb02fc9cfdBafA1Fd3",
    "topic": "0xeafe605af9663e0f15f1dd40dad79f119df71e8d2affb3f6857cb9707c6c4b3ea",
    "event": "TokenReceived",
    "args": {
      "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
      "1": "2",
      "2": "0x"
    }
  }
]
```

Figure 14.2 – The result showing the event logs after sending the ERC721 token to a contract

ERC721 provides numerous extensions, and this chapter covered the burning of a token as an extension. There are other extensions that are not covered in this book. Those are advanced extensions, such as enumerating tokens, incorporating royalty and metadata, loans, and halting executions, which are all built on the top of the contract that was shown in this section.

This section on ERC721 is the foundation for all NFTs, and all extensions are add-ons. It is necessary to master the basic concepts of ERC721 to get into advanced scenarios. And, as mentioned, these advanced scenarios are just either deviations or additions to an existing contract.

Summary

This chapter was relatively more complex compared to other chapters. This chapter dealt primarily with creating your own cryptocurrencies and NFTs. Cryptocurrencies can be either fungible or non-fungible. This chapter covered both types of tokens by implementing ERC20 and ERC721 tokens.

A complete implementation along with the conceptual understanding of ERC20 was provided in the first half of the chapter, and the latter half was dedicated to the ERC721 implementation.

The ERC721 implementation also has dependencies on other standards such as ERC223 and ERC165. Both these standards were covered, both implementation- and concept-wise. It's now time to move on to learn Solidity design patterns, which will be discussed at length in the next chapter.

Questions

1. What are the two different categories of tokens? What are their corresponding ERC standards?
2. What is the purpose of ERC223?
3. What is the purpose of ERC165?

Further reading

- <https://eips.ethereum.org/> has all the Ethereum improvement proposals along with their corresponding ERCs.
- ERC standards are available at <https://eips.ethereum.org/erc>.

15

Solidity Design Patterns

Solidity is a contract-based language. You might wonder what a language has to do with entity modeling. On the face of it, it does not sound right; however, after security, entity modeling is probably one of the most important activities for writing smart contracts. But why is entity modeling an important exercise for smart contracts? To understand this question, we must understand the purpose of smart contracts. Smart contracts comprise two important facets:

1. Entities
2. Logic

Data stored in smart contracts is stored permanently within Ethereum storage. The use of storage comes with the cost of reading and writing to it. It is for this reason that it is very important to store optimal data that is necessary for the functioning of the smart contract and use case. In this chapter, we will understand how entities should be modeled for smart contracts, the different data types used for storing data, the read and write patterns, the redundancy of data, and relationships between data.

We will cover the following topics in the chapter:

- Understanding data entities in Solidity
- Nested versus reference fields in structs

- Different types of relationships between structs
- Performing CRUD operations in contracts
- Ownership in smart contracts
- Multiownership in contracts
- MultiSig contracts
- Pausable and stoppable smart contracts

This chapter will give you a detailed insight into the topics mentioned and it is advised that you try to execute the associated code in a Remix environment.

Technical requirements

For this chapter, you'll need the following:

- A web browser with an internet connection
- Access to the Remix portal from the web browser

The code files for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Solidity-Programming-Essentials-Second-Edition/tree/main/Chapter15>.

Introducing entity modeling

Entity modeling is not a new concept. It is an age-old practice employed whenever large amounts of data are stored in any system. Entity modeling is used for data stores, such as SQL Server and Oracle. Entity modeling is required for two reasons:

- Increase the performance of an application while reading and writing data to a data store.
- Optimize storage to reduce the cost of storage.

Data stores such as SQL Server or Oracle do not have a cost associated with them while reading and writing data. However, Ethereum has a cost of 5,000 gwei for reading and 20,000 gwei for writing data. There is a recurring cost every time data is accessed.

Entity modeling refers to determining the global storage for smart contracts that serve a particular use case.

Ethereum storage

Ethereum provides multiple types of storage:

- **Global storage:** This is permanent storage in Ethereum and the data is persisted permanently. This data can be read in the future at any point in time.
- **Memory storage:** This is temporary storage and its scope is only within a function. Once the function goes out of the scope, the values stored within these variables are also not available anymore.
- **Calldata:** This is also temporary storage used by Ethereum for storing the function call data. The Ethereum EVM uses it to determine and identify appropriate functions and pass arguments within the function.
- **Stack:** This is again temporary storage that the Ethereum EVM uses to bring data from other storage to work on them. The stack is also available for storing temporary variables within a function.

The primary costs in Ethereum are for global storage and memory storage. However, the cost of global storage is many times the memory storage. Data modeling in Ethereum and Solidity is about the right usage of both memory and global storage.

The cost of global storage is 20,000 wei for writing the first time, 5,000 wei for updating the same storage location, and 200 wei for reading the storage. It is to be noted that these costs are per 32 bytes of storage. For example, reading 64 bytes will cost $2 * 200$ wei, that is, 400 wei.

The cost of memory storage for both reading and writing 32 bytes of data is 2 wei. The cost of memory is way cheaper than global storage.

Can we ignore storing data in global storage? We cannot if we want the data to survive even after the functions go out of scope. If we want data to persist, global storage must be used.

You will now understand that data stored within global storage should be minimal but optimal for the proper functioning of the smart contract. There should not be redundant data stored in global storage, but at the same time, the data should not be so little that it impacts performance and functionality.

Data types in Ethereum

We have already seen, in *Chapter 3, Introducing Solidity*, the native data types available in Solidity, such as bool, int, and bytes.

The data types that help in data modeling are specifically the following:

- Mappings
- Arrays
- Structs

Let's talk about them in detail.

Mappings

Mappings are like hash maps and dictionaries in other programming languages. Mappings comprise two pairs of data – a key and a value. The value can be retrieved using the key and the key should be unique within the mapping. If the key is not unique and data is inserted into the mapping, it would automatically update the existing key-value pair rather than creating a new record with an existing key.

In other programming languages, generally, there are provisions to loop through dictionaries. However, looping and iterating over a mapping is not possible at the time of writing. For example, all employee data is stored in a single mapping; hence, without knowing the key for an employee, it is not possible to extract the value stored within it.

One of the workarounds to these problems is to declare an additional dynamic array storing just the employee identifier. Arrays can be looped, and after looping, appropriate employees can be fetched from the mapping.

It turns out that using mapping is a good choice if the keys are sent as parameters to the function invocation or stored within an array since in either case, looping will not be required to get the values.

It is to be noted that an unlimited amount of data can be stored within a single mapping.

Arrays

Solidity supports both fixed-sized as well as dynamic arrays. To create a generic contract that can store a large amount of data or an unknown number of records, it is ideal to create dynamic arrays. One of the biggest advantages of arrays over mappings is that arrays can be looped over and data can be fetched out of them. It is possible to store structures within an array. However, this is also one of the biggest pain areas of arrays as well. A value from a mapping can be fetched directly without any loops just by using the key. It would return a null value if the key does not already exist within the mapping. However, it will return the appropriate data if the key exists.

If there are multiple requirements for fetching single records, mapping is an ideal candidate since no looping is required. However, if the key is not known, then looping is the only way to fetch the data and arrays would be preferred.

It is important to understand the characteristics of each of the data types before deciding whether they are suitable and should be used for storing data.

We will understand structs in more detail in the next section.

Understanding data modeling in Solidity

Data stored in Ethereum has a fixed schema and layout. Once a smart contract is created with global storage and deployed, it cannot be modified. Its layout is determined, and it remains the same for the lifetime of the contract.

All data required by a decentralized application can be stored within a single smart contract. However, managing such a smart contract can easily become a nightmare and there can be more than one reason for versioning the smart contract and redeploying. A better strategy is to divide all the required data into smaller data buckets and put them in separate smart contracts. This will ensure that even in the case of changes, only part of the application and smart contract will be redeployed instead of the entire application.

However, this approach poses a different set of challenges, such as how should related data be retrieved together from multiple smart contracts, how can we put related data in multiple smart contracts, and what is the performance impact of this?

In this section, we will understand and try to solve the problem of embedding all data within a single smart contract versus multiple smart contracts.

Nested versus reference fields

Before we understand the details of embedding data, let's investigate an example that will help us in understanding embedding data.

Imagine we want to store employee data within smart contracts. The data to be stored is `employeeId`, `firstname`, `lastname`, `emailAddress`, and `phoneNumber`, along with address details. There are multiple addresses for an employee, such as a permanent address, current address, and birth address. Each address, in turn, comprises `streetName`, `City`, `state`, and `zipCode`.

Out of these, `employeeId`, `firstname`, `lastname`, `emailAddress`, `phoneNumber`, and other address details can all be of the `bytes32` or `String` data type.

Since `Employee` is a single entity that should comprise all this data as properties, it is better to think of `employee` as a struct rather than storing all these attributes independently. For example, we can create a smart contract, as shown here:

```
contract Employee {  
    string[] employeeId;  
    string[] firstName;  
    string[] lastName;  
    string[] emailAddress;  
    string[] phoneNumber;  
}
```

The previous example is a valid approach to store the data for an employee. However, this approach has a few problems. To start with, this approach does not relate the variables together as an employee. There is no relationship between these different variables in a contract. Each one can evolve independently whereas they belong to a single employee entity.

In such scenarios, it is better to convert such data into a new employee data type. Structs help in creating a new data type and encapsulating multiple properties.

Since we are storing data for multiple employees, we have the choice of storing the structs in a mapping, as shown here:

```
struct Employee {  
    string employeeId;  
    string firstName;  
    string lastName;  
    string emailAddress;  
    string phoneNumber;  
  
}  
mapping (string => Employee) students;
```

Or we can store them as an array, as shown here:

```
Employee[] employees;
```

Now that we have set up the context with regard to the employee, it will help in discussing the embedded nature of modeling.

We have created the employee structure; however, we are yet to include the data type for the address of an employee. The address type would look as shown here:

```
struct Address {  
    string streetName;  
    string city;  
    string state;  
}
```

Now that we have both the structures, it's time to understand how we can bring them together such that it is easy to both read and write data while maintaining adequate performance.

The first way to model is to use the process of embedding one structure in another, as shown:

```
struct employee {  
    string employeeId;  
    string firstName;  
    string lastName;  
    string emailAddress;  
    string phoneNumber;  
    Address employeeAddress;  
}
```

In this approach, the employee address is embedded within the `employee` structure. It becomes an integral part of the `employee` structure. So, whenever employee records are read or written, the employee address is also read and written.

The next approach is by way of using references. In this approach, the address struct does not become part of the employee structure, but rather stays independent. It requires small changes to the structure:

The `employee` structure is shown next:

```
struct employee {  
    string employeeId;  
    string firstName;  
    string lastName;
```

```
        string emailAddress;
        string phoneNumber;
        int256 addressReference;
    }
```

The `Address` structure is shown next:

```
struct Address {
    int256 addressID;
    string streetName;
    string city;
    string state;
    string employeeId;
}
```

In this approach, a reference is maintained between both structures. An address for an employee can be reached using `addressID`, stored within the `employee` structure and vice versa.

The main points of difference between the two approaches are that address and employee structs do not have additional redundant attributes, such as maintaining the `addressID` and `employeeId` fields in the `Address` structure (although the `employeeId` field is completely optional). In the case of references, the `Address` structure can be placed within another smart contract, which can help with better maintenance, and it does not need to be updated if the employee's contract is updated.

It is important to realize that representing business entities as structures is an important and best practice for smart contracts. The structures could include other structures as nested structures or just references to other structures using an identifier. We will look into different ways to model entities along with their relationships in the next section.

Exploring types of relationships

In terms of references between two structures, there are three different possibilities, which we will explore in the next sections.

One-to-one relationships

In a one-to-one relationship, two structures are related to each other, with each having a single entity, and they are related using an identifier. The related instance contains a single record of data for each main structure. It is to be noted that the relationship can be navigated from both the related structure to the main structure and the main structure to the related structure. However, it is mandatory to have a relationship from the main to the related structure. The following code block shows that there is a main structure and a related structure. Both structures have a common property, `familyId`, through which they are related and interconnected:

```
struct Main {
    address customerEtherAddress;
    uint256 familyId;
}

struct Related {
    uint256 familyId;
    string[] childNames;
}
```

One-to-many relationships

In one-to-many relationships, the related structure contains multiple records of data for each main structure. It is to be noted that the relationship can be navigated from both the related structure to the main structure and the main structure to the related structure. However, it is mandatory to have a relationship from the main to the related structure. Note that in the following example, there is a main structure and a related structure. Both structures have a common property, `familyId`, through which they are related and interconnected. However, notice that `familyId` is a dynamic array in the main structure, whereas it is of the `uint256` type in the related contract. This means that the main structure can be related to multiple related structures – not just one:

```
struct Main {
    address customerEtherAddress;
    uint256[] familyId;
}

struct Related {
    uint256 familyId;
    string[] childNames;
}
```

Many-to-many relationships

In many-to-many relationships, the related structure contains multiple records of data for each main structure and the main structure contains multiple records of data for each related structure. It is mandatory to have a relationship from the main to the related structure and vice versa. Note that in the following example, `familyId` is a dynamic array that can contain multiple data items, each one referring to a unique record in the related structure, and the related structure contains a dynamic address array, each one referring to a record in the main structure:

```
struct Main {  
    address customerEtherAddress;  
    uint256[] familyId;  
}  
struct Related {  
    uint256 familyId;  
    string[] childNames;  
    address[] mainAddresses  
}
```

Now that we understand different ways to model and relate entities, there are a few best practices that govern whether embedding other structs is better than designing and referencing structures. In the next section, we are going to review them.

Reviewing the rules for embedding structures

By now, we understand both references and nested/embedded structures at a conceptual level. It is important to also know the rules for using nested structures compared to references. It is to be noted that these rules represent best practices and can change depending on the context:

- The embedded structure is not going to change anytime in the near future.
- There is a containment relationship between structures.
- There are one-to-few relationships between structures.
- The data that is embedded does not change frequently.
- The embedded data is needed frequently along with the parent structure.
- The embedded data will not grow out of bounds.

Let's understand each of these rules in detail.

Data cohesion

If the data from multiple structures is read together, it is wise to put it together as well such that it can be read in a single step. Otherwise, the application must invoke multiple calls to different contracts to get the data, as is generally the case with reference data.

Out-of-bounds nested structure

If a nested structure grows unlimited in number, then reading a single outer structure will lead to reading all its related nested data stored in the nested structure. This can lead to reading quite a lot of data that might not be needed at all. In such circumstances, the gas consumed will be very high and it might even lead to an out-of-gas exception. Moreover, there might be times when data in a nested structure is so large that it might require more than 8 million gas, which is the gas limit of a block.

Static data within a nested structure

If the data of a nested structure is non-volatile, that is, it changes frequently, it makes sense to load this data and write it back along with the container structure data in one step. This requires that the data is stored along with the container structure. If data does not change frequently for both the container as well as the nested structure, it is better to use references since only data that changes can be loaded and updated, rather than the whole big structure where most of the data remains unchanged.

The nested structure will not change in the near future

This rule means that the change in the nested structure should not be the reason to deploy the container structure containing the nested structure. If it is envisioned that the nested structure will evolve frequently in the future, it is wise to put it into another structure and contract such that the container structure continues to work as is without any disruption in the event of changes within a nested structure.

Containment relationship

A containment relationship is established when there is a parent-child relationship or when the contained structure cannot exist without the container structure; for example, the sales order item structure is a child of the Sales Order structure. The Sales Order item cannot exist without having a Sales Order.

Having few relationships

If we have an embedded or nested structure, it means there is a relationship between the two structures. If the relationship between the structures is one-to-one or one-to-few, then the contained structure can be part of the overall container structure. However, if there are one-to-many relationships, it is wiser to put them into a separate structure and storage as well.

With an understanding of these rules, let's go on to further learn about entity modeling using an example.

Performing data modeling using an example

The previous discussion should ignite your minds with thoughts that data modeling is an extremely important exercise for smart contracts and using the right data types for storing the values is equally important.

Let's understand the entire process using an example and assume that we again want to store employee and address information as part of a decentralized application.

Structures

The structure for `Employee` is quite simple and it has been purposely kept so. You can add as many elements to it as you deem necessary:

```
struct Employee {  
    address identifier,  
    bytes32 name,  
    uint8 age,  
    bytes32 email  
}
```

An `Address` struct is shown next:

```
struct ContactAddress {  
    string city;  
    string state;  
}
```

The `employee` struct in this example embeds the address structure as a nested element:

```
struct employee {
    address identifier;
    bytes32 name;
    uint8 age;
    bytes32 email;
    ContactAddress contact;
}
```

State variables

We will also declare variables that will hold the values of these structs within a mapping and a reference to all the employees as a counter within an array, as shown:

```
mapping (address => employee) allEmployees;
address[] employeeReference;
```

Adding Employees

There should be a function to add an employee to state variables, as shown:

```
function AddEmployee(address _identifier, string memory _name,
    uint8 _age, string memory _email, string memory _state,
    string memory _city) external returns (bool) {
    ContactAddress memory contactadd = ContactAddress(_city,
    _state);
    allemployees[_identifier].identifier = _identifier;
    allemployees[_identifier].name = _name;
    allemployees[_identifier].age = _age;
    allemployees[_identifier].email = _email;
    allemployees[_identifier].contact = contactadd;
    employeeReference.push(_identifier); }
```

Retrieving a single Employees record

The next function returns a single employee to the caller. In this case, the caller will send the identifier information and that should be enough to fetch the information from the mapping. Additional validation about whether the identifier exists within the array or not can also be written:

```
function GetAnEmployee(address _identifier) external returns
    (string memory _name, uint8 _age, string memory _email,
     string memory _city, string memory _state) {

    employee memory temp = allemployees[_identifier];
    _name = temp.name;
    _age = temp.age;
    _email = temp.email;
    ContactAddress memory addr = temp.contact;
    _city = addr.city;
    _state = addr.state;
}
```

Updating Employees

Updating employee data is very similar to adding a new record, as shown here:

```
function UpdateEmployee(address _identifier, string memory _name,
    uint8 _age, string memory _email, string memory _state,
    string memory _city) external returns (bool) {
    ContactAddress memory contactadd = ContactAddress(
        _city, _state);
    allemployees[_identifier].identifier = _identifier;
    allemployees[_identifier].name = _name;
    allemployees[_identifier].age = _age;
    allemployees[_identifier].email = _email;
    allemployees[_identifier].contact = contactadd;
```

Retrieving all Employees

Retrieving all the records can pose a challenge due to limited gas availability. In such circumstances, the smart contract can use paging and return only the first 100 records and if the user needs more data, then the smart contract will send data related to the next 100 records:

```
function GetAllEmployee(uint startRecord, uint endrecord)
    external returns ( string[] memory, uint8[] memory,
        string[] memory, address[] memory , string[] memory ,
        string[] memory)
    {
        uint8[] memory _age = new uint8[]
            (employeeReference.length);
        string[] memory _name = new string[]
            (employeeReference.length);
        string[] memory _email = new string[]
            (employeeReference.length);
        address[] memory _identifier = new address[]
            (employeeReference.length);
        string[] memory _state = new string[]
            (employeeReference.length);
        string[] memory _city = new string[]
            (employeeReference.length);
        for(uint i= startRecord; i <= endrecord; i++) {
            address addressinArray = employeeReference[i-1];
            _age[i-1] = allemployees[addressinArray].age;
            _name[i-1] = allemployees[addressinArray].name;
            _email[i-1] = allemployees[addressinArray].email;
            _identifier[i-1] = allemployees[addressinArray]
                (10) .identifier;
            _state[i-1] = allemployees[addressinArray].contact
                .state;
            _city[i-1] = allemployees[addressinArray].contact
                .city;
        }
    }
```

```
        return ( _name, _age, _email , _identifier , _state ,
        _city);
    }
```

After implementing all functions related to reading, updating, and adding new records using smart contracts, it's time to get into the details of other Solidity patterns, starting with ownership-based contracts, which we will discuss in the next section.

Ownership in smart contracts

Ownership is an important concept in smart contracts. Smart contracts are custodians of digital assets. These assets are of value and can be traded on exchanges as well. Smart contracts are capable of transferring these assets between addresses. It is also possible to have certain logic within smart contracts where only privileged users should be able to execute it. These functions are not meant for all other users of smart contracts. Again, instead of one, there could be multiple owners of a contract, also known as **multisignature (MultiSig)** contracts. Approvals or signatures of all the owners within a MultiSig contract are required to execute the privileged function. Now, we will explore ownership-related patterns within Solidity.

Exploring ownership in Solidity

Solidity has some unique features that are not available in other programming languages, and it requires us to rethink it from a design patterns perspective. Solidity provides unique constructs, such as modifiers, which have a different way of handling and throwing exceptions, and has unique concepts, such as gas, Ether, consumption of gas, payable functions, and fallback functions.

Solidity was envisioned to provide a means to write contracts, although writing a contract looks very similar to writing an object-oriented class along with its methods and variables. In essence, contracts are a completely different genre and provide implementations that are unique to contract-based languages.

In this chapter, we will investigate design patterns that do not demand a complete chapter on their own but are very important from the usage and implementation perspectives. We will first understand what ownership is, and then cover the following:

- Ownable pattern
- Single-owner ownable pattern
- Transfer of ownership pattern

- Multiowner ownable pattern
- MultiSig pattern
- Pause or haltable pattern

One of the premises of blockchain, such as Bitcoin and Ethereum, is to establish ownership. The ownership of Bitcoin and Ether is built using the concept of an individually owned account backed up by a pair of cryptographic keys. These keys are also known as public and private keys. The holder of the private key can unlock their account and transfer the ownership of their Bitcoin/Ether to another address.

Ethereum is a functional blockchain that helps in extending it by writing smart contracts. The smart contracts consist of custom logic that can be executed on the Ethereum EVM, and the state is stored as a world state across all nodes on the network. With the help of smart contracts, it is possible to write functional programs that can store any digital asset on Ethereum. These digital assets could include real estate property, land, equities, stocks, and anything else that has intrinsic value.

If these digital assets can be stored as a smart contract state, then it should also be possible to ascertain the owner of these assets and the ownership of these assets can only be transferred to another account by the current owner.

Another way of thinking of ownership with regard to smart contracts is that a smart contract can be owned by someone. This means that the existence and working of a smart contract can be influenced by an owner.

There can be a single owner or multiple owners of a smart contract. Moreover, there could be operations within the smart contract that only privileged accounts can execute. For example, the transfer of an asset can only be executed by the owner of the contract. If you are not the owner or one of the owners of the contract, you would not be able to transfer an asset.

In the next section, we will investigate patterns that provide us with a solution for both of the following kinds of ownership transfer:

- Transfer of ownership of a digital asset by only its owner
- Ownership of a contract and privileged execution of its methods

A large part of ownership patterns utilizes the "modifier" feature of smart contracts. Although not mandatory, modifiers help in writing more readable and manageable smart contracts. They help in abstracting certain behaviors of smart contracts into separate functions that can then be applied on a when-needed basis to multiple functions across the inheritance chain of contracts.

Modifier

A modifier is declared the same way as a Solidity function within a smart contract. A simple modifier can be declared, as shown next. This modifier within a smart contract is declared using the `modifier` keyword followed by its name:

```
modifier onlyOwner {  
}  
}
```

A modifier can contain code just like any other Solidity function:

```
modifier onlyOwner {  
    require(msg.value == 1 ether);  
}
```

In the previous code, there is a single line of Solidity code that checks, using the `require` keyword, whether exactly 1 Ether is sent while calling a function within the contract. We have not yet associated this modifier with any function; we will be doing so soon.

A modifier can be associated with a function within a smart contract by using its name within the function declaration, as shown:

```
function Deposit() public onlyOwner {  
}
```

In this code listing, notice how the `onlyOwner` modifier has been tagged within the function declaration. A function can have as many modifiers associated with it as it deems necessary.

The previously shown modifier is not of much use since the only thing it does is to check the amount of Ether sent along with the call to the `deposit` function. In this case, the `deposit` function will be called, and before any line of code within this function is executed, the modifier code is executed. The modifier code checks the amount of Ether sent and stops there. There is no way the execution returns to the called function from the modifier. To ensure that the code within the function can execute after the modifier code has finished executing, the modifier code should contain the `_` keyword, as shown next:

```
modifier onlyowner {  
    require(msg.value == 1 ether);  
_  
}
```

The underscore keyword is the modifier's way of saying to replace the function code with itself. In this way, the modifier ensures that the function code can be executed after its execution.

A modifier can take parameters as well, as shown next:

```
modifier onlyowner (uint minAmount) {  
    require(msg.value > minAmount);  
}
```

A function can also be associated with a parameter-based modifier, as shown next:

```
function Deposit() public onlyowner(2 ether) {  
}
```

Now that we have established a basic understanding of modifiers, we can move on to smart contracts and start exploring ownership-based patterns.

Establishing ownership of a smart contract

A smart contract can be owned and controlled by an externally owned or another contract account.

Sometimes, a smart contract needs to be owned by someone. For example, if you are deploying a smart contract to store your employee details, it should be possible for you to manage some of its operations exclusively. Moreover, you should have control over the lifetime of the smart contract. There are many other use cases, such as in the case of **Initial Coin Offerings (ICOs)**, **Decentralized Autonomous Organizations (DAOs)**, and other industry-related use cases.

The next example shows one of the ways to own a smart contract. It is named `ownable` and contains a single state variable, `owner`, of the address type. There is a constructor that gets executed at the time of deployment of the contract. Within the constructor, the externally owned address deploying the contract is captured using the `msg.sender` global property and is assigned to the state variable. Furthermore, a modifier named `onlyOwner` is defined, which checks whether the current caller address is equal to the already-stored address within the state variable. If they are the same, then the modifier allows the execution of subsequent code; otherwise, an exception is raised and the entire state is rolled back to the previous state with unutilized gas returned to the caller.

This contract is then inherited by another contract, called `myICO`. The important thing within this contract is that it defines a `Withdraw` function with the `onlyOwner` modifier associated with it. What it means is that the address that deployed the contract can only invoke this function and execute the code within it. Other externally owned accounts or contract accounts will not be able to execute this function from this contract. Now, there could be multiple such functions declared within the contract that are associated with the same modifier or other modifiers.

This pattern helps in ensuring that certain functions can be called by certain individuals only:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ownable {
    address owner;
    constructor() public {
        owner = msg.sender;
    }

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

contract myICO is onlyOwner{
    uint8 _amount;

    function Withdraw(uint8 amount) onlyowner public {
        _amount = amount;
    }
}
```

The `ownable` and `myICO` contract is one of the ways through which a certain function can only be invoked by a single address. Next, we will see an implementation related to multiple owners instead of just a single owner.

Multiownership

The previous pattern shows the way an individual contract is owned by a single externally owned account. In the next couple of patterns, we are going to see an implementation in which there are multiple owners for a single smart contract.

There are situations in which multiple people might own an asset jointly. In such cases, there are two possibilities:

- A function call can be executed from any of the existing owners. There is no need for all owners to allow the execution of the function.
- A transaction can only be executed after all owners agree to such execution. This is also known as a MultiSig contract.

Let's see how to establish this.

Establishing multiownership of a smart contract

It is also possible that a single smart contract is owned by multiple individuals. This means that there are multiple owners of the contract and each of them has the right to act on the contract.

The contract shown next is named `MultiOwnable` and contains multiple state variables. The `owner` state variable, of the address type, is used to track the original address responsible for the deployment of the contract. The other two state variables, `isOwner` and `Owners`, are used for tracking multiple owners of the smart contract. The `Owners` state variable of a dynamic array is of the address type and holds all addresses that are owners for the contract. The `isOwner` state variable is an optional variable and is of the mapping type. The key for the mapping is an address and the value is of the Boolean type, denoting whether the address is an owner.

There is a constructor that gets executed at the time of deploying the contract. Within the constructor, the externally owned address deploying the contract is captured using the `msg.sender` global property and assigned to the `owner` state variable. The same account is added to the mapping state variable with `true` as its value. The account is also added to the address array state variable using the `push` array method.

Furthermore, a modifier named `onlyOwner` is defined, which checks whether the current caller address is equal to the already-stored address within the state variable. If they are the same, the modifier allows the execution of subsequent code; otherwise, an exception is raised and the entire state is rolled back to the previous state with unutilized gas returned to the caller.

An additional function, `addOwners`, is added to the contract that helps in adding new owners to the existing list of owners. This function can only be invoked by the original account that deployed the contract and it accepts an address parameter. This parameter refers to the account address that should be added as an additional owner of the contract. Again, the account is added to the mapping state variable with `true` as its value and pushed to the address array state variable.

An additional modifier, `checkOwners`, is also added that checks for all the owners that were added using the `AddOwner` function. It checks the existence of an address within the mapping having a value of `true`. If it does have a `true` value, then it allows the execution of the function with which it is associated.

This contract is then inherited by another contract, called `myICO`. The important thing within this contract is that it defines a `Withdraw` function with the `checkOwners` modifier associated with it. What this means is that only the address that is available with a value of `true` with the mapping can invoke this function and execute the code within it. Other externally owned accounts or contract accounts will not be able to execute this function from this contract.

This pattern helps in ensuring that certain functions can be called by a few individuals only:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MultiOwnable {
    mapping (address => bool) isOwner;
    address owner;
    address[] owners;

    constructor() public {
        owners.push(msg.sender);
        isOwner[msg.sender] = true;
        owner = msg.sender;
    }

    function addowners(address additionalAddresses) public
        onlyOwner {
        owners.push(additionalAddresses);
```

```
        isOwner[additionalAddresses] = true;
    }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    modifier checkOwners {
        require(isOwner[msg.sender] == true);
        _;
    }
}

contract MyICO is multiOwnable {
    uint8 _amount;
    function Withdraw(uint8 amount) checkOwners public {
        _amount = amount;
    }
}
```

Establishing ownership and proving ownership are the central tenets of smart contracts. Smart contracts are custodians of digital assets and there are certain operations that only the owner of the smart contract should be able to execute. Ownership patterns help in doing so. In the single ownership pattern, there is a single owner; however, how can that owner transfer ownership to some other address? The next section will address this.

Transfer of ownership

Now that we have seen establishing ownership of a contract to an individual as well as a group of addresses, it should also be possible to transfer ownership from one address to another.

The transfer of ownership of digital assets is one of the main propellants of functional blockchain such as Ethereum. In the next pattern, we will write a smart contract that allows the transfer of contract ownership from one account to another.

The next example uses the single ownership example again. However, the multiple-owners example could easily be written as well. The next contract is still named `ownable` and contains a single state variable, `owner`, within it. Most of the contract is the same as what we wrote for the first pattern in this chapter. There is, however, a small, additional function added to the contract. The `transferOwnership` function is added to the contract and accepts a parameter of the address type. This address reflects the new address that should become the owner of the contract. It can only be invoked by the existing owner due to the association with the `onlyowner` modifier. The only thing this contract does is modifies the owner state variable with the new incoming address value. From this point on, the old address will not be able to invoke any function with the `onlyowner` modifier associated with them. These functions can only be invoked by the new address that was supplied to the `transferownership` function as a parameter:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract ownable {
    address owner;
    constructor() public {
        owner = msg.sender;
    }

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

    function transferOwnership(address newOwner) onlyOwner
public {
        owner = newOwner;
    }
}

contract MyICO is ownable {
    uint8 _amount;
    function Withdraw(uint8 amount) onlyowner public {
        _amount = amount;
    }
}
```

```
    }  
}
```

This concludes the details of writing single owner-based smart contracts. While a contract can be owned at the time of deployment itself using the constructor function, the ownership can be transferred by the original owner to another address. This transfer of ownership can only be done by the existing owner. Now it's time to understand MultiSig contracts.

MultiSig contracts

There are times when multiple accounts should all approve or provide a positive vote for executing a transaction. This can be achieved using the MultiSig pattern.

MultiSig means that there should be multiple approvals or votes or a signature already available before a transaction can be executed. This is a very common requirement in multiple businesses where multiple directors, including the finance department and the CEO, should sign on documents before they can be termed as legal. The MultiSig pattern helps in solving the same problem on an Ethereum blockchain using Solidity.

In the next contract, there are two static accounts referred for MultiSig. There can be as many accounts as needed for MultiSig. Moreover, it need not be static but a dynamic set of accounts that can be used for MultiSig. In short, there are multiple different ways MultiSig contracts can be implemented.

There is a `vote` function defined that each signatory or voter should call to provide their consent. Even if just one of the voters does not provide consent, the actual transaction will not be executed. The `vote` function ensures that it adds the vote to the mapping with a value of `true`.

There is also a `VoteStatus` function that checks whether every voter has provided consent and executes its logic if it finds all signatures have provided a positive vote. Finally, the notes are again turned back to `false` for the next set of voting and function executions to take place:

```
// SPDX-License-Identifier: GPL-3.0  
pragma solidity >=0.7.0 <0.9.0;  
  
contract MultiSigContract {  
  
    address ownerOne;  
    address ownerTwo;
```

```
mapping(address => bool) vote;

constructor() public {
    ownerOne = 0aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa;
    ownerTwo = 0xbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaa;
}

function Vote() public {
    require (msg.sender == ownerOne || msg.sender == ownerTwo);
    require (vote[msg.sender] == false);
    vote[msg.sender] = true;
}

function VoteStatus() public returns (string) {
    require (vote[ownerOne] == true && vote[ownerTwo] == true);
    //execute your logic
    vote[ownerOne] = false;
    vote[ownerTwo] = false;
}
}
```

Ownership in contracts is an important concept and we have seen different ways to establish ownership in smart contracts – single owner, multiple owners, and MultiSig contracts. Now it's time to learn how to transfer ownership from one address to another in smart contracts.

Transfer of the ownership of assets within a smart contract

So far, we have looked into patterns that solve the problem of contract ownership. The next pattern that we will go through is about establishing ownership of a digital asset and transferring these assets from one account to another.

This pattern is again heavily used for creating ERC20 tokens in Ethereum. The ERC20 specification also talks about `transfer` and `transferFrom` functions.

In this pattern, state variables are used to store the ownership details of the digital asset and the functions within the smart contract are responsible for transferring the ownership from one account to another.

The smart contract used for implementing this pattern is called `AssetBank` and is derived from an ownable smart contract. A structure is declared within the smart contract named `assetHolders`. It comprises two fields: `assetValue` and `hasAsset`. The `assetValue` field is used to store the value of an asset and `hasAsset` refers to whether there is any value left or stored for an account.

Another state variable of the mapping type is declared. It has an address type as its key and the `assetHolder` structure as its value. This mapping is used to store the ownership details. An address is used as an identifier and the corresponding structure is filled up with appropriate values to represent as a value.

Furthermore, a modifier named `legalOwner` is defined, which checks whether the current caller address is already available within the mapping and that there is an asset available against this address. If the value is `true`, then it will allow the transfer of an asset by allowing the execution of subsequent code; otherwise, an exception is raised, the entire state is rolled back to the previous state, and unutilized gas is returned to the caller.

Finally, the function that transfers the ownership is implemented. It is named `transferAsset` and has the `legalOwner` modifier associated with it. It executes a series of validations before changing the balances of the two accounts:

```
contract AssetBank is ownable {  
    struct assetHolders {  
        bool hasAsset;  
        uint256 assetValue;  
    }  
  
    mapping (address => assetHolders) balances;  
    constructor() public {}  
  
    modifier legalOwner {  
        require(balances[msg.sender].hasAsset == true);  
        _;  
    }  
  
    function transferAsset(address toAddress, uint256 amount)  
        external  
        legalOwner returns (bool){  
        require(toAddress != address(0));  
    }
```

```
require(amount > 0 );
require(balances[msg.sender].assetValue - amount <
amount );
require(balances[toAddress].assetValue + amount >
amount );
balances[msg.sender].assetValue = balances[msg.sender]
.assetValue - amount;
balances[toAddress].assetValue = balances[toAddress]
.assetValue + amount;
if (balances[msg.sender].assetValue <= 0) {
    balances[msg.sender].hasAsset = false;
}
}
```

In addition to the ownership-based pattern, there are other patterns that help us stop executing functions within a contract during planned and unplanned events.

Stoppable/haltable smart contract pattern

The next pattern we will discuss is used heavily with smart contracts related to ICOs. We all know that smart contracts deployed on public Ethereum can be accessed by anyone. Thus, these contracts can be subjected to multiple types of hacking attacks. There have been several occasions, either because of the shortcomings of a smart contract or the hacker being able to find vulnerabilities, where smart contracts have been hacked, their balances siphoned off, and even the ownership being changed to some other externally owned account. We also know that smart contracts are immutable and there is no way they can be deactivated or activated at will. At such times, it is important that the smart contract execution can be stopped to minimize losses. The pattern we will discuss next is known as the haltable, stoppable, or pausable pattern.

This pattern stops the execution of functions within a smart contract and can resume the execution after the owner deems it fit to run those functions again. This pattern also uses the concept of modifiers heavily.

The contract shown next is named `Stoppable` and contains a single state variable, `isStopped`, of the Boolean type. It derives from the ownable smart contract and thereby can utilize the state variable and modifiers declared within the ownable smart contract. There is a constructor that gets executed at the time of deployment of the contract. Within the constructor, the value of the `isStopped` state variable is assigned `false`. False means that the functions within the contract should not pause and should continue to execute. Furthermore, a modifier named `shouldBeStopped` is defined, which checks whether the current value is stored in the state variable and compares whether it is `false`, in which case it lets the function execution continue; otherwise, it stops the execution.

There are an additional couple of functions, `stopActivities` and `startActivities`, defined within the contract. The purpose of these functions is to toggle the `isStopped` state variable. Also note that these functions can only be called by the owner of the contract, which we know is someone who has deployed the contract.

The contract also defines a `Withdraw` function with the `shouldBeStopped` modifier associated with it. This means that every time this function is invoked, the modifier checks whether the value of the state variable is `true` or `false`. If the value is `false`, it lets the function continue its execution; otherwise, it stops it.

Using this pattern, account owners can stop the execution of their contract functions on demand and ensure that there are minimized losses in the event of an untoward happening:

```
contract Stoppable is ownable {
    bool isStopped;

    constructor() public payable {
        isStopped = false;
    }

    function stopActivities() external onlyowner {
        isStopped = true;
    }

    function startActivities() external onlyowner {
        isStopped = false;
    }

    modifier shouldBeStopped {
        require(isStopped == false);
    }
}
```

```
    _;
}

function withdraw(uint256 amount) external shouldBeStopped
returns
(bool) {
//msg.sender.transfer(1 ether);
return true;
}

}
```

This concludes the implementation of a stoppable smart contract using Solidity. A stoppable smart contract is an important part of smart contract implementation as it helps in freezing the smart contract due to unforeseen circumstances.

Summary

The focus of this chapter was to understand data modeling with regard to smart contracts. Smart contracts contain state variables, and they are costly in terms of their usage. This chapter introduced concepts related to nested or embedded structures viz-a-viz reference structures and concepts related to the use of references versus nested structures. However, modeling structures and storing them in mappings viz-a-viz arrays can make big differences in terms of access and write patterns. Looping plays a big role in deciding whether arrays or mappings should be used.

We also saw a complete example of a smart contract with multiple structures used in various operations, such as reading, writing, and looping. In this chapter, two other important patterns were discussed. The first set of patterns was related to establishing and transferring ownership of both assets and contracts, and the next pattern was to stop executing functions in a contract in the event of an emergency. These are very important patterns for any contract that participates in the transfer of assets, including Ether, digital assets, and payments in other forms. To understand these patterns better, it is important to understand the workings of modifiers because they help in implementing these patterns easier.

Almost every smart contract has storage data and entity modeling as an important concept for writing efficient and optimized smart contracts from a cost perspective. Entity modeling, along with its various approaches, was explained in depth in this chapter. Contracts store and transfer the ownership of digital assets and actions can, at times, be performed only by privileged users. Introducing these privileged users by means of ownership patterns is an essential concept and there are multiple ways in which the ownership of assets and contracts can be established, such as MultiSig and multiple-owner smart contracts. Finally, contracts that can be stopped or halted due to unforeseen circumstances were also covered in depth in this chapter.

This was the last chapter of the book and one of the most important chapters, detailing some key Solidity-related patterns that should be implemented within smart contracts. Solidity is gaining a lot of traction in the market and writing quality smart contracts is of paramount importance due to its nature of holding, storing, and transferring digital assets. You should be able to benefit from this book by learning how to write quality smart contracts for your use cases. This book covered important concepts related to blockchain, specific to Ethereum, introduced the Solidity programming language and its data types, variables, memory locations, global variables and functions, and expressions. It looked in depth at the language for writing smart contracts, inheritance, exception handling, testing, and deploying them. In the latter part of the book, important patterns related to upgradeable smart contracts, ownership, stoppable and haltable, and entity modeling were covered. Some very specific usage examples of smart contracts in terms of writing ERC20 and NFT using ERC721 were also covered in the book. The important concept of writing assembly language within a smart contract was also covered.

Questions

1. What kind of relationship between structs, implemented in the child structs, can have an unlimited number of instances?
2. What feature of Solidity helps in implementing the ownership of contracts?

Further reading

To learn more about common patterns in Solidity, visit here: <https://docs.soliditylang.org/en/latest/common-patterns.html>

Assessments

In this section, you will find answers to questions from the chapters.

Chapter 1, Introduction to Blockchain, Ethereum, and Smart Contracts

1. Proof of work and proof of stake
2. ABI and bytecode

Chapter 2, Installing Ethereum and Solidity

1. Miner.setEtherbase().
2. Personal.newAccount().
3. MetaMask is a software wallet to store Ether as well as any ERC20-based token.
4. ganache-cli is a lightweight Ethereum implementation typically used for DevTest purposes instead of for use on the actual test or main network.

Chapter 3, Introducing Solidity

1. mapping
2. No
3. Arrays, structs, and mappings

Chapter 4, Global Variables and Functions

1. Tx.origin
2. Private, internal, and public

Chapter 5, Expressions and Control Structures

1. The `for` loop
2. Either the `do.. while` or `while` loop
3. `break` and `continue`

Chapter 6, Writing Smart Contracts

1. Using the new keyword or the address of an existing deployed contract.
2. Yes, it is possible to do so.
3. Both abstract contracts and interfaces help in achieving abstraction. However, the main difference between them is that interfaces do not have any implementations at all, whereas abstract contracts can have functions with default implementations.

Chapter 7, Functions, Modifiers, and Fallbacks

1. There are four types of scope visibility modifiers available in Solidity – private, internal, public, and external. Also, state variables cannot have external scope visibility.
2. The `fallback` and `receive` functions are special Solidity functions that are executed automatically, based on certain conditions. A `fallback` function is executed if a function call on a contract is made and such a function does not exist, whereas a `receive` function is executed when any value in terms of Ether, wei, or gwei is sent to a contract without any function invocation – for example, using the `send` or `transfer` functions.

Chapter 8, Exceptions, Events, and Logging

1. `REVERT`, `REQUIRE`, and `ASSERT`
2. `EMIT`

Chapter 9, Truffle Basics and Unit Testing

1. `Init` is used for initializing a `truffle` project, `Migrate` is used to build and deploy contracts on the target network, `test` is used to execute unit tests within a `truffle` project, and `console` is used to interactively work with the network using `truffle`.

2. By modifying the `truffle.json` file and adding network details in the `json` format:

```
networks: {
    development: {
        host: "127.0.0.1",
        port: 8545,
        network_id: "*" // Match any network id
    }
}
```

Chapter 10, Debugging Contracts

1. Debugging of smart contracts using the Remix editor can be done in two ways – by clicking on the **Debug** button after executing a transaction, and by copying the hash of the transaction and pasting it in the **Debugger** tab to start debugging.
2. **Stack, Memory, Call Stack, Return Value, Global Variable, Storage, Solidity Stack, Locals, and State.**

Chapter 11, Assembly Programming

1. `sload` for reading and `sstore` for storing and updating storage slots
2. `extcodesize`
3. `0x40`

Chapter 12, Upgradable Smart Contracts

1. A contract address can be supplied to another contract either at the time of contract deployment or following deployment. A constructor can be used to supply the value of the contract address at the time of deployment, while a function can help to update the contract address. This pattern aids in writing upgradable smart contracts.
2. Yes, it is possible to do so.
3. We need upgradable smart contracts to future-proof smart contracts. Smart contracts, once deployed, are immutable, and upgradable smart contracts assist in overcoming this feature of smart contracts.

Chapter 13, Writing Secure Contracts

1. `Msg.sender` refers to the immediate caller while `tx.origin` refers to the original caller in chain. `tx.origin` is always an externally owned account whereas the `msg.sender` value can be a contract account or an externally owned account.
2. Recursion happens because the `receive` function in the `hacker` contract calls the `withdraw` function, which, in turn, calls the `receive` function unknowingly because it transfers Ether to the `hacker` contract.
3. Checks, effects, and the interaction pattern are three distinct stages in a sequence within a function that change the contract state and help transfer tokens and Ethers to other accounts securely. All incoming argument validation for correctness is executed as part of the check stage. This stage also includes validating the current state of the contract. By checking that the context and environment are at the conducive stage, the check stage ensures that nothing goes wrong from a state and incoming argument perspective. It stops execution in case any of the checks fail. Then comes the effects stage. At this stage, based on the logic implemented, all state changes are executed – state variables are updated with current values, state transitions are made, and local variables are updated with necessary calculations. At this stage, it is still possible to revert the changes if something untoward happens at the interact stage. Till now, no Ether has been transferred or external communication has happened with other third-party addresses and accounts. At the last stage, during the interaction, all external communications and Ether transfers happen. This pattern ensures that any interaction happens only after all the state changes have happened. Even if someone tries to implement a reentrancy attack on the contract, it will not be successful.

Chapter 14, Writing Token Contracts

1. Fungible (ERC20) and non-fungible (ERC721).
2. ERC223 helps in contract-to-contract communication, especially with regard to token transfer from one contract to another. It provides guidance for the implementation of the `tokenReceiver` function and its call to ensure that tokens are not lost.
3. ERC165 helps to publish the interfaces supported by a contract. If an interface is supported, all its functions will also be available for invocation. Clients can use the `supportsInterface` function provided by ERC165 to check whether a contract implements an interface and, if so, can consume its function.

Chapter 15, Solidity Design Patterns

1. References
2. Modifiers

Index

Symbols

^ character 64

A

abstract contracts 159, 160

address

recovering, with ecrecover 117-122

address data type

about 99

contract function 99

functions 99

payable address 99

send function 99

transfer function 99

variations 99

address global variables 116

address-related functions

about 179

address callcode method 185

address call method 181-184

address delegatecall method 185, 186

address send method 179, 180

address transfer method 181

allowance 317

Application Binary Interface

(ABI) 32, 181

application development life

cycle management 222

array-out-of-index errors 194

arrays

about 90, 342

dynamic arrays 91

fixed arrays 91

properties 95

special arrays 93

structure 95-97

assembly programming 253-255

assert function 202-204

assignment

rules 79-85

asymmetric cryptography 7

B

blockchain

about 5, 6

database 5

decentralized 5

distributed 5

- ledger 5
 - need for 6
 - block explorer
 - using 244-247
 - block-level global variables 112, 113
 - blocks
 - about 23, 24, 256
 - relating, to transactions 12
 - relationship between 10, 11
 - Boolean data type 87, 88
 - break statement 134
 - browser-solidity 26
 - bytecode 62, 251
 - byte data type 88-90
 - bytes array 93, 94
- ## C
- C3 linearization 154
 - caret 64
 - Check-Deduct-Transfer (CDF) 180
 - Check-Effects-Interaction (CEI) 180
 - composition
 - simple solutions, implementing with 281-285
 - consensus
 - about 13
 - Proof-of-Stake (PoS) 16
 - Proof-of-Work (PoW) 13-15
 - console command 233
 - consortium network 38
 - constant functions 178
 - constant qualifier 70
 - constitutes upgradability 270-272
 - containment relationship 349
 - continue statement 135
- contract accounts 18
 - contract address
 - determining 265, 266
 - contract composition 150
 - contract constructor 148, 149
 - contract deployment
 - instance addresses, providing during 273
 - instance addresses, providing following 274
 - contract functions
 - calling 263-265
 - contract global variables
 - selfdestruct 117
 - suicide 117
 - this 116, 117
 - contract-level global variables (state variable)
 - about 108
 - internal 109
 - private 109
 - public 108
 - contract polymorphism 157, 158
 - contracts
 - about 66
 - structure 67
 - cryptographic global variables 115
 - cryptography
 - about 7
 - asymmetric cryptography 7
 - digital signatures 8-10
 - hashing 7, 8
 - relationship, between blocks 10, 11
 - symmetric cryptography 7
 - transactions, relating to blocks 12
 - types 7
 - custom errors 207

D

data cohesion 349
 data locations, of variables
 factors 79
 rules 79-85
 data modeling
 in Solidity 343
 data modeling, example
 data, adding 351
 data, updating 352
 performing 350
 records, retrieving 353, 354
 single data record, retrieving 352
 state variable 351
 structure 350
 data structures, for storing variables
 calldata 79
 memory 79
 stack 79
 storage 79
 data-type-overflow errors 194
 data types
 arrays 342
 exploring 76
 in Ethereum 341
 mappings 342
 reference types 77, 78
 value types 76
 debugging
 overview 238
 Decentralized Application
 Organization (DAO) 308
 Decentralized Applications (DApp) 97
 Decentralized Autonomous
 Organizations (DAOs) 357

dependency injection
 about 272
 instance addresses, providing during
 contract deployment 273
 instance addresses, providing following
 contract deployment 274
 digital signatures 8-10
 divide-by-zero errors 194
 do...while loop 131, 132
 dynamic arrays 91

E

ecrecover
 used, for recovering address 117-122
 EIP-165 327
 EIP223 327, 330-332
 Elliptic Curve Digital Signature
 Algorithm (ECDSA) 117
 embedded structure rules
 containment relationship 349
 data cohesion 349
 nested structure 349
 out-of-bounds nested structure 349
 relationships within 350
 reviewing 348
 static data, within nested structure 349
 encapsulation 155
 end-to-end transaction 25
 entity modeling
 about 340
 Ethereum storage 341
 enumerations 74, 97, 98
 enum keyword 74
 ERC20
 about 309, 310
 events 312-320
 functionality 310-312

ERC20 tokens 309
ERC165 334-337
ERC721
 about 309, 322, 332-334, 337
 implementation 323-329
 reference link 322
Ether 18, 307
Ethereum
 about 36, 307, 355
 data types 341
 main Ethereum network 37
 main network 37
 test network 37
Ethereum accounts
 about 17
 contract accounts 18
 Ether 18
 externally owned accounts 17, 18
 gas 20
 transaction 20-23
Ethereum Blockchain Explorer
 reference link 244
Ethereum Natural Specification (Natspec)
 reference link 64
Ethereum nodes
 about 16
 Ethereum validators 17
 EVM 16
 mining nodes 17
 types 16
Ethereum Requests for Comment
 (ERC) 307-309
Ethereum storage
 about 341
 calldata 341
 global storage 341
 memory storage 341
 stack 341

Ethereum validators 17
Ethereum Virtual Machine (EVM)
 about 9, 16, 62, 142, 187
 code, executing 62
 data storage 79
EtherPot contract 297, 298
events
 about 73, 214-218
 all events, watching 217-218
 declaring 215
 individual events, watching 216
 using 215, 244
exception handling
 about 194-196
 implementing, with try-catch
 blocks 209-214
 with assert function 202-204
 with require function 197-201
 with revert function 204-208
explicit conversion 110
externally owned accounts 17, 18

F

fallback functions 187-190
first block 11
fixed arrays
 about 91
 example 91
fixed-sized byte arrays 88
for loop 132, 133
function input 170-172
function-level local variables 108
function output 170-172
function polymorphism 156

functions
 about 74
 constant 75
 declaring 75
 external 75
 internal 75
 payable 76
 private 75
 public 75
 pure 76
 view 76
 fungible token 309

G

ganache-cli
 installing 47-50
 gas 20
 Geth
 configuring 38
 installing, on macOS 39
 installing, on Windows 39, 40
 requisites for configuration 41
 global variables
 versus msg.sender 114
 Goerli
 about 38
 URL 38

H

Hacker contract 299, 300
 haltable smart contract pattern 366-368
 hashing 7, 8

I

if decision control 128
 if...else statement 128, 129
 implicit conversion 110
 import keyword
 about 65
 using 65
 inheritance
 about 150
 hierarchical inheritance 152
 multilevel inheritance 152
 multiple inheritance 153-155
 simple solutions, implementing
 with 277-281
 single inheritance 150-152
 init command 226
 Initial Coin Offerings (ICOs) 357
 instance addresses
 providing, during contract
 deployment 273
 providing, following contract
 deployment 274
 integer overflow 294-296
 integers
 about 86, 87
 signed integers 86
 unsigned integers 86
 integer underflow 294-296
 interfaces
 about 160-162
 advanced interfaces 162-164
 Intermediate Code (IL) 251
 internal qualifier 69
 Inter Process Communication (IPC) 40

K

Kovan
about 37
reference link 37

L

library
about 164, 165
importing 165
literals
using 85, 86
logging 214-218

M

macOS
Geth, installing on 39
main Ethereum network 37
major build number 64
many-to-many relationship 348
mappings
about 100, 342
apparent mapping 103
child mapping 103
nested mapping 103
particular value, accessing 100
single mapping identifier 103
value, storing 100
working with 100-104
memory slots
working with 258-260
message calls between contracts 247
MetaMask
about 52, 58
download link 52
installing 52-57

method overloading 156
method overriding 158
Method Resolution Order (MRO) 154
migrate command 230
mining nodes 17
minor build number 64
modifier 71, 72, 172-175, 356, 357
Morden 37
multiownership 359
multisignature (MultiSig) contracts 354, 359, 363, 364

N

nested fields
versus reference fields 343-346
nested mapping 103
nested structure
about 349
static data, within 349
new keyword
using 144-146
non-fungible token 309, 320, 321

O

Object-Oriented Programming (OOP) 63
one-to-many relationship 347
one-to-one relationship 347
operators 328
out-of-bounds nested structure 349
out-of-gas errors 194
ownership
exploring, in Solidity 354, 355
in smart contracts 354
transferring 361-363

P

Peer-to-Peer (P2P) 13
 polymorphism
 about 156
 contract polymorphism 157, 158
 function polymorphism 156
 pragma 63
 private network
 about 38
 creating 42-46
 private qualifier 69
 problematic smart contracts
 reviewing 275-277
 Proof of Authority (PoA) 37
 Proof-of-Stake (PoS) 16
 Proof-of-Work (PoW) 13-15, 37
 proxy contracts
 advanced solutions, implementing
 with 285-289
 public qualifier 69
 pure abstract contracts 160
 pure functions 178

R

receive functions 187-190
 reentrancy attack 296
 reentrancy attack, solutions
 about 302, 303
 call direction, modifying 302
 call functions 301
 Checks-Effects-Interactions,
 adopting 300
 contract accounts, checking 300
 contract, halting 302
 msg.sender address, validating 301
 tx.origin address, validating 301

ReentrancyGuard
 reference link 302
 reference fields
 versus nested fields 343-346
 reference types
 about 77, 78
 arrays 78
 mappings 78
 passing 78
 structs 78
 relationships types
 exploring 346
 many-to-many relationship 348
 one-to-many relationship 347
 one-to-one relationship 347
 Remix editor
 about 238-243
 URL 26
 Remote Procedure Calls (RPC) 40
 require function 197-201
 return keyword 136
 revert function 204-208
 Rinkeby 37
 Ropsten 37

S

SafeMath 294-296
 scope 256, 257
 security
 best practices 304, 305
 security tools, for smart contracts
 reference link 305
 signed integer 86
 smart contract deployment
 internals 32, 33
 smart contract multiownership
 establishing 359-361

- smart contract ownership
 - digital asset, transferring
 - within 364, 365
 - establishing 357, 358
 - multiownership 359
 - MultiSig contract 363, 364
 - smart contracts
 - about 26, 67, 142
 - address, using 147, 148
 - constructor 148, 149
 - creating 144
 - new keyword, using 144-146
 - ownership 354
 - writing 26-32, 143, 144
 - smart contract structure
 - about 67
 - enumeration 74
 - events 73
 - functions 74, 75
 - modifiers 71, 72
 - state variables 69
 - structs 70, 71
 - Solidity
 - about 62, 63, 252
 - advantages 252
 - code writing 62
 - data modeling 343
 - data types 70, 76
 - ownership, exploring 354, 355
 - Solidity compiler
 - installing 50
 - Solidity contracts
 - debugging, with events 244
 - debugging, with Remix editor 238-243
 - Solidity expressions 126-128
 - Solidity file
 - about 63
 - comments 64
 - contract 66
 - import 65
 - pragma 63, 64
 - special arrays
 - about 93
 - bytes array 93, 94
 - string array 94
 - state mutability 176
 - state variables
 - about 69
 - constant 70
 - internal 69
 - private 69
 - public 69
 - qualifiers 70
 - static data
 - within nested structure 349
 - stoppable smart contract pattern 366-368
 - storage slots
 - working with 261, 262
 - string array 94
 - structs
 - about 70
 - instance, creating 71
 - SWC Registry
 - reference link 305
 - symmetric cryptography 7
- T**
- temporary variable 130
 - test command 232
 - test Ethereum network
 - about 37
 - Goerli 38
 - Kovan 37
 - Rinkeby 37
 - Ropsten 37

tokens 308
transaction Merkle root hash 12

transactions
about 20-23
relating, to blocks 12
types 20, 21

transaction variables 112, 113

Truffle
about 223
commands 224
developing with 225-230
installing 223
interactive console, using 232-234
testing with 230-232
truffle help migrate command 224, 225

try-catch blocks 209-214

tuple 172

tx.origin
versus msg.sender 114

type conversion

about 109, 110
explicit conversion 110-112
implicit conversion 110

U

uint data type 78
unit testing 230
unsigned integer 70, 86
upgradable contracts
writing, with upgradable
storage 290-292

V

values
returning 257, 258
value type
about 76
address 77
bool 77
byte 77
enum 77
int 77
passing 77
unit 77
variable
contract-level global variables 108
function-level local variables 108
scoping 108, 109
view functions 178
visibility scope 175, 176

W

web3 framework
installing 51, 52
web3.js
reference link 216
WebSocket (WS) 41
wei 18
while loop
exploring 129-131
white-box testing 230
Windows
Geth, installing on 39, 40



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

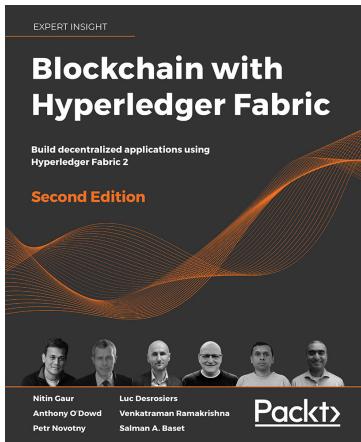
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



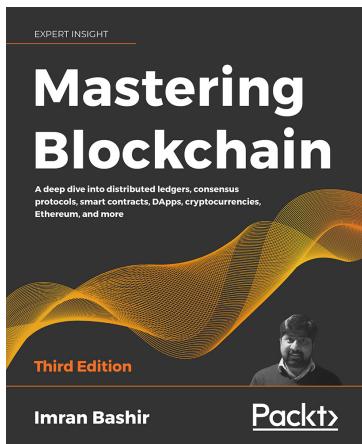
Blockchain with Hyperledger Fabric - Second Edition

Nitin Gaur, Anthony O'Dowd, Petr Novotny, Luc Desrosiers, Venkatraman Ramakrishna, Salman A. Baset

ISBN: 9781839218750

- Discover why blockchain is a technology and business game changer
- Set up blockchain networks using Hyperledger Fabric version 2
- Understand how to create decentralized applications
- Learn how to integrate blockchains with existing systems
- Write smart contracts and services quickly with Hyperledger Fabric and Visual Studio Code

- Design transaction models and smart contracts with Java, JavaScript, TypeScript, and Golang
- Deploy REST gateways to access smart contracts and understand how wallets maintain user identities for access control
- Maintain, monitor, and govern your blockchain solutions



Mastering Blockchain - Third Edition

Imran Bashir

ISBN: 9781839213199

- Grasp the mechanisms behind Bitcoin, Ethereum, and alternative cryptocurrencies
- Understand cryptography and its usage in blockchain
- Understand the theoretical foundations of smart contracts
- Develop decentralized applications using Solidity, Remix, Truffle, Ganache and Drizzle
- Identify and examine applications of blockchain beyond cryptocurrencies
- Understand the architecture and development of Ethereum 2.0
- Explore research topics and the future scope of blockchain

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Solidity Programming Essentials*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click [here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

