Ethereum DApps for Web developers

Asif Mallik

EARLY RELEASE
Raw & Unedited

Table of Contents

Introduction

Ever since the invention of Bitcoin in 2012, blockchain technologies and cryptocurrencies have taken the world by storm. New cryptocurrencies are being released constantly and older ones are being improved upon. One such blockchain technology is Ethereum, which sets itself apart from the rest by allowing users to write and deploy smart contracts, code that run directly on the blockchain.

Ethereum has opened up a whole new type of apps that are completely decentralized and do not rely on any central server, hence it is trustless and no one can tamper with the data. On top of that, it provides anonymity and allows easy exchange of money in the form of the cryptocurrency Ether. This combination of features were never possible before, making Ethereum an exciting new platform for app development.

This book consists of 7 chapters, each of which lays out an aspect of DApp development.

In the first chapter, we will familiarize ourselves with Ethereum, its underlying technologies and terminologies we will use throughout the book.

In chapter 2, we are going to learn about Solidity the programming language for smart contracts in Ethereum and Remix, an IDE for Solidity.

In chapter 3, we will dive into more advanced smart contracts such as those involving more cryptography and external data

In chapter 4, we will learn to deploy and interact with smart contracts on a private blockchain.

In chapter 5, we will learn to test smart contracts using both JavaScript and solidity

In chapter 6, we are going to create our very own DApp frontend for an ICO we built in chapter 3.

In the final chapter, we are going to discuss some of the security issues that arise in DApp development and how to navigate around them.

Prerequisites

Throughout the book, we will assume readers have prior experience with programming. We will not go through the basic concepts of programming such as variables, loops and functions. Moreover, in the later chapters such as testing smart contracts and DApp frontend development, we will also assume readers are familiar with JavaScript and web technologies. However, it is not absolutely necessary and can be picked up as we progress.

At the end of the book, we will know enough to write our very own secure DApp. Hope you enjoy!

The Ethereum Platform

Ethereum is a blockchain based platform for running smart contracts and decentralized applications (DApp). It consists of many nodes which can be run by anyone with a computer. Every single node in the network stores the blockchain and verifies each transaction. This ensures decentralization and security. Central to Ethereum is the cryptocurrency Ether (ETH) which is necessary for deploying and running smart contracts.

Ethereum is being used for various purposes such as value transfer, DApps, initial coin offerings (ICO) for crowdfunding and decentralized autonomous organizations (DAO). This is because Ethereum has been made to be value-agnostic which means it allows developers to decide what to use it for, instead of assuming it will be used in a certain way. Hence, it is possible and in fact, highly likely, that the Ethereum platform will be used to build various systems we cannot anticipate today.

Blockchain

Invented by Satoshi Nakamura in 2012, it is what started the cryptocurrency revolution. Without blockchains, Bitcoin and Ethereum would never have been possible. A blockchain is a public database in which every single transaction of a cryptocurrency, such as Ethereum, is saved chronologically. Every node running Ethereum clients have a copy of the entire blockchain.

Accounts

There are two types of account:

- Externally Owned Accounts (EOA)
- Contract Accounts

Every account has an address, can hold and transfer ether to other accounts and also call other contracts.

Externally Owned Accounts

These are accounts owned and controlled by real users who hold their private keys. Anyone with the private key are owners of the account and can send transactions with it.

Contract Accounts

These are essentially accounts designated to smart contracts. Each of them has some code, set during deployment, associated with it. They cannot initiate transactions but can respond to them in a predefined manner according to its contract code. It can read from and write to an internal storage dedicated to it on the blockchain. It can also call other contracts.

Smart contracts are persistent, so unless there is a predefined way to destroy the contract, it will keep on running forever as long as the Ethereum network is alive. They are also static, so once they are deployed, their code can never be changed by anyone.

Solidity

Solidity is a statically typed, contract-oriented, high level programming language that is similar to JavaScript. It has been specifically designed to be used for writing smart contracts for Ethereum which are compiled to bytecode. After compilation, bytecode can be deployed to the blockchain. Bytecode is a low level code that can be executed efficiently by a virtual machine. While there are other programming languages for writing smart contracts such as LLL, Viper and Serpent, Solidity is by far the most popular.

Ethereum Virtual Machine (EVM)

Every node in the Ethereum network has an EVM. The bytecode in smart contracts are run on the EVM by every node to reach a consensus about the result of calling the contract. By reaching a consensus it can be ensured that every node is correctly executing the contract code and not maliciously manipulating the blockchain. Since, every node runs the code, the EVM can be thought of as a distributed computer.

Transactions and Messages

A transaction can only be sent by an externally controlled account as it has to be signed by the private key of an EOA. A transaction can send ether to an account, call contract code or both.

Messages on the other hand are sent by contract accounts. Messages can also send ether to an account, call other contracts or both. However, they can only be sent once the contract is called by an initial transaction. So, they are analogous to function calls in other programming environments whereas transactions can be thought of as initial executions.

Gas

Gas is a special unit in the EVM used to measure computational work. Each operation carried out in the EVM as a result of a transaction has a computational cost attached to it. The more complex the transaction, the greater the computational cost, hence, the greater the gas cost. The EOA sending the transaction will have to pay the gas cost associated with it.

However, gas is not something accounts can hold. It needs to be converted to ether so that EOA can pay it. This is where gas price comes in. Miners, who choose transactions to add to the blockchain, indirectly sets gas prices. In actuality, it is the EOA which is sending the transaction which sets the gas price. However, it is only accepted into the blockchain if the miners deem the price to be right. Therefore, the higher the gas price set, the likelier and faster it will be confirmed. The fee paid in ether, therefore, is:

```
Fee (in ether) = Gas cost * Gas price
```

The reason why computational cost is not directly denoted in ether is due to the unstable price (in fiat currency) of ether. If that had been the case, the cost of sending transactions would highly vary according to the price of ether. Instead the gas price is made flexible so that it can accurately reflect the actual cost of running the computations by miners. For instance, when the price of ether rise from 200 USD/ETH to 400 USD/ETH, the gas price can be halved by miners.

Fees are enforced to ensure users do not spam the Ethereum network with many "useless" transactions which would not only prevent more important transactions from getting processed but also make the blockchain too large for many to even run a node, leading to centralization.

Gas Limit

Gas limit is another property in transactions and messages. This sets the highest gas cost the account sending it is willing to pay. If the gas limit exceeds while running the transaction or the message, all the changes to the blockchain is reverted but the fee is still incurred.

Limitations of Ethereum

Despite the great potential of the Ethereum platform, it has its pitfalls and limitations. Some of these will be solved in future phases of Ethereum while others are intrinsic.

- High cost of computation and storage: As described above, each operation costs ether. This makes any computation costly. Even simple-looking computations such as string manipulations can have non-trivial cost.
- No privacy on the blockchain: Everything on the blockchain is public, hence smart contracts cannot have any secret variable. While private variables exists in solidity, anyone with the blockchain can look into its data and find all the data stored.
- Cannot access external data: Smart contract cannot get external data such as data from various online APIs. They can only get data from the blockchain.
- Smart contracts are static: While this makes smart contracts trustless, it also makes it impossible to update it to add new features or fix bugs. If bugs are found in the smart contract, they may be exploited by malicious users to steal ether from the contract and nothing can be done immediately to prevent it. This has been the case with The DAO hack.
- Scalability / Low number of transactions per second (optional): Currently, Ethereum can only handle 13 transactions per second. This prevents mass adoption as that would require a much greater rate of

transaction. This will most likely be achieved through a number of techniques such as sharding.

Writing Smart Contracts with Solidity

Solidity is a contract-oriented programming language. Contracts are the highest unit of code in Solidity. These can be instantiated and have state variables and functions associated with them. These functions can access and modify the state variables. In other words, contracts are analogous to classes in other languages such as Java and C++.

The instantiated contracts are stored on the blockchain in the form of contract accounts, and their member functions provide an interface to other accounts to view and modify their state variables.

Remix IDE

Throughout this chapter, we are going to use Remix IDE. Remix is an IDE for Solidity which can compile and run smart contracts on a blockchain running in a JavaScript VM. No changes are made to the actual Ethereum blockchain which would make development very costly. The Remix IDE runs on the browser so there is no need to download anything.

Basic Structure of a Smart Contract

```
pragma solidity ^0.4.17;

contract Ballot {
    address public owner;

    function Ballot() {
        owner = msg.sender;
    }
}
```

The above code shows a very basic smart contract.

`pragma solidity ^0.4.17;`

Ethereum and solidity is still evolving. As a result, new features are constantly being added and removed from the language. So, the version of solidity being used must be explicitly declared at the beginning of every solidity file. We are going to use version 0.4.17 throughout this book.

`contract Ballot { ... }`

This declares a contract called Ballot and the curly braces contain code defining the behavior of the contract which is represented by `...` here.

`address public owner;`

This declares a state variable called `owner` in the contract of type `address`. State variable are declared directly under the contract `Ballot` and they be accessed anywhere from within the contract. The `public` keyword allows the variable to be read from other accounts.

```
function Ballot() { ... }
```

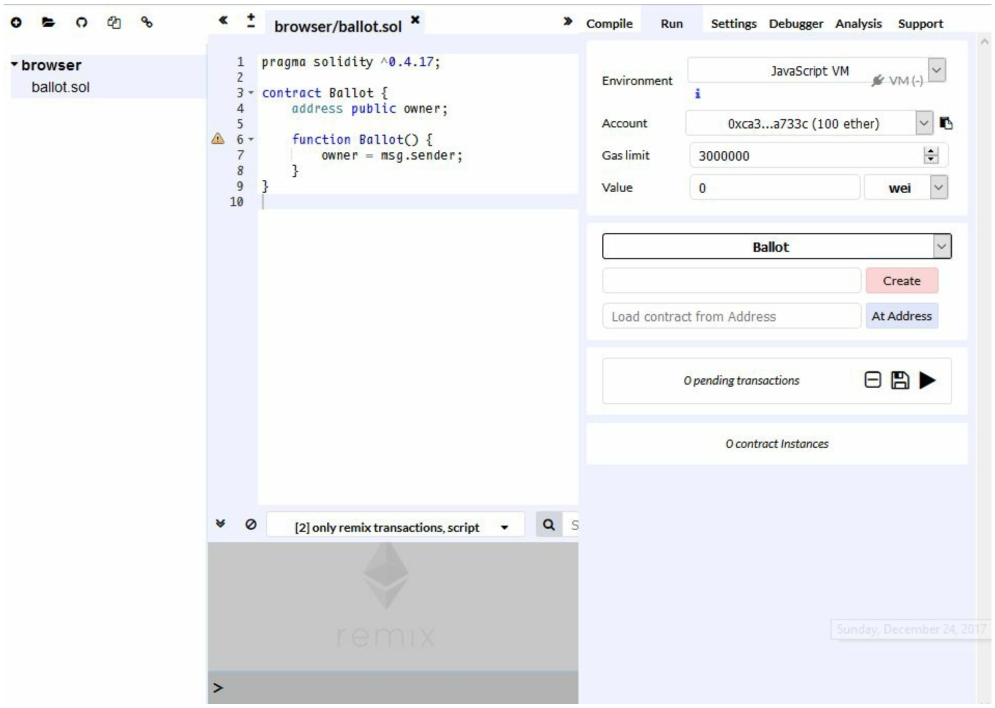
This is how a function is declared. A function with the same name as the contract is its constructor function and is only run when the contract is deployed. All the code inside the function, is therefore, only run once.

```
owner = msg.sender;
```

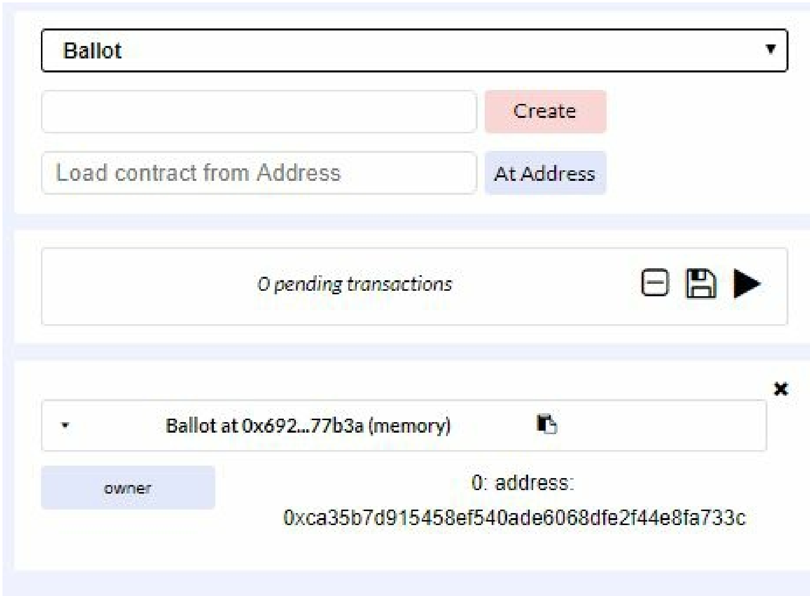
`msg.sender` refers to the address of the account sending the message, which in this context is the user deploying the contract. So, the owner variable is set as the address of the user deploying the contract.

Deploying and running smart contracts in Remix

To deploy it in Remix, we are first going to open it up by clicking here. Next, we are going to erase the content in `browser/ballot.sol` and replace it with the above code. Then we will click the `Run` tab.



The `Account` dropdown reveals the addresses of the accounts with which we can deploy our contract along with their Ether balance. To deploy this contract, we are going to click `Create`. Under `Ballot`, we will now see an instance of `Ballot` with its address in parentheses. We can also observe that the balance of our main account has decreased. Clicking on the `owner` button in our `Ballot` instance reveals the address of our main account as we used it to deploy the contract.


Ballot
Create
Load contract from Address
At Address
0 pending transactions
Ballot at 0x692...77b3a (memory)
owner
0: address:
0xca35b7d915458ef540ade6068dfe2f44e8fa733c

Similarities with JavaScript

Solidity has been developed to be similar to JavaScript to make it easier for existing JavaScript developers to switch to Solidity. Some of the similarities in syntax are

- comments
- logical operators and `if/else`
- `while` and `for` loops

We are going to assume everyone here already knows JavaScript, so we will not discuss these in detail.

Basic Types

Unlike JavaScript, Solidity is statically typed. This means, for every variable, the type must be declared along with the variable name. This is why when we declared `owner` in the previous example, we had to specify the type `address` instead of using something like the `var` keyword in JavaScript. The general syntax for declaring a variable is `type_name variable_name;`. There are many different types in Solidity.

Address

`address` refers to a hexadecimal number that points to the address of an account which could either be a contract or an externally owned account. Solidity cannot differentiate between the two without using assembly. This is an example of an `address` variable

```
address owner = donation_address;
```

`address` variables have many methods and properties and one such is `balance`. For instance, `owner.balance` will return the ether balance of the `owner` account.

Integers

Integers are round numbers and unsigned means they must be greater or equal to 0. There are two major kinds of integers, signed and unsigned. Signed integers can be both positive and negative but unsigned integers can only be greater than or equal to 0. Here are examples of both types of integers.

Unsigned integers: `uint count = 0;`

Signed integers: `int temperature = -12;`

Strings

A string works the same way in JavaScript, they are just array of characters.

`string greeting = "Hello World";`

Booleans

Booleans, perhaps the simplest type, can be either `true` or `false` and it works the same way as in JavaScript.

`bool completed = false;`

Type Declaration in Function Arguments

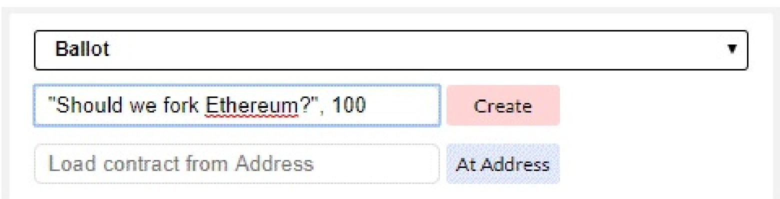
Unlike JavaScript, in Solidity, the arguments must have its types declared and when running those function, only arguments of those types can be passed.

```
pragma solidity ^0.4.17;

contract Ballot {
    address public owner;
    string public proposal;
    uint public maxVote;
```

```
    function Ballot(string _proposal, uint _maxVote) {
        owner = msg.sender;
        proposal = _proposal;
        maxVote = _maxVote;
    }
}
```

In order to pass arguments into the constructor function, before deploying, we will have to insert the arguments into the input box, separated by a comma, beside the `Create` button like this



Now, clicking `proposal` and `maxVote` reveals the arguments we passed in.

Member Functions (Methods)

These are functions that are part of a contract but are not constructor functions. Functions are the only way users can read to and write from state variables of a contract. We will now write a new function for voting on the `Ballot` contract.

```
pragma solidity ^0.4.17;

contract Ballot {
    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public positiveVoteCount;
    uint public negativeVoteCount;

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function vote(bool option) {
        if(option) {
            positiveVoteCount++;
        }else{
            negativeVoteCount++;
        }
    }
}
```

The function `vote` can be called by any account for any number of time and increment the value of `positiveVoteCount` or `negativeVoteCount` depending on whether or not they set `option` as `true` or `false`. The idea is that, at the end, we will get to see which option has a greater vote count.

Another thing to notice here is that the values of `positiveVoteCount` and `negativeVoteCount` was never set to `0`, yet, they can be incremented. While this would lead to an evaluation of `NaN` in JavaScript, as there is no type declaration. Here, however, when variables are declared, they are set to default values for each type, for example, `0` for `uint` and `int`, `false` for `bool`, `""` for `string` and `0x0000000000000000000000000000000000000000` for `address`.

In Remix, this member function will be shown under the contract instance as another button `vote`. The parameter `option` is set in the input box beside it.

The public Keyword

The `public` keyword that we used earlier to allow state variables to be read by other accounts such as `address public owner;`, in fact, generates a getter member function automatically during compilation such as this one

```
function owner() returns (address) {
    return owner;
}
```

Throwing an Error

Sometimes, the input are invalid or certain conditions are not met, so, the transaction (function call) needs to be stopped immediately. In this case, we would like to ensure that the `vote` function does not run if `maxVoteCount` has already been reached.

Like in most programming languages, this is done by throwing an error. In Solidity, we use the `require` function, which takes in a `bool` as an argument and if `true` it throws an error, otherwise, the function proceeds to run as normal.

```
pragma solidity ^0.4.17;

contract Ballot {
    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    uint public positiveVoteCount;
    uint public negativeVoteCount;

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function vote(bool option) {
        require(maxVoteCount > totalVoteCount);
```

```
        totalVoteCount++;
        if(option) {
            positiveVoteCount++;
        }else{
            negativeVoteCount++;
        }
    }
}
```

Here, every time the `vote` function runs, `totalVoteCount` is incremented independent of the `option` the user voted. Once the statement `maxVoteCount > totalVoteCount` no longer evaluates to `true` which will happen once they are equal, the `require` function throws an error, effectively stopping the `Ballot` contract from ever being changed again.

Complex Types

Mapping

The issue with the `Ballot` contract now is that a single account can vote as many times as they want and manipulate the outcome of the `Ballot`. To prevent this, we are going to use an a complex data type called `mapping`. What `mapping` does is it maps one key to a value and the key and value could be of two different types. In our case, we could map `address` of voters to `bool` which will indicate whether they voted or not.

```
pragma solidity ^0.4.17;

contract Ballot {
    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    uint public positiveVoteCount;
    uint public negativeVoteCount;
    mapping (address => bool) public voted;

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function vote(bool option) {
        require(maxVoteCount > totalVoteCount);
```

```
            require(!voted[msg.sender]);
            totalVoteCount++;
            voted[msg.sender] = true;
            if(option) {
                positiveVoteCount++;
            }else{
                negativeVoteCount++;
            }
        }
    }
```

`mapping (address => bool) public voted;`

This is the syntax for mapping `address` keys to `bool` values. In general it is `mapping (key_type => value_type) variable_name;`. When initialized, all possible keys of the `key_type` are mapped to the default value of the `value_type`, in this case, `false`.

`require(!voted[msg.sender]);`

`voted[address_key]` is the syntax used to get the `bool` mapped to the `address_key`. This is quite similar to how JavaScript object properties can be accessed dynamically. When ran first, this will be `false`, so, the function will continue to run, however, the next time, due to

`voted[msg.sender] = true;`

being ran once by the same address, the function will throw an error. Moreover, using `mapping`, we can further simplify the contract code by replacing `positiveVoteCount` and `negativeVoteCount` by another `mapping`. See if you can solve this on your own before proceeding.

```
pragma solidity ^0.4.17;

contract Ballot {
    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    mapping (bool => uint) public optionsVoteCount;
    mapping (address => bool) public voted;

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function vote(bool option) {
        require(maxVoteCount > totalVoteCount);
        require(!voted[msg.sender]);
        totalVoteCount++;
        voted[msg.sender] = true;
        optionsVoteCount[option]++;
    }
}
```

Here instead of having two separate variables for `true` and `false` votes, instead we just use `mapping` for both of them and increment them instead.

Struct

Right now, while we can check whether a particular account has voted, we cannot know which option it voted. While we can create a separate `mapping` for this, there is a cleaner method. `struct` allows us to create a new type with predefined properties. This is an example of how `struct` works.

```
struct Person {
    string name;
    uint age;
    bool lovesMeta;
}

Person author;
author.name = "Asif Mallik";
author.age = 19;
author.lovesMeta = true;
```

Now, all the properties of `author` can be accessed. This provides a neat way of dealing with related variables. For this purpose, we can create a `struct` `Vote` which will have two `bool` properties, `voted` and `option`.

```
pragma solidity ^0.4.17;

contract Ballot {
    struct Vote {
        bool voted;
        bool option;
    }

    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
```

```
    mapping (bool => uint) public optionsVoteCount;
    mapping (address => Vote) public votes;

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function vote(bool option) {
        require(maxVoteCount > totalVoteCount);
        require(!votes[msg.sender].voted);
        totalVoteCount++;
        votes[msg.sender].voted = true;
        votes[msg.sender].option = option;
        optionsVoteCount[option]++;
    }
}
```

Now, anyone can look up what an account voted with its `address`.

```
votes[msg.sender].voted = true;
votes[msg.sender].option = option;
```

When `vote()` is ran, the `voted` property of `Vote` mapped to the `address` of the sender account is set to true, whereas `option` property is set to the `option` variable passed in by the sender.

Arrays

Right now, the `Ballot` contract is only restricted to two options, `true` and `false`, but in reality, `Ballot` can be a lot more complex with many different options. To allow for this, we are going to have to use arrays. Arrays in Solidity can only hold one type of values for example `uint`.

```
pragma solidity ^0.4.17;

contract Ballot {
    struct Vote {
        bool voted;
        uint option;
    }

    struct Option {
        uint voteCount;
        string description;
    }

    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    Option[] public options;
    mapping (address => Vote) public votes;

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function addOption(string option) {
        require(msg.sender == owner);
        require(maxVoteCount > totalVoteCount);
```

```
        options.push(Option({
            description: option,
            voteCount: 0
        }));
    }

    function vote(uint option) {
        require(maxVoteCount > totalVoteCount);
        require(!votes[msg.sender].voted);
        require(option < options.length);
        totalVoteCount++;
        votes[msg.sender].voted = true;
        votes[msg.sender].option = option;
        options[option].voteCount++;
    }
}
```

Here, we use arrays to save multiple options.

```
struct Option {
    uint voteCount;
    string description;
}
```

This declares a `struct` `Option` with two properties, `voteCount` which denotes the number of votes the option got and `description` which will be the description of the option.

```
Option[] public options;
```

This declares an array of `struct` `Option` which is publicly accessible. Arrays can be composed of anything, `struct`, other arrays and even `mapping`. This replaces the `optionsVoteCount` `mapping` used before.

```
function addOption(string option) {
    require(msg.sender == owner);
    require(maxVoteCount > totalVoteCount);
    options.push(Option({
        description: option,
        voteCount: 0
    }));
}
```

This function will allow the `owner` to add new option for voting. The first line checks that the account sending the transaction is the owner. Next, it checks that the vote is still not over. Next, it pushes a new `Option` item to the `options` array.

The `vote` function now takes in a `uint` which indicates the `index` of the option the user is voting in the `options` array.

`require(option < options.length);`

This simply ensures that the account provides an option that is less than the `length`, which indicates the number of item in `options`, so that the `option` voted by the user actually exists.

Lastly `options[option].voteCount++;` just increments the vote count of the option voted by the account.

Iterating over arrays

Until now, it was easy to check the outcome of the `Ballot` by simply comparing `positiveVoteCount` and `negativeVoteCount` but now with many possible options, this is difficult. So, we are going to introduce a new function that returns the outcome of the `Ballot`.

```
function getResult() returns (bool, uint) {
        uint winningOption;
        uint highestVoteCount;
        for(uint i = 0; i < options.length; i++) {
            if(options[i].voteCount > highestVoteCount) {
                highestVoteCount = options[i].voteCount;
                winningOption = i;
            }
        }
        return (totalVoteCount == maxVoteCount, winningOption);
    }
```

This function will return whether the `Ballot` is completed and the index of the winning option.

`function getResult() returns (bool, uint) { ... }`

In Solidity, we can return multiple values. However, when a function returns something, the types of the values must be predefined. A `return` keyword should be followed by a parenthesis containing the types of the values returned separated by commas. Here, it is returning a `bool` and a `uint`.

Next, we defined two `uint`. These are `memory` variables which means they will not be stored in the blockchain and are only present in the EVM while running the function. Unless specified otherwise, all variables declared within a function are `memory` variables (we will learn more about this later).

`winningOption` and `highestVoteCount` will both be updated as the iteration takes place. At any given point in the iteration, `winningOption` will refer to the option with the highest vote out of the options that have been iterated over and `highestVoteCount` will refer to the vote count of the winning option.

```
for(uint i = 0; i < options.length; i++) { ... }
```

This simply loops through 0 to `options.length - 1` so that each item can be accessed by `options[i]`.

Within the for loop, it simply checks whether the current option has more votes than the previous winning option. If so, the winning option is set to the index of the current option and the `highestVoteCount` is set to the current option's vote count. Hence, at the end of it, the `winningOption` variable will be the index of the option with the highest vote.

```
return (totalVoteCount == maxVoteCount, winningOption);
```

This is the syntax for returning values. The first part of the statement evaluates to a `bool`. If `true` it means the `Ballot` is completed.

Memory and Storage variables

When declaring a complex type local variable, the location for the variable must be `declared`. This is done using either the `memory` or `storage` keyword. `memory` keyword indicates that the variable is stored in the EVM memory only during runtime but deleted afterwards. On the other hand, `storage` keyword indicates that the variable is a reference to the state variable in the blockchain. Any change made to a `storage` variable will be reflected as a change in the state variable.

Previously, in `vote`, we had

```
votes[msg.sender].voted = true;
votes[msg.sender].option = option;
```

Instead, we can assign `votes[msg.sender]` to another local variable and then set `voted` and `option`.

```
Vote storage senderVote = votes[msg.sender];
senderVote.voted = true;
senderVote.option = option;
```

Here, since the variable location is `storage`, when the properties are modified, the change is reflected in the `votes` state variable. Hence, the two are equivalent.

In the case of `addOption`, if we were to separate the following statement

```
options.push(Option({
    description: option,
    voteCount: 0
}));
```

into two, we would replace it with

```
Option memory newOption = Option({
    description: option,
    voteCount: 0
});
options.push(newOption);
```

Here we are creating a local variable `newOption` which is a `memory` variable. If `newOption` is modified in any way, after pushing it into `options`, these changes will not be reflected in the `options` array because it is saved in the runtime location.

Self-destruct

Now, we would like to create a function to allow the `owner` of `Ballot` to destroy the contract. When this is done, functions can no longer be called and Ether can no longer be sent to this contract. This can be done with the `selfdestruct` function.

```
function destroy() {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

The `require` statement ensures that the account calling the function is the owner. The `selfdestruct` function takes in one parameter to which all the Ether in the contract account is sent to. While `Ballot` is not designed to hold any Ether, many contracts are, so this is necessary.

Modifiers

Modifiers can be used to modify a function. These can check whether certain criteria are met and the function will only be run if they do. In our case, we can create a modifier to check if the account calling the function is the `owner`. If not, the function is not run.

```
pragma solidity ^0.4.17;

contract Ballot {
    struct Vote {
        bool voted;
        uint option;
    }

    struct Option {
        uint voteCount;
        string description;
    }

    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    Option[] public options;
    mapping (address => Vote) public votes;

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

```
function Ballot(string _proposal, uint _maxVoteCount) {
    owner = msg.sender;
    proposal = _proposal;
    maxVoteCount = _maxVoteCount;
}

function addOption(string option) onlyOwner {
    require(maxVoteCount > totalVoteCount);
    options.push(Option({
        description: option,
        voteCount: 0
    }));
}

function vote(uint option) {
    require(maxVoteCount > totalVoteCount);
    require(!votes[msg.sender].voted);
    require(option < options.length);
    totalVoteCount++;
    votes[msg.sender].voted = true;
    votes[msg.sender].option = option;
    options[option].voteCount++;
}

function getResult() returns (bool, uint) {
    uint winningOption;
    uint highestVoteCount;
    for(uint i = 0; i < options.length; i++) {
        if(options[i].voteCount > highestVoteCount) {
            highestVoteCount = options[i].voteCount;
            winningOption = i;
        }
    }
    return (totalVoteCount == maxVoteCount, winningOption);
}
```

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

The `require(msg.sender == owner);` statements in both `addOption` and `destroy` functions have been removed and replaced by `onlyOwner` after the parameters. This refers to the modifier `onlyOwner` declared by

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

The code within `onlyOwner` modifier runs before the function. `_;` is necessary because it is replaced by the content of the function being modified. So, after modification, the `destroy` function would be same as before. Multiple modifiers may be used like

```
function someFunctionName(bool someBoolean) onlyOwner modifier2 modifier3
returns(bool, uint) { ... }
```

The modifiers will run in the order they are arranged in the function.

Constant functions

Some functions, such as `getResult`, just read data from the blockchain. Since every single node has the blockchain stored, it does not make sense to call such functions in a transaction as that costs Ether. Users can simply run the function just on their own node. This can be achieved by using the `constant` keyword, which lets the EVM know that the function does not modify the state variables, as shown below

```
function getResult() constant returns (bool, uint) { ... }
```

However, note that when the `getResult` function is called from another function which is not a `constant` function (hence inside a transaction), the computation has to be carried out on every node for verification, thus will cost Ether in the form of gas cost.

Working with Ether

Ether is the cryptocurrency of Ethereum. What sets it apart from other cryptocurrency is that Ether can be sent and received by contract accounts with autonomous code. Till now, we did not explicitly transfer Ether with smart contracts (however, we did pay Ether from our main in the form of gas costs to call functions and deploying contracts).

Representation and Units of Ether

Just like distance can be measured in many different units such as kilometers, meters, centimeters and millimeters, Ether too can be measured in ether, microether, gigawei, wei, etc. Similarly, just like it does not make sense to denote size of a bacteria in meters, it does not make sense to measure micro-donations in ether as wei might be more suitable.

1 ether equals to the following in different units:

```
1000000000000000000 wei
1000000000000000 Kwei
1000000000000 Mwei
1000000000 nanoether
1000000 szabo/microether
1000 finney/milliether
1 ether (surprisingly)
0.001 Kether
0.000001 Mether
0.000000001 Gether
```

```
0.000000000001 Tether
```

As mentioned previously, Solidity does not support floating point numbers. Instead it uses integers everywhere. That makes representation of Ether problematic as that would make 1 ether indivisible. To get around this problem, in Solidity, all Ether are represented in wei, the smallet unit of Ether. So, when transferring 1 ether to another account `1000000000000000000` is used instead of `1`.

However, to make things easier, Solidity introduced several units, namely, `ether`, `finney` and `szabo`. When an integer is followed by any of these units, they are evaluated to wei and is represented by an integer. So, `1 ether` evaluates to `1000000000000000000`.

Receiving Ether

Now, we are going to write a new contract `Crowdfund`. This will take Ether from many different accounts and send the Ether to a predefined `receiver` account, if the `verifier` believes that the `receiver` has completed whatever task they promised. If not, the `funders` will be refunded.

```
pragma solidity ^0.4.17;

contract Crowdfund {
    address public verifier;
    address public receiver;
    address[] public funders;
    mapping (address => uint) public funds;

    function Crowdfund(address _verifier, address _receiver) {
```

```
        verifier = _verifier;
        receiver = _receiver;
    }

    function fund() payable {
        require(msg.value != 0);
        if(funds[msg.sender] == 0){
            funders.push(msg.sender);
        }
        funds[msg.sender] += msg.value;
    }
}
```

`funders` is an array which contains every account that sent Ether to the contract. `funds` is a `mapping` that maps those funder accounts to the amount they paid in ether.

`function fund() payable { ... }`

The `payable` keyword is a built-in `modifier` that allows an account to send Ether to the contract while calling that function. If the `payable` `modifier` was not present, the transaction would have failed.

`require(msg.value != 0);`

`msg.value` refers to the amount of Ether sent in the function call in wei. This statement simply ensures that the function does not run unless some Ether is sent.

```
if(funds[msg.sender] == 0){
    funders.push(msg.sender);
}
```

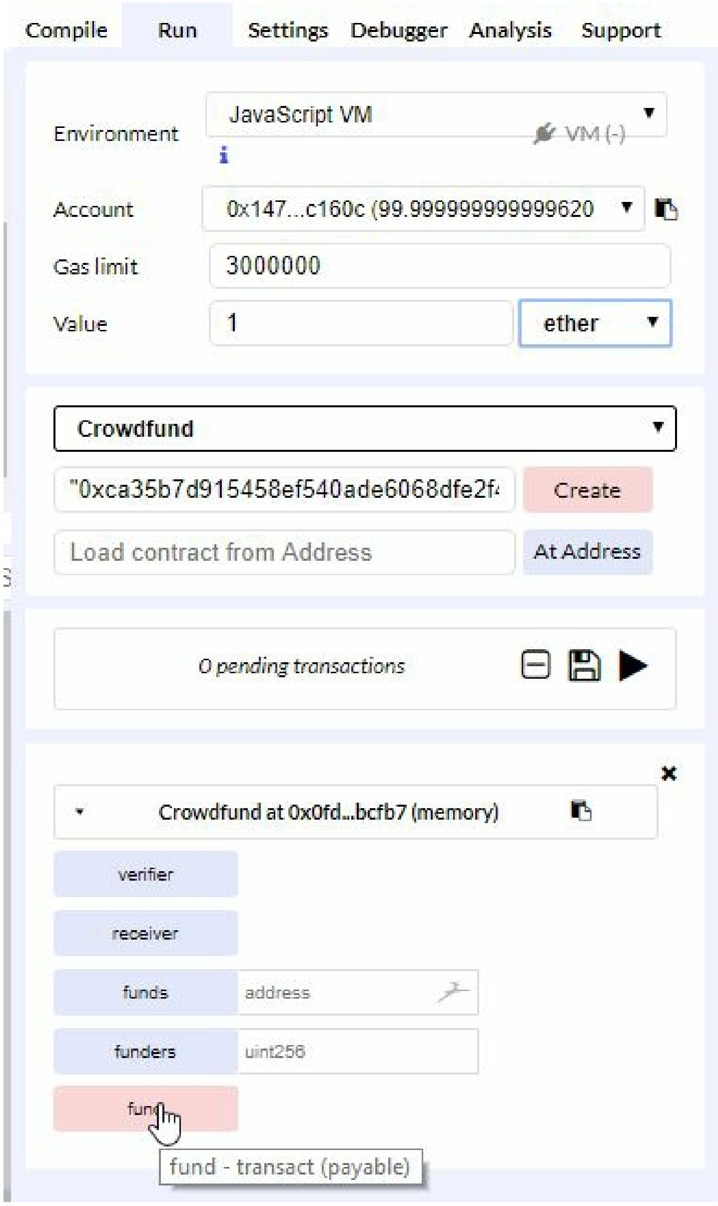
This checks whether the sender has any existing fund in the contract. If not, they are added to the `funders` array, ensuring every funder is present only exactly once.

```
funds[msg.sender] += msg.value;
```

Lastly, this adds the new amount of Ether sent to the existing fund (which may be 0).

Calling a function with Ether in Remix

In order to send Ether during a function call, we are going to have to modify the `Value` input under the `Run` tab. In order to send 1 Ether, we are going to select `ether` in the dropdown menu beside it and type in 1 before clicking the `fund` button



Fallback function

Contracts that accept Ether should have a fallback function that runs when Ether is sent to the contract without specifying a function. The function cannot have any parameter. We are now going to add the following to our contract.

```
function() payable {
    fund();
}
```

The fallback function is essentially an unnamed function in the contract. We are going to assume that anyone sending Ether to the contract is doing so to fund to the project and call `fund`.

Sending Ether

Now, we are going to add new functions to our contract to allow the `verifier` to either refund to the funders or release the funds to the receiver, depending on whether the project was completed as promised.

```
pragma solidity ^0.4.17;

contract Crowdfund {
    bool public over;
    bool public refunded;
    bool public funded;
    address public verifier;
    address public receiver;
    address[] public funders;
    mapping (address => uint) public funds;

    modifier onlyVerifier() {
        require(msg.sender == verifier);
        _;
    }

    modifier isNotOver() {
```

```
        require(!over);
        _;
    }

    function Crowdfund(address _verifier, address _receiver) {
        verifier = _verifier;
        receiver = _receiver;
    }

    function fund() payable isNotOver {
        require(msg.value != 0);
        if(funds[msg.sender] == 0){
            funders.push(msg.sender);
        }
        funds[msg.sender] += msg.value;
    }

    function release() {
        funded = true;
        receiver.transfer(this.balance);
    }

    function refund() {
        refunded = true;
        for(uint i = 0; i < funders.length; i++){
            address funder = funders[i];
            funder.send(funds[funder]);
        }
    }

    function approve(bool completed) onlyVerifier isNotOver {
        over = true;
        if(completed){
            release();
        }else{
            refund();
```

```
        }
    }
}
```

Here, we introduce three new `public` `bool`. `over` indicates whether crowdfunding is over, this is set to `true` only when either refunding or release to the receiver has taken place. `refunded` is set to `true` if `funders` are refunded whereas `funded` is set to `true` if the Ether is released to the `receiver`, which means the crowdfunding was successful.

Then we introduce two new modifiers, `isNotOver` which ensures that functions are only run if `over` is false (crowdfunding is not yet over) and `isVerifier` which can be only run by the `verifier`. `isNotOver` is applied to `fund` function so that it cannot be run to send funds after crowdfunding is over.

```
function approve(bool completed) onlyVerifier isNotOver {
    over = true;
    if(completed){
        release();
    }else{
        refund();
    }
}
```

`approve` function can only be called successfully once by the `verifier` as indicated by `onlyVerifier` and `isNotOver` as over is set to `true` as soon as `approve` runs. It takes in a `bool` which indicates whether the `receiver` has done the task they promised and if so, the fund is released. If not, the `funders` are refunded.

```
function release() {
    funded = true;
    receiver.transfer(this.balance);
}
```

The `release` function transfers all the Ether of `Crowdfund` to the `receiver`. Every `address` has a member function `transfer` which when ran, sends the amount of Ether in wei as specified by the argument. `this` refers to the `address` of the contract. `balance` is another member of `address` which is set to the value of the Ether balance in wei of the account. So, overall, the statement `receiver.transfer(this.balance);` transfers the balance of the `Crowdfund` contract to the `receiver`.

```
function refund() {
    refunded = true;
    for(uint i = 0; i < funders.length; i++){
        address funder = funders[i];
        funder.send(funds[funder]);
    }
}
```

Most of the work of the `refund` function takes place inside the `for` loop. The `for` loop iterates through every `i` from 0 to the number of `funders`. Therefore, as the loop runs, `funder` will be set to every single item of `funders`. Then, the fund contributed by the `funder` will be read from `funds` `mapping` which will then be sent to the `funder` by the `send` member function of `address`. The `send` function is the same as the `transfer` function with a subtle but important difference which is discussed below.

address.send() vs address.transfer()

`address` has two members `send` and `transfer` which both sends Ether from the current contract account to another specified `address`. However, the difference is that if `transfer` fails for some reason, an error will be thrown and the whole call will be reversed. So, when `receiver.transfer(this.balance);` fails, it throws an error and `over` and `funded` are set back to `false`. This ensures the `verifier` can run `approve` again, to either refund it or send it to the `receiver` once the issue is resolved. Otherwise, the Ether would be stuck.

Sending Ether can fail for several reason. One possibility is that the fallback function in the contract account throws an error. Another possibility is that the fallback function carries out too many computations. `transfer` and `send` allows only 2300 gas to be used to send the Ether. If too computations are done, it will throw an error. Furthermore, it could be because the fallback function does not have the `payable` modifier.

However, `send` allows the transfer to fail silently. It returns `false` when it fails and does not throw an error which allows the function to keep on running. This is necessary because even if one of the `funders` cannot accept Ether, the rest should still be refunded.

Function Visibility

Currently, any can run the `refund` and `release` function which is problematic because the `receiver` may run the `release` function to get all the funds without going through the verification process. We could use the `onlyVerifier` modifier but that is still not enough since the `verifier` may act maliciously and run the `release` function without calling `approve` and that would mean that `over` would not be set to `true`. This would mean other accounts can still send Ether to the contract even though it was not intended to act this way.

Functions can have different visibilities. By default, they have `public` visibility. This means, the function can be called both by the contract itself and other accounts (both EOA and contract accounts). When functions are set to `internal` visibility, functions can only be called by the contract itself. On the other hand, `external` means only other accounts can call the function.

```
function fund() payable isNotOver external { ... }
function release() internal { ... }
function refund() internal { ... }
function approve(bool completed) onlyVerifier isNotOver external { ...
}
```

This is how functions can be assigned different visibilities. Now, we have ensured that `release` and `refund` are only called internally whereas `approve` and `fund` are only called externally. Attempting to call `release` from another account would lead to an error being thrown.

Events

Now, we would like to modify our `Ballot` contract so that when `maxVoteCount` is reached, the contract logs that it is complete and this can be done using `event`.

When certain changes occur in smart contracts, `event` can be used to log them in Solidity. The data from the `event`, when fired, are stored in the blockchain as part of the transaction. Users can watch events in Mist wallet and DApps can react to them. However, they cannot be listened to or accessed by contracts.

This is the final version of our `Ballot` contract so the full source code is shown below

```
pragma solidity ^0.4.17;

contract Ballot {
    struct Vote {
        bool voted;
        uint option;
    }

    struct Option {
        uint voteCount;
        string description;
    }

    event LogVote(
        address indexed voter,
```

```
        uint option
    );

    event LogCompletion(
        uint winningOption
    );

    address public owner;
    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    Option[] public options;
    mapping (address => Vote) public votes;

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

    function Ballot(string _proposal, uint _maxVoteCount) {
        owner = msg.sender;
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function addOption(string option) public onlyOwner {
        require(maxVoteCount > totalVoteCount);
        options.push(Option({
            description: option,
            voteCount: 0
        }));
    }

    function complete(uint winningOption) internal {
        LogCompletion(winningOption);
    }
```

```
    function vote(uint option) external {
        require(maxVoteCount > totalVoteCount);
        require(!votes[msg.sender].voted);
        require(option < options.length);
        totalVoteCount++;
        votes[msg.sender].voted = true;
        votes[msg.sender].option = option;
        options[option].voteCount++;
        LogVote(msg.sender, option);
        if(totalVoteCount == maxVoteCount) {
            bool completed;
            uint winningOption;
            (completed, winningOption) = getResult();
            complete(winningOption);
        }
    }

    function getResult() constant public returns (bool, uint) {
        uint winningOption;
        uint highestVoteCount;
        for(uint i = 0; i < options.length; i++) {
            if(options[i].voteCount > highestVoteCount) {
                highestVoteCount = options[i].voteCount;
                winningOption = i;
            }
        }
        return (totalVoteCount == maxVoteCount, winningOption);
    }

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

Here, first of all, we modified each function and explicitly mentioned their visibility. Then we have introduced two `event` and a new function `complete` which is called when the final vote is cast. Finally we modified the `vote` function so that it calls `complete` when `vote` runs for the last time.

```
event LogVote(
    address indexed voter,
    uint option
);

event LogCompletion(
    uint winningOption
);
```

Here, we have declared two `event`. The first `event` `LogVote`, when fired, takes two arguments, `voter` which is the `address` of the account calling the `vote` function and `option` which is the index of the option. The `indexed` keyword allows us to search for a particular event log by that argument. So, if we want to search for a vote carried out by a particular account, we can simply search for that `address`. However, `indexed` data cannot be retrieved from the event log itself, that is, a hash of the argument is stored.

```
function complete(uint winningOption) internal {
    LogCompletion(winningOption);
}
```

This is a short function that is only called after the last vote is cast. It just fires the `LogCompletion` event with the `winningOption` as passed on from the `vote` function.

```
LogVote(msg.sender, option);
if(totalVoteCount == maxVoteCount) {
    bool completed;
    uint winningOption;
    (completed, winningOption) = getResult();
    complete(winningOption);
}
```

At the end of the `vote` function, first, the `LogVote` is fired with the appropriate arguments. Next, it checks whether `maxVoteCount` has been reached, if so, the `winningOption` is computed by calling `getResult` and is passed into `complete` function. `(completed, winningOption) = getResult();` is one way multiple assignment takes place when a function returns multiple values. However, the code within the block could be simplified

```
var (completed, winningOption) = getResult();
complete(winningOption);
```

Instead of declaring them separately, it can be done along with their assignment. `var` can be used instead of specifying exact type when the compiler can infer the type. Here, `getResult` returns `bool` and `uint` so, `highestVoteCount` and `winningOption` must be those respectively. The first line can be further simplified to `var (,winningOption) = getResult();` as `completed` is not required.

Contract Inheritance

Now, we are going to develop a contract that will allow users to

Contracts, just like classes in other languages, can inherit from other contracts. This means, all the state variables, modifiers, events and functions of the parent contract, unless overridden, will now be that of the child contract. Even when overridden, they will still be accessible by the child contract. Inheritance allows contracts to extend and modify an existing contract to fit the new needs without having to copy it and manually edit its code. As a result, code duplication is minimized.

```
pragma solidity ^0.4.17;

contract Ballot { ... }

contract CharityBallot is Ballot {

}
```

This is the syntax for inheritance. Here, every single member of `Ballot` is copied to `CharityBallot`. This, however, does not include the constructor and so, it is not possible to deploy this contract just yet.

Before going any further, we are going to create a separate file for `CharityBallot` naming it `CharityBallot.sol`. Then we are going to replace the `Ballot` contract with an `import` statement

```
import "Ballot.sol";
```

This will import all the contracts in `Ballot.sol` namely, the `Ballot` contract, so it is as if the `import` statement is replaced by the code from `Ballot.sol`.

Calling Parent Constructor

```
pragma solidity ^0.4.17;

import "Ballot.sol";

contract CharityBallot is Ballot("Which charity should we donate to?",
50) {

}
```

Here, we have defined the `CharityBallot` constructor to be the constructor of `Ballot` but with constant arguments. So, the `maxVoteCount` will be 50 and `proposal` will be `"Which charity should we donate to?"`. These cannot be changed during deployment.

However, that is quite restrictive as we cannot add flexibility into how the parent constructor is run. Moreover, we cannot add new functionality in the constructor function.

```
pragma solidity ^0.4.17;

import "Ballot.sol";

contract CharityBallot is Ballot {
```

```
    function CharityBallot(uint _maxVoteCount) Ballot("Which charity should we donate to?", _maxVoteCount) payable {

    }
}
```

That is why we are going to use an alternative method of calling the parent constructor function. We can access the constructor function as a `modifier` here and pass in the appropriate arguments. Furthermore, we can do further processing in the function, although we will not need this right now, so we are going to keep it empty. Lastly, the `payable` modifier will allow the `owner` to deposit Ether during deployment and this will be sent to the charities once the voting is done.

However, we have not yet added a functionalities to store and add charity `address`.

Calling Parent Functions

```
pragma solidity ^0.4.17;

import "Ballot.sol";

contract CharityBallot is Ballot {
    address[] charities;

    function CharityBallot(uint _maxVoteCount) Ballot("Which charity should we donate to?", _maxVoteCount) payable {

    }
```

```
    function addCharity(string option, address charity) external {
        Ballot.addOption(option);
        charities.push(charity);
    }
}
```

Here, we have introduced a new function `addCharity` which takes in the name of the charity and its `address`. The `address` is pushed into the `charities` array. It is followed by

`Ballot.addOption(option);`

This is one way of accessing parent member functions. Only `public` and `internal` functions of parent contracts can be called from the child contract.

Because both the `address` of the charity and the `Option` are pushed at the same time, the same index will be used to retrieve both of them.

Overriding Parent Functions

One issue with the contract is that `addOption` is inherited from `Ballot` and is still accessible to the `owner`. So, they may mistakenly or maliciously use it to add options without adding charity `address` and that would lead to fund being sent to the wrong charity. To prevent this we are going to override the `addOption` function.

```
pragma solidity ^0.4.17;

import "Ballot.sol";

contract CharityBallot is Ballot {
```

```
    address[] charities;

    function CharityBallot(uint _maxVoteCount) Ballot("Which charity
should we donate to?", _maxVoteCount) payable {

    }

    function addOption(string option) public {

    }

    function addCharity(string option, address charity) external {
        Ballot.addOption(option);
        charities.push(charity);
    }

    function complete(uint winningOption) internal {
        Ballot.complete(winningOption);
        charities[winningOption].send(this.balance);
    }
}
```

First, using an empty function `addOption` we have overridden the parent function from being called to make any changes. Secondly, we have also overridden the function `complete`. The new `complete` function first calls the `Ballot` `complete` function, then runs

`charities[winningOption].send(this.balance);`

This will send all the Ether balance of the contract account to the corresponding winner charity `address`. So, when the `vote` function of `Ballot` calls `complete` it will not run `Ballot.complete`, rather, `CharityBallot.complete` since when unspecified, the most derived (last overridden) function is called. There it is, we have successfully changed extended our base `Ballot` contract to send money to charity!

Function Overloading

Right now, we are overriding `addOption` and creating a new function `addCharity`. One could argue that instead of creating a new function, we should use `addOption` like

```
function addOption(string option, address charity) public {
    Ballot.addOption(option);
    charities.push(charity);
}
```

However, this does not actually override `addOption` of the `Ballot`. In fact, there will be two `addOption` functions. This is because of the function overloading feature of Solidity. Function signature, which is used to identify and call functions, in Solidity exists as a hash of the function name and all of its parameter types such as

`addOption(string,address)`

As a result, creating a function with the same name but with different types as parameters will not override the function of `Ballot`.

Multiple Inheritance

Now, we are going to write a new contract for crowdfunding which would let funders to vote on whether the promise was kept and release or refund accordingly. For this, instead of extending one parent contract `Ballot` or `Crowdfund` and add the functions of the other to the child contract, we are going to inherit from both contracts.

For the purpose of this section, we are going to use a simpler `Ballot` contract called `ApprovalBallot` that has only two options.

```
pragma solidity ^0.4.17;

contract ApprovalBallot {
    struct Vote {
        bool voted;
        bool option;
    }

    event LogVote(
        address indexed voter,
        bool option
    );

    event LogCompletion(
        bool winningOption
    );

    string public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
    mapping (bool => uint) public options;
    mapping (address => Vote) public votes;

    function ApprovalBallot(string _proposal, uint _maxVoteCount) {
```

```
        proposal = _proposal;
        maxVoteCount = _maxVoteCount;
    }

    function complete(bool winningOption) internal {
        LogCompletion(winningOption);
    }

    function vote(bool option) public {
        require(maxVoteCount > totalVoteCount);
        require(!votes[msg.sender].voted);
        totalVoteCount++;
        votes[msg.sender].voted = true;
        votes[msg.sender].option = option;
        options[option]++;
        LogVote(msg.sender, option);
        var (completed, winningOption) = getResult();
        if(completed) {
            complete(winningOption);
        }
    }

    function getResult() constant public returns (bool, bool) {
        return (totalVoteCount == maxVoteCount, options[true] >
options[false]);
    }
}
```

As for `Crowdfund` contract, we will just use its latest version.

```
pragma solidity ^0.4.17;

import "ApprovalBallot.sol";
import "Crowdfund.sol";
```

```
}
```

```
contract CrowdfundApprovalBallot is ApprovalBallot, Crowdfund {
    uint public fundingGoal;

    modifier ifGoalReached(bool state) {
        require(state == (this.balance >= fundingGoal));
        _;
    }

    function CrowdfundApprovalBallot(uint _maxVoteCount, address
_receiver, uint _fundingGoal)
    ApprovalBallot("Should the fund be released?", _maxVoteCount)
    Crowdfund(address(0), _receiver)
    payable {
        fundingGoal = _fundingGoal;
    }

    function vote(bool option) public isNotOver ifGoalReached(true) {
        require(funds[msg.sender] != 0);
        ApprovalBallot.vote(option);
    }

    function complete(bool release) isNotOver internal {
        over = true;
        ApprovalBallot.complete(release);
        if(release){
            release();
        }else{
            refund();
        }
    }

    function fund() public payable ifGoalReached(false) {
        Crowdfund.fund();
    }
```

```
    function isFundingGoalReached() returns (bool) {
        return this.balance >= fundingGoal;
    }
}
```

This contract inherits from both `Crowdfund` and `ApprovalBallot`. It creates a new state variable `fundingGoal`, which indicates the minimum fund in wei needed for voting to begin and the maximum fund after which no more Ether can be transferred to the contract. The `modifier` `ifGoalReached` makes use of `fundingGoal` in order to achieve this. The `modifier` takes `bool` argument which if `true`, the `modifier` ensures that `fundingGoal` has been reached, if `false` it ensures that `fundingGoal` is yet to be reached.

```
contract CrowdfundApprovalBallot is ApprovalBallot, Crowdfund { ... }
```

This is the syntax for multiple inheritance. Now, `CrowdfundApprovalBallot` will inherit and have access to all the members of both contracts.

The order of parent contracts matter when they have functions with same signature. When this happens, the function of the latter will override that of the former. So, if both contracts had declared `function vote(bool option) { ... }` `CrowdfundApprovalBallot` would have inherited the one implemented by `Crowdfund`. However, in this case, `ApprovalBallot` and `Crowdfund` do not declare any function with the same name, so the order does not matter.

```
function CrowdfundApprovalBallot(uint _maxVoteCount, address
_receiver, uint _fundingGoal)
ApprovalBallot("Should the fund be released?", _maxVoteCount)
Crowdfund(address(0), _receiver)
payable {
```

```
    fundingGoal = _fundingGoal;
}
```

The constructor function of `CrowdfundApprovalBallot` takes `_maxVoteCount` which is passed onto `Ballot` constructor and `_receiver` which is passed onto `Crowdfund`. For `_proposal`, `ApprovalBallot` is passed a constant `string` and for `Crowdfund`, `_verifier` is passed `address(0)` which evaluates to `0x0000000000000000000000000000000000000000`, so that there are no centralized `_verifier`. The `CrowdfundApprovalBallot` itself will carry out the verification.

`vote` uses both `ifGoalReached(true)` and `isNotOver` to ensure voting can only take place between reaching the funding goal and before funding being released. It also requires that the sender has contributed some Ether to the `Crowdfund` to prevent the `receiver` from gaming the system by sending `true` votes from other accounts and vice versa. On the other hand, `fund` uses `ifGoalReached(false)` to ensure funding goal has not been reached and so Ether is accepted when `balance` is still below that.

While the bulk of the `complete` function is copied from `approve` function in `Crowdfund`, it cannot be called as it has `onlyVerifier` and `external` applied to it. So, the `complete` function needs to check the result of the vote and `refund` or `release` accordingly by itself.

Interacting with other Contracts

What makes contracts in Ethereum so powerful is their ability to interact with other contracts. Just like EOA, contract accounts can call functions of other contract accounts and send Ether. This allows the ability to create highly complex ecosystem of contracts.

Calling functions from other Contracts

In the Inheritance section, we looked at how we can inherit the functionalities of two contracts, so that people can vote to release the funds. Here, instead, we will modify the `ApprovalBallot` contract to make it even simpler and so that it calls a function from another contract defined by the deploying account with the `winningOption` as an argument. This is so that this can be used with any type of contract. We will call this `ApproverBallot`.

```
pragma solidity ^0.4.17;

import "Crowdfund.sol";

contract ApproverBallot {
    struct Vote {
        bool voted;
        bool option;
    }

    Crowdfund public crowdfund;
    uint public maxVoteCount;
```

```
    uint public totalVoteCount;
    mapping (bool => uint) options;
    mapping (address => Vote) public votes;

    function ApproverBallot(uint _maxVoteCount, address _crowdfund) {
        maxVoteCount = _maxVoteCount;
        crowdfund = Crowdfund(_crowdfund);
    }

    function complete(bool winningOption) internal {
        crowdfund.approve(winningOption);
    }

    function vote(bool option) external {
        require(maxVoteCount > totalVoteCount);
        require(!votes[msg.sender].voted);
        totalVoteCount++;
        votes[msg.sender].voted = true;
        votes[msg.sender].option = option;
        options[option]++;
        var (completed, winningOption) = getResult();
        if(completed) {
            complete(winningOption);
        }
    }

    function getResult() constant public returns (bool, bool) {
        return (totalVoteCount == maxVoteCount, options[true] >
options[false]);
    }
}
```

This works mostly like `ApprovalBallot` except, `proposal` and `event` logging has been removed. Instead, the constructor function has another parameter `_crowdfund`. This refers to the `address` of the `Crowdfund` contract.

```
crowdfund = Crowdfund(_crowdfund);
```

The constructor then runs the above line which creates an interface for the `Crowdfund` contract and sets it to the state variable `crowdfund`. Now, all the `external` and `public` functions can be called by `ApproverBallot`. This is exactly what it does in `complete` function, where it calls the `approve` function of `Crowdfund` passing the `winningOption` as an argument. Note that when a call is made to another contract, a message is created which means that `msg.sender` will refer to the `address` of the `ApproverBallot` account and not the account that voted last.

To make this work, a small change needs to be introduced in `Crowdfund`. Since, the constructor function of `ApproverBallot` takes the `address` of `Crowdfund` as an argument, by definition, `Crowdfund` must be deployed beforehand. Moreover, this means `Crowdfund`, when deployed, cannot set the `verifier` as the `address` of `ApproverBallot` because it has not been deployed yet, so its `address` is unknown. So, instead we are going to introduce `changeVerifier` function in `Crowdfund` which will allow a temporary `verifier` to set `verifier` to `ApproverBallot` contract account `address`. After this, it can call the `approve` function.

```
function changeVerifier(address _verifier) onlyVerifier external {
        verifier = _verifier;
    }
```

Using Abstract Contracts as Interfaces

There is one problem with `ApproverBallot` approach. Right now, `ApproverBallot` is unnecessarily assuming that it is working with a `Crowdfund` contract, even though it can practically be used with any contract which implements `approve(bool)`. Hence, we are going to introduce an abstract `contract` that does not assume this.

An abstract `contract` has unimplemented functions and as such, cannot be deployed. It can be inherited and those function may be implemented and then this child contract can be deployed but not on its own. One use case of abstract `contract` is to use it as an interface. Instead of assuming what a contract does, we can enforce that certain functions are implemented which needs to be called.

```
pragma solidity ^0.4.17;

contract Approvable {
    function approve(bool approved);
}

contract ApproverBallot {
    struct Vote {
        bool voted;
        bool option;
    }

    Approvable public proposal;
    uint public maxVoteCount;
    uint public totalVoteCount;
```

```
    mapping (bool => uint) options;
    mapping (address => Vote) public votes;

    function ApproverBallot(uint _maxVoteCount, address _proposal) {
        maxVoteCount = _maxVoteCount;
        proposal = Approvable(_proposal);
    }

    function complete(bool winningOption) internal {
        proposal.approve(winningOption);
    }

    function vote(bool option) external {
        require(maxVoteCount > totalVoteCount);
        require(!votes[msg.sender].voted);
        totalVoteCount++;
        votes[msg.sender].voted = true;
        votes[msg.sender].option = option;
        options[option]++;
        var (completed, winningOption) = getResult();
        if(completed) {
            complete(winningOption);
        }
    }

    function getResult() constant public returns (bool, bool) {
        return (totalVoteCount == maxVoteCount, options[true] >
options[false]);
    }
}
```

This contract will work with `Crowdfund` without any further modification to it.

```
contract Approvable {
    function approve(bool approved);
}
```

This abstract `contract` defines an unimplemented function `approve` which takes in `bool` as its only argument. Now, back in the `ApproverBallot`, we are going to use `Approvable` type to store our `Crowdfund` or any other contract and using this interface, we can call the `approve` function.

Value and Gas

Let us say we have a `Shop` contract which has the following state variables and function

```
contract Shop {
    mapping (uint => uint) prices;
    mapping (address => mapping (uint => uint)) buyers;

    ...

    function buy(uint productId, uint number) payable {
        require(prices[productId] * number == msg.value);
        buyers[address][productId] += number;
    }
}
```

Now, we would like to create a wallet account for a family that would allow any of the family members to purchase anything from the shop. However, they cannot withdraw the balance so as to prevent the children from using it for other purposes.

```
pragma solidity ^0.4.17;

import "Shop.sol";

contract FamilyShoppingWallet {
    mapping (address => bool) isFamilyMember;
    Shop shop;

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

    modifier onlyFamilyMembers() {
        require(familyMembers[msg.sender]);
        _;
    }

    function FamilyShoppingWallet(address _shop) payable {
        owner = msg.sender;
        isFamilyMember[msg.sender] = true;
        shop = Shop(_shop);
    }

    function addFamilyMember(address _familyMember) onlyOwner {
        isFamilyMember[_familyMember] = true;
    }

    function buy(uint amount, uint number, uint productId)
onlyFamilyMembers payable {
        require(this.balance >= amount);
        shop.buy.value(amount)(productId, number);
    }
```

```
    function() payable {

    }
}
```

This is one use case of sending Ether to a function.

```
shop.buy.value(amount)(productId, number);
```

Any external function can be modified in two ways. `.value(x)` modifies the function so that when called, it sends `x` wei to the function. Here it sends `amount` wei to the `buy` function with `productId` and `number` as arguments. `.gas(x)` modifies the function so that the function call is limited to `x` units of gas. They can be chained together like `shop.buy.value(amount).gas(10000)(productId, number);` so that the function call is limited to `10000` gas.

Deploying a Contract

Contracts can deploy other contracts. Previously, we have introduced `changeVerifier` function to `Crowdfund` in order to allow a temporary `verifier` to change itself to the newly deployed `ApproverBallot`. Instead of doing that, we can deploy `ApproverBallot` from within the constructor function of `Crowdfund`.

```
function Crowdfund(uint _maxVoteCount, address _receiver) {
        ApproverBallot approverBallot = new
ApproverBallot(_maxVoteCount, this);
        verifier = address(approverBallot);
        receiver = _receiver;
    }
```

The above shows the new constructor function. To access `ApproverBallot`, it must be imported at the top of the file.

```
ApproverBallot approverBallot = new ApproverBallot(_maxVoteCount, this);
```

When a `contract` type is preceded by the `new` keyword, it deploys a new contract with the arguments passed within the parentheses.

```
verifier = address(approverBallot);
```

`address(approverBallot)` returns the `address` of the contract account `approverBallot` which is then assigned to `verifier`.

address.call

One limitation of `address.send()` and `address.transfer()` functions we looked at earlier is that not much computation can be done with it due to no gas being allocated to the message. So, if we want to use a `Dispenser` contract which divides all the Ether sent to it to several accounts such as

```
pragma solidity ^0.4.17;

contract Dispenser {
    address owner;
    address[] dispenseTo;

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

```
    function Dispenser(address _owner) {
        owner = _owner;
    }

    function dispenseTo(address _dispenseTo) onlyOwner external {
//adds a new account to dispenseTo
        dispenseTo.push(_dispenseTo);
    }

    function dispense() payable { //divides all the Ether equally and
sends them to every account in dispenseTo
        uint amount = msg.value/dispenseTo.length; //calculates amount
for each account
        for(uint i = 0; i < dispenseTo.length; i++) {
            dispenseTo[i].send(amount); //sends amount to each account
        }
    }

    function() payable {
        dispense();
    }
}
```

with our `Crowdfund` contract and we try to send Ether with `address.send()` the message will fail as there is not enough gas to forward the Ether to other accounts and the transfer of Ether will not take place.

To prevent this, we can modify the release function of `Crowdfund` to

```
function release() internal {
        funded = true;
        bool success = receiver.call.value(this.balance)();
        require(success);
```

```
    }
```

`bool success = receiver.call.value(this.balance)();`

Just like `contract.function` can be modified by `value` and `gas`, the `.value(this.balance)` modifies the call to the receiver. Then the `()` at the end actually carries out the function call which returns a `bool` depending on whether the call failed or not. Unlike, `send` and `transfer`, this forwards all the remaining gas to the function call. If it fails, `success` will be `false` and in the next line, due to, `require(success);`, the entire transaction will fail.

Limiting Gas

`address.call` gives more control over the gas forwarded. The above call to the receiver can be further modified to limit the gas like

`bool success = receiver.call.value(this.balance).gas(10000)();`

This will limit the gas spent in the `receiver` fallback function to `10000` gas units. While in this case it is not very useful, when calling multiple functions, it is more useful as it should not allow one function to use all the gas preventing the others from running and ultimately causing the transaction itself to fail.

Library

`library` in Solidity are just like libraries in JavaScript such as jQuery. They are only deployed once and used by many different contracts. Under the hood, a `library` is deployed as contract. However, unlike `contract` they cannot have state variables. When called by a normal contract, they can access and modify the state variables of that contract.

```
pragma solidity ^0.4.17;

library ArrayLib {
    function getMaxValue(uint[] storage self) returns (uint) {
        uint max;
        for(uint i = 0; i < self.length; i++) {
            if(max > self[i]) {
                max = self[i];
            }
        }
        return max;
    }

    function fill(uint[] storage self, uint newNumber) {
        for(uint i = 0; i < self.length; i++) {
            self[i] = newNumber;
        }
    }
}
```

This `library` defines two functions `getMaxValue` which returns the highest value in a `uint` array and `fill` which replaces all the items in the array with a single `uint` which is its second argument. Both of them takes a `uint` array as the first argument.

The `storage` keyword here indicates that the location of the variable is being passed and a copy is not being made. As a result, when the `fill` function replaces all the items, the array being passed from the `contract` will have all its items replaced. Using `memory` keyword instead will not have the same effect as a copy is made.

Next, we are going to write a `contract` to use the `ArrayLib`.

```
pragma solidity ^0.4.17;

import "ArrayLib.sol";

contract UseArrayLib {
    uint[] public someArray;

    function UseArrayLib() {
        someArray.push(1);
        someArray.push(9);
        someArray.push(5);
        someArray.push(25);
        someArray.push(12);
    }

    function getSomeArrayMaxValue() returns (uint) {
        return ArrayLib.getMaxValue(someArray);
    }
```

```
    function replaceWith(uint someUint) {
        ArrayLib.fill(someArray, someUint);
    }
}
```

The only state variable in the contract is `someArray`, a `uint` array that we are going to use with `ArrayLib`. The constructor first populates `someArray` with some values.

`getSomeArrayMaxValue` returns the maximum `uint` inside `someArray` by calling `UseArrayLib.getMaxValue`. While the syntax for calling a library function is same as calling parent contract function when contract inheritance takes place, here, the code is not copied. Rather an actual external function call takes place to another `contract`.

`library` can also be used to add methods to different type.

```
pragma solidity ^0.4.17;

import "ArrayLib.sol";

contract UseArrayLib {
    using ArrayLib for uint[];
    uint[] public someArray;

    function UseArrayLib() {
        someArray.push(1);
        someArray.push(9);
        someArray.push(5);
        someArray.push(25);
        someArray.push(12);
    }
```

```
    function getSomeArrayMaxValue() returns (uint) {
        return someArray.getMaxValue();
    }

    function replaceWith(uint someUint) {
        someArray.fill(someUint);
    }
}
```

`using ArrayLib for uint[];`

This line tells the compiler that for every `uint` array, when a member is called, to look for the member in `ArrayLib` `library` for that function. If it does match any, the `uint` array on which the member is called, is passed to the function as its first argument. So,

`someArray.fill(someUint);`

is equivalent to

`ArrayLib.fill(someArray, someUint);`

Time

Now we are going to create a smart contract that allows another person to inherit all the Ether in the contract in case the owner can passes away. This can be done by keeping track of time.

```
pragma solidity ^0.4.17;

contract Inheritable {
    address public owner;
    address public child;
    uint public lastUpdated;
    uint public interval;

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

    function Inheritable(uint _numDays, address _child) payable {
        lastUpdated = now;
        interval = _numDays * 1 days;
        child = _child;
        owner = msg.sender;
    }

    function update() onlyOwner {
        lastUpdated = now;
    }

    function inherit() {
```

```
        require(interval + lastUpdated < now);
        selfdestruct(child);
    }
}
```

There are two functions here, `update` which is called by the `owner` periodically to let the contract know that he did not pass away and `inherit` which if called a certain number of days, as defined by the `owner`, after the `owner` last called `update`, it assumes the `owner` passed away and sends the balance to the `child`

`lastUpdated = now;`

Smart contracts can access the current time in Solidity by using `now` or `blockchain.timestamp`. This gives the timestamp of the blockchain as seconds in Unix time.

`interval = _numDays days;`

`days` is a unit in Solidity which, just like Ether related units such as `finney`, multiplies it so that it is converted to seconds. So, when it follows `_numDays`, it multiplies it by 86400. Other time-related units are

```
seconds: 1
minutes: 60
hours: 3600
days: 86400
weeks: 604800
years: 31536000
```

`require(interval + lastUpdated < now);`

This statement in the `inherit` function checks whether the `interval`, as defined in days by `owner` in the constructor function, has passed after the `lastUpdated` time.

Finally, the contract is destroyed and the balance is sent to the `child` `address` as defined by the `owner` during deployment.

More about Types

Integers

While we have only looked at `uint` and `int`, there are variety of other unsigned integer and integer types, each being able to hold different sizes of numbers, starting from `uint8` and `int8` all the way up to `uint256` and `int256` (they increase at steps of 8). The number at the end represents the number of bits, for example, a `uint8` variable can hold up to `255` as its value. In fact, `uint` and `int` are just aliases for `uint256` and `int256` respectively.

Arrays

We have looked at `storage` arrays but we did not look at `memory` arrays. One important distinction between them is that `memory` arrays have a fixed length. This means we will not be able to call the `push` member function for `memory` arrays as that increments the length of the array. When initializing a `memory` array, its length must be specified in the following fashion

`uint[] memory x = new uint[](7);`

This creates an empty `memory` `uint` array with `length` 7. Now, we can read from and write to the array using `x[n]`. Assigning a `storage` array to a `memory` array will make a copy such as

```
address[] funders;
```

```
function x() {
    address[] memory fundersInMemory = funders;
}
```

This also illustrates that while `memory` array length are fixed, their length can be a variable during initialization. This is because `funders` is a dynamically-sized `storage` array whose `length` may change.

Another limitation of arrays is that, contracts cannot read dynamically-sized arrays returned from functions in other contracts.

```
pragma solidity 0.4.17;

contract A {
    uint[] x = [1, 2, 3];

    function returnsDynamicallySizedArray() returns (uint[]) {
        return x;
    }

    function returnsFixedSizedArray() returns (uint[3]) {
        uint[3] memory y = [uint(1),2,3];
        return y;
    }
}

contract B {
    A a;

    function B(address _a) {
        a = A(_a);
    }
```

```
    function callsDynamicallySizedArray() returns (uint[]) {
        return a.returnsDynamicallySizedArray();
    }

    function callsFixedSizedArray() returns (uint[3]) {
        return a.returnsFixedSizedArray();
    }
}
```

When we attempt to compile this, in `callsDynamicallySizedArray` we get the following error

```
TypeError: Return argument type inaccessible dynamic type is not
implicitly convertible to expected type (type of first return
variable) uint256[] memory.
        return a.returnsDynamicallySizedArray();
               ^----------------------------^
```

because we are trying to access a dynamically-sized array returned by `returnsDynamicallySizedArray`.

String

Strings are a dynamically-sized as well. As a result, it is constricted by the same limitations as above. It cannot be returned to functions in other contracts. Another limitation of `string` is that it costs high amount of gas to store it also due to `string` being dynamically-sized. This can be circumvented by using `bytes32` instead.

Bytes

`bytes` is a dynamically-sized array of bytes. With this, it is possible to store arbitrary data of any size. However, it is restricted by the same limitations as `string` and other dynamically-sized arrays. It is not possible to return it to another contract.

However, there are other fixed length bytes arrays such as `bytes1` all the way up to `bytes32`. These can be used to save and return strings to other contracts. An example is shown below.

```
contract Author {
    function name() returns (bytes32) {
        bytes32 authorName = "Asif Mallik";
        return authorName;
    }
}

contract Book {
    function getAuthorName(address _author) returns (bytes32) {
        return Author(_author).name();
    }
}
```

Struct

One important limitation of all `struct` types is the inability to pass into or return `struct` types from a function in another contract even when the `struct` type is defined in both contracts.

Advanced Smart Contracts

Token

Cryptocurrencies are essentially the result of a public ledger which keeps track of which account has what amount of the currency. So, if we are able to implement such a ledger on top of Ethereum, as smart contracts, we can create new cryptocurrencies. We call these tokens.

Tokens can be used for variety of things.

```
pragma solidity ^0.4.17;

contract ShopToken {
    mapping (address => uint) public ledger;

    function ShopToken(uint supply) {
        ledger[msg.sender] = supply;
    }

    function send(address to, uint amount) external {
        require(ledger[msg.sender] >= amount);
        ledger[msg.sender] -= amount;
        ledger[to] += amount;
    }
}
```

This is an example of a token. The `ledger` maps every `address` to a `uint` which refers to the balance of `ShopToken` an account owns. The constructor takes a `uint` as a parameter which is the total supply of `ShopToken` and initially the account that deploys it will have all of it. The `send` function simply deducts the sent balance from sending account and adds it to the receiving account.

However, there are many possible ways of creating a token smart contract. They can have different function names, different methods of transferring tokens, etc. While this allows for flexibility, the lack of a standard means there is no common interface. As a result, it is difficult to create develop DApps such as a "Decentralized Token Exchange" that deals with large number of tokens without having to add support for each token.

ERC20 Token

This is where ERC20 Token comes in. It standardizes tokens and defines a set of functions, its argument types and how these functions should loosely work. Other contracts can now use an interface for ERC20 tokens to interact with any token that follows this standard. The following is an abstract contract which defines ERC20 Tokens

```
pragma solidity ^0.4.17;

contract ERC20 {
    event Transfer(address indexed _from, address indexed _to, uint
_value);
    event Approval(address indexed _owner, address indexed _spender,
uint _value);
```

```
    function totalSupply() constant returns (uint totalSupply);
    function balanceOf(address _owner) constant returns (uint
balance);
    function transfer(address _to, uint _value) returns (bool
success);
    function transferFrom(address _from, address _to, uint _value)
returns (bool success);
    function approve(address _spender, uint _value) returns (bool
success);
    function allowance(address _owner, address _spender) constant
returns (uint remaining);
}
```

Every ERC20-compliant tokens must implement these functions.

- `totalSupply` returns the total supply of the tokens
- `balanceOf` returns the balance of `_owner`
- `transfer` is used to send `_value` amount of tokens to `_to` .
- `approve` is used to allow `_spender` to spend `_value` amount of tokens using `transferFrom` from their balance
- `transferFrom` allows a spender with allowance as set by `approve` to transfer tokens of amount `_value`
- `allowance` returns the allowance set by `_owner` to `_spender`

Now, we are going to make `ShopToken` ERC20-compliant.

```
pragma solidity ^0.4.17;

import "./ERC20.sol";

contract ShopToken is ERC20 {
```

```
    uint totalShopTokens;
    mapping (address => uint) ledger;
    mapping (address => mapping (address => uint)) allowances;
    string public name = "Shop Token";
    string public symbol = "SHT";
    uint8 public decimals = 18;

    function ShopToken(uint _totalShopTokens) {
        ledger[msg.sender] = _totalShopTokens;
        totalShopTokens = _totalShopTokens;
    }

    function balanceOf(address _owner) constant returns (uint) {
        return ledger[_owner];
    }

    function transfer(address _to, uint _value) returns (bool) {
        if(ledger[msg.sender] >= _value && _value > 0) {
            ledger[msg.sender] -= amount;
            ledger[_to] += amount;
            Transfer(msg.sender, _to, _value);
            return true;
        }else{
            return false;
        }
    }

    function approve(address _spender, uint _value) returns (bool) {
        allowances[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }

    function transferFrom(address _from, address _to, uint _value)
returns (bool) {
        if(allowances[_from][msg.sender] >= _value && _value > 0 &&
```

```
ledger[_from] >= _value) {
            ledger[_from] -= _value;
            ledger[_to] += _value;
            allowances[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        }else{
            return false;
        }
    }

    function allowance(address _owner, address _spender) constant
returns (uint) {
        return allowances[_owner][_spender];
    }

    function totalSupply() constant returns (uint) {
        return totalShopTokens;
    }
}
```

This is a very basic token which implements the necessary functions as defined in ERC20. However, it is possible to add new functions or even modify existing ones

```
function createToken(uint _value) payable {
    require(msg.value == _value);
    ledger[msg.sender] += _value;
    totalSupply += _value;
}
```

This allows anyone to create new tokens and add them to their balance by paying its equivalent in wei. Moreover, another function can be introduced for the owner to withdraw the Ether in the token contract.

Initial Coin Offerings (ICO)

However, there is a better way of selling tokens for Ether: ICOs. This can be done for crowdfunding a specific DApp. A separate contract for the ICO is created which owns the tokens. The ICO will transfer the tokens to accounts in exchange for Ether. They run for a specified period of time before they close and usually they have a maximum and minimum funding. If it does not reach the minimum funding goal, the Ether is refunded to the users.

The tokens bought by users will be utilized in the DApp created. As a result, people who believe the DApp will become successful will buy them anticipating the token's price to rise in the future so they can sell them at a profit. The Ether raised by the ICOs will be withdrawn by the developers so that they can build the DApp.

Now we are going to write a smart contract for an ICO for `ShopToken`.

```
pragma solidity ^0.4.17;

import "./ERC20.sol";

contract ShopTokenICO {
    ERC20 public shopToken;
    address public owner;
    uint public maxFunding;
    uint public deadline;
```

```
    uint public fundingGoal;
    uint public tokensPerEther;
    bool public withdrawalAllowed;
    bool public refundingAllowed;
    bool public started;

    modifier ensureStarted() {
        require(started);
        _;
    }

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

    modifier afterDeadline() {
        require(deadline < now);
        _;
    }

    modifier beforeDeadline() {
        require(deadline >= now);
        _;
    }

    modifier ensureRefundingAllowed() {
        require(refundingAllowed);
        _;
    }

    modifier ensureWithdrawalAllowed() {
        require(withdrawalAllowed);
        _;
    }
```

```
    function ShopTokenICO(address _shopToken, uint _price, uint
_maxFunding, uint _fundingGoal, uint _deadline) {
        shopToken = ERC20(_shopToken);
        tokensPerEther = _price;
        maxFunding = _maxFunding;
        fundingGoal = _fundingGoal;
        deadline = _deadline;
        owner = msg.sender;
    }

    function begin() onlyOwner {
        require(shopToken.balanceOf(this) >= (maxFunding *
tokensPerEther)/(1 ether));
        started = true;
    }

    function buy(address tokensReceiver) payable ensureStarted
beforeDeadline {
        uint tokensBought = (msg.value * tokensPerEther)/(1 ether);
        require(maxFunding >= msg.value + this.balance);
        require(shopToken.transfer(tokensReceiver, tokensBought));
    }

    function withdraw(address receiver) onlyOwner
ensureWithdrawalAllowed {
        receiver.transfer(this.balance);
    }

    function sell(address receiver, uint value) ensureRefundingAllowed
{
        require(shopToken.transferFrom(msg.sender, owner, value));
        receiver.transfer((value * 1 ether)/tokensPerEther);
    }

    function end() afterDeadline {
        if(this.balance >= fundingGoal) {
```

```
            withdrawalAllowed = true;
        }else{
            refundingAllowed = true;
        }
    }

    function getTokensLeft() constant returns (uint) {
        return shopToken.balanceOf(this);
    }

    function() payable {
        buy(msg.sender);
    }

}
```

The important state variables in the contract are:

- `shopToken` is the interface to the `address` of the `ShopToken` used to transfer Shop Tokens.
- `maxFunding` is the maximum amount of funding in wei the contract can receive. Any call that leads to the balance exceeding this fails.
- `deadline` is the UNIX time at which the ICO ends.
- `fundingGoal` is the minimum funding required for `owner` to `withdraw` the fund. If this is not reached by `deadline`, buyers can `sell` their tokens.
- `tokensPerEther` is the number of tokens a buyer receives for every Ether.
- `owner` can begin the ICO and `withdraw` the fund after deadline if `fundingGoal` is reached.

`begin` can only be called by the `owner`. This will start the ICO and anyone can call the `buy` function after this happens.

```
require(shopToken.balanceOf(this) >= (maxFunding * tokensPerEther)/(1
ether));
```

This ensures that `ShopTokenICO` holds enough tokens to raise the amount of Ether at `tokensPerEther` price to reach the `maxFunding`. `maxFunding` needs to be divided by `1 ether` because it is in wei.

The `buy` function takes one argument `tokensReceiver` to which the tokens are sent to. It can only be called before the deadline and after `begin` is called successfully.

```
require(shopToken.transfer(tokensReceiver, tokensBought));
```

This attempts to send `tokensBought` amount of tokens to `tokensReceiver` from the ICO contract. If it fails, it returns `false`, so the `buy` function fails as well.

The `end` function can be called by anyone after `deadline` is passed. When called, it checks if the `fundingGoal` has been reached. If not, it allows buyers to sell their tokens for Ether. If it does, the `owner` can withdraw the funds.

In order to sell `value` number of tokens using the `sell` function, the account must first call `approve` from `ShopToken` with the `ShopTokenICO` `address` and `value` number of tokens. This will allow, `ShopTokenICO` to run `require(shopToken.transferFrom(msg.sender, owner, value));` successfully which transfers the tokens to the `owner`.

Blind Auction

Next, we are going to look at how hashing can be used to store signature of private information on the blockchain and reveal them later. Hash functions take arbitrarily large data as argument and return a fixed length output. They are deterministic which means given the same data, the hash (output) will always be the same. They are also made to be difficult to reverse, which means given a hash, it is not possible to find out the input without brute-forcing (calling the hash functions with arbitrary data to get the matching hash) which is computationally time consuming and practically impossible. Any small change to the original data will change the hash completely. All of these qualities of hash functions allow hashes to be used as a signature for private data.

In order to build a blind auction, we can create a contract which has three stages

- Bidding stage: During this time, anyone can make a bid. They submit a hash of the value of Ether they are bidding and a secret data they generate. This hash will be the signature of the bid which is used to verify the bid in the next stage.
- Revealing stage: During this stage, bidders reveal their bids and send the value of Ether along with specified by the hash and the secret data. After checking whether the revealed bid is valid, it then checks against the highest bid to see if this bid is greater. If it is, the highest bid is updated.
- Release stage: After the revealing stage is over, the highest bid amount

is sent to the beneficiary.

```
contract BlindAuction {
    uint public biddingDeadline;
    uint public revealDeadline;
    mapping (address => bytes32) bids;
    mapping (address => uint) refunds;
    address public highestBidder;
    uint public highestBidValue;
    address public beneficiary;
    bool public released;

    modifier biddingAllowed() {
        require(now <= biddingDeadline);
        _;
    }

    modifier revealAllowed() {
        require(now <= revealDeadline && now > biddingDeadline);
        _;
    }

    modifier auctionOver() {
        require(now > revealDeadline);
        _;
    }

    function BlindAuction(uint _biddingTime, uint _revealTime, address
_beneficiary) {
        biddingDeadline = now + _biddingTime;
        revealDeadline = biddingDeadline + _revealTime;
        beneficiary = _beneficiary;
    }

    function bid(bytes32 _data) biddingAllowed {
```

```
        bids[msg.sender] = _data;
    }

    function reveal(bytes32 _secret) revealAllowed {
        require(bids[msg.sender] == keccak256(msg.value, _secret));
        require(msg.value > highestBidValue);
        refunds[highestBidder] = highestBidValue;
        highestBidder = msg.sender;
        highestBidValue = msg.value;
    }

    function refund(address _to) {
        uint value = refunds[_to];
        refunds[_to] = 0;
        _to.transfer(value);
    }

    function release() auctionOver {
        require(!released);
        released = true;
        beneficiary.transfer(highestBidValue);
    }
}
```

The constructor simply sets the deadlines by adding the duration of the stages to `now` and then sets the `beneficiary`. There are three different modifiers, each of which corresponds to a stage and when applied, functions can run only in that stage.

`bid` can only be called during bidding stage due to `biddingAllowed` modifier. When called, the `_data` passed is mapped to the `msg.sender` in `bids`. `_data` refers to the hash of the secret and the value of the bid. The secret is generated randomly and saved offline by the user bidding.

`reveal` can be only called during revealing stage due to the modifier. It is called with an argument `_secret` which is a randomly generated data passed along with the value of the bid to generate the hash that was previously saved by calling `bid`.

```
require(bids[msg.sender] == keccak256(msg.value, _secret));
```

Here the `msg.value` and the `_secret` is passed into `keccak256` which can take in any number of arguments. This returns a hash which is compared to `bids[msg.sender]` so as to ensure that the revealed bid is valid. The reason why the secret data is necessary is so that other users cannot brute-force by calling `keccak256` within reasonable range of values in order to find out the value of the bid.

The `refund` function is used to refund bids which have been succeeded by greater bids.

Oraclize

One limitation of smart contracts on Ethereum is their inability to access external websites and APIs. This is partly solved by Oraclize, which is an organization that handles HTTP requests made by smart contracts and return their results.

We want to create a smart contract with which you can donate to any github users. However, not all github have addresses on Ethereum. So, this smart contract will keep the donations until the user is on Ethereum and verifies that the `address` owns the github account. Only then can they claim the donations.

```
pragma solidity ^0.4.17;

import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract GithubDonations is usingOraclize {

    struct PendingVerification {
        bool exists;
        address account;
        string username;
    }

    mapping (address => string) public githubUsernames;
    mapping (bytes32 => PendingVerification) pendingVerifications;
    mapping (bytes32 => uint) public donations;

    function __callback(bytes32 queryId, string result) {
```

```
        PendingVerification memory verification =
pendingVerifications[queryId];
        require(verification.exists);
        require(msg.sender == oraclize_cbAddress());
        if(verification.account == parseAddr(result)) {
            githubUsernames[verification.account] =
verification.username;
        }
        delete pendingVerifications[queryId];
    }

    function verify(string gistPath, string username) payable {
        require(oraclize_getPrice("URL") >= msg.value);
        bytes32 id = oraclize_query("URL",
strConcat("https://gist.githubusercontent.com/", username, "/",
gistPath, "verifymyethereumaddress"));
        pendingVerifications[id] = PendingVerification({
            exists: true,
            account: msg.sender,
            username: username
        });
    }

    function donate(string username) payable {
        require(msg.value > 0);
        donations[keccak256(username)] += msg.value;
    }

    function claim() {
        uint donation =
donations[keccak256(githubUsernames[msg.sender])];
        require(donation > 0);
        donations[keccak256(githubUsernames[msg.sender])] = 0;
        msg.sender.call.value(donation)();
    }
}
```

```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";
```

Here, the contract imports the latest version of the Oraclize API from github.

```
contract GithubDonations is usingOraclize { ... }
```

Our contract `GithubDonations` inherits all the members from `usingOraclize` which implements the API and helper functions required to send request to the Oraclize.

Next, we are going to look at the `verify` function which takes `gistPath` which is part of the URL to the gist and `username` which refers to the github username of the sender. Gists can be used to create files on github and we are going to use it to verify the github account. So, if

`https://gist.github.com/asifmallik/31fdef67356a1d2b806093913717de3a/raw/fd416729f8896a1ecc88f78616b1d2782df8f36c/verifymyethereumaddress`

is the URL to the gist, `"asifmallik"` will be the `username` and `"31fdef67356a1d2b806093913717de3a/raw/fd416729f8896a1ecc88f78616b1d2782df8f36c/"` will be the gist part. The username must be passed separately so that the smart contract can store it. `"verifymyethereumaddress"` is excluded from the `gistPath` so that malicious users cannot use gist files with `address` made for other intent for verifying them.

```
require(oraclize_getPrice("URL") >= msg.value);
```

This line checks if the Ether sent to the function is equal or greater than the price of sending a `"URL"` request to oraclize. Oraclize charges a fee for each request made as they need to finance their servers and gas cost for receiving requests, making those requests to APIs and then sending the results back to the smart contracts.

```
bytes32 id = oraclize_query("URL",
strConcat("https://gist.githubusercontent.com/", username, "/", gistPath,
"verifymyethereumaddress"));
```

`oraclize_query` can be used for various types of requests to Oraclize. The first argument specifies the type of request we want Oraclize to make on our behalf, which in this case is an HTTP request, so we pass in `"URL"`. The second argument will hold the data for the request, which, in this case is our URL. `strConcat` is a helper function in `usingOraclize` that concatenates `string` together. `oraclize_query` will return a `byte32` data which will refer to the id of the request that has been made.

```
pendingVerifications[id] = PendingVerification({
    exists: true,
    account: msg.sender,
    username: username
});
```

This maps the `id` to a `struct` containing the `username` and the `address` of the person to check against the data in the URL in `__callback`.

```
function __callback(bytes32 queryId, string result) { ... }
```

The request is made and processed asynchronously, as it takes place off-chain. After it is done, oraclize calls the function `__callback` with `result` which is the data from the URL and the `queryId` is passed along with it to identify which request the data is for. In this case, if verification is successful, the `result` will be a `string` containing the address.

```
PendingVerification memory verification =
pendingVerifications[queryId];
require(verification.exists);
require(msg.sender == oraclize_cbAddress());
```

First, we get the corresponding `PendingVerification` for the `queryId` and ensure it exists. If it does, then we check if oraclize has sent this request by calling `oraclize_cbAddress()` which returns the address of oraclize account. This ensures a malicious user does not send a call to the function.

```
if(verification.account == parseAddr(result)) {
    githubUsernames[verification.account] = verification.username;
}
delete pendingVerifications[queryId];
```

Next, we check if the `result` has an `address` equal to the account that made the verification request previously. If so, we then set the github username for that `address` to the username they claimed initially. Now, they can claim the donations by calling the `claim` function. Finally, the `PendingVerification` for the `queryId` is deleted to ensure it is not called again by oraclize, maliciously or mistakenly.

JSON and XML parsing

Since most APIs returns data in the form of JSON and XML, oraclize supports parsing both types of data and return the necessary piece of data as request by the smart contract. This is because `string` manipulation within EVM costs a lot of gas. `http://api.fixer.io/latest?symbols=USD,GBP` returns

```
{
    "base": "EUR",
    "date": "2017-10-06",
    "rates": {
        "GBP": 0.89535,
        "USD": 1.1707
    }
}
```

If we want to get the rate of `GBP`, that is `"0.89535"` from our smart contract, we send the following query.

```
oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD,GBP).rates.GBP");
```

`json(url)` parses the content of the `url` by a JSON parser, then, `.rates.GBP` gets the `rates` child followed `GBP` which is passed as a `string` to `__callback`. To parse XML, `xml()` is used.

Delays

We could send request to oraclize to schedule a query in the future. If we were to set a delay of one day for the above query, we would run

```
oraclize_query(86400, "URL", "json(http://api.fixer.io/latest?symbols=USD,GBP).rates.GBP");
```

The first argument here refers to the number of seconds after which the request will be made to the URL (60 *60* 24).

Check out the documentation for more information about the oraclize API.

Security

It is important to keep in mind that we are essentially trusting Oraclize to provide the correct data from the HTTP request. While there are many safeguards in place, such as TLSNotary proofs, which can be checked to see whether Oraclize has sent the correct request, the proofs are verified off-chain. So, there must be some method in the smart contract to respond to this. For example, an `owner` could be set for the contract account who can turn it off if oraclize start sending malicious callbacks. However, this also leads to more centralization as the `owner` themselves can be malicious and turn it off for no reason.

Moreover, we are also trusting the APIs to send back reliable information. They could also respond with false information in order to manipulate the smart contracts. This can be overcome by sending requests to multiple APIs and reaching a consensus as it is much less likely for multiple APIs to collude to manipulate smart contracts.

Another issue is that Oraclize or the APIs, unlike our smart contracts, may suspend its operation in the future, so the contracts must be prepared to deal with such a scenario.

Verifying signatures

Verifying signatures

Verifying signatures

Every externally owned account has a private key and public address associated with it. The private key can be used to sign data to get a signature. Anyone with the signature and the data can then process them to get the address. Thus, they can verify that whoever owns that address has signed the data. This can be done in smart contracts using a function called `ecrecover`.

One possible application of this is microtransactions. Right now, it is not feasible to transfer small amount of Ether directly due to relatively high transaction fee. One possible solution to this is using offchain transactions. For example, a platform like YouTube can exist which requires a small amount of Ether to play videos. Users are expected to watch many videos, as a result need to send many transactions. This will prove be expensive.

Instead, we can have a contract that acts as an escrow. Users can send some Ether to this contract and before watching a video, the platform will ask for a signature for withdrawing the Ether. Upon sending this signature, the platform verifies it and allows user to access the video. At the end of the month, after accumulating all the transfers, the platform can withdraw the

}

sum of transfers from the contract using just one transaction which verifies that the user has signed those microtransactions. This will minimize the transaction fee per microtransaction.

```
pragma solidity ^0.4.17;

contract Channel {
    address public sender;
    address public receiver;
    uint public received;

    function Channel(_sender, _receiver) payable {
        sender = _sender;
        receiver = _receiver;
    }

    function receive(bytes32 hash, uint8 v, bytes32 r, bytes32 s, uint
value) {
        bytes32 messageHash = keccak256(this, value);
        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 prefixedHash = keccak256(prefix, messageHash);
        require(prefixedHash == hash);

        address signer = ecrecover(hash, v, r, s);
        require(signer == sender);

        require(value > received);
        uint receivingValue = value - received;
        received = value;
        receiver.transfer(receivingValue);
    }

    function() payable { }
}
```

The constructor function takes two arguments and sets two state variables, `sender` who sends microtransactions and `receiver` who receives them. Payment into the `Channel` can be done either through the fallback function or by sending Ether directly into the constructor function.

`receive` has the bulk of the functionality of the contract. It takes several arguments. `hash` is the message that was signed by the private key. `v` , `r` and `s` contains data regarding the signature. `value` is the amount of Ether `receiver` is attempting to get.

`bytes32 messageHash = keccak256(this, value);`

Usually when signing a message with Ethereum, we sign its `keccak256` hash. The message here is the address of the contract concatenated with the total amount of Ether being transferred.

```
bytes memory prefix = "\x19Ethereum Signed Message:\n32";
bytes32 prefixedHash = keccak256(prefix, messageHash);
require(prefixedHash == hash);
```

After the message hash is passed to the Ethereum client, it is prepended by `\x19Ethereum Signed Message:\n32` to ensure DApps cannot abuse it to sign messages for other purposes not intended by the user. The new hash `prefixedHash` is what is signed by the private key. This is then checked against the `hash` to ensure the user passed in the correct hash that was signed by the `sender` .

```
address signer = ecrecover(hash, v, r, s);
```

```
require(signer == sender);
```

The data signed, `hash`, is then passed along with the signature data, `v, r, s` into `ecrecover` which returns the address of the account whose private key was used to sign the data. Then, finally, we can verify that the `signer` is indeed the `sender`.

```
require(value > received);
uint receivingValue = value - received;
received = value;
receiver.transfer(receivingValue);
```

Since we would like to make `Channel` reusable, we have to be able to call `receive` multiple times. Each time `sender` sends a microtransaction to the `receiver`, they add the amount of Ether being transferred to the sum of Ether transferred in the past. However, this would allow `sender` to call `receive` with the same arguments again and again until `Channel` is drained. To prevent this, each time `receive` is called, `received` is updated which is the sum of Ether already transferred to the `receiver` on the main Ethereum network (as opposed to offchain transfer). `receivingValue` is then calculated by deducting the `received` amount from `value`. Then, after updating `received`, the `receivingValue` is transferred to `receiver`.

However, this introduces a problem as the user may want to withdraw the funds from the contract if they no longer wants to use the platform. Allowing this may lead to the double spending problem where user can sign microtransactions to the platform and watch the videos then before the platform can settle the payments on the contract, the user may withdraw the funds. We can solve this by having a 24 hours delay for withdrawals allowing the platform to withdraw the sum of microtransactions already signed by the user. This is what we are going to implement next using `startWithdrawal` and `withdraw` functions.

```
pragma solidity ^0.4.17;

contract Channel {
    address public sender;
    address public receiver;
    uint public received;
    uint public disputeTime;
    bool public withdrawalOngoing;
    uint public withdrawalValue;
    uint public withdrawalTime;

    modifier onlySender() {
        require(msg.sender == sender);
        _;
    }

    function Channel(_sender, _receiver, _disputeTime) payable {
        sender = _sender;
        receiver = _receiver;
        disputeTime = _disputeTime;
    }

    function receive(bytes32 hash, uint8 v, bytes32 r, bytes32 s, uint
value) {
        bytes32 messageHash = keccak256(this, value);
        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 prefixedHash = keccak256(prefix, messageHash);
        require(prefixedHash == hash);
```

```
        address signer = ecrecover(hash, v, r, s);
        require(signer == sender);

        require(value > received);
        uint receivingValue = value - received;
        received = value;
        receiver.transfer(receivingValue);
    }

    function initiateWithdrawal(uint value) onlySender {
        withdrawalTime = now + disputeTime;
        withdrawalValue = value;
        withdrawalOngoing = true;
    }

    function withdraw() onlySender {
        require(withdrawalOngoing && now > withdrawalTime);
        withdrawalOngoing = false;
        sender.transfer(withdrawalValue);
        withdrawalValue = 0;
        withdrawalTime = 0;
    }

    function balanceAfterWithdrawal() constant returns (uint) {
        return this.balance - withdrawalValue;
    }

    function() payable { }
}
```

We have added four new state variables. `disputeTime` is set only once in the constructor function. This is the time, in seconds, after calling `startWithdrawal` the `sender` can call `withdraw` in order to get the funds back. One possible value for this is 24 hours. `withdrawalTime` , `withdrawalValue` and `withdrawalOngoing` are used by `withdraw` and `startWithdrawal` to keep track of withdrawals in process. We have also applied the `onlySender` modifier to these two functions which ensures only the `sender` can call them successfully.

`startWithdrawal` must be called before calling `withdraw` . It sets `withdrawalTime` by adding `disputeTime` to `now` thereby giving `receiver` enough time to settle payments before `sender` can call `withdraw` . Other than this, `withdrawalOngoing` is set to `true` and the value of the withdrawal is stored in `withdrawalValue` .

After the `disputeTime` is over, `send` calls `withdraw` and all the withdrawal related state variables are erased and the transfer takes place, if there's enough Ether left in the contract.

Between calling `startWithdrawal` and `withdraw` , the `sender` may sign new microtransactions. `receiver` then has to decide whether to accept these as `sender` may withdraw the fund before they can call `receive` . `receiver` can call `balanceAfterWithdrawal` to check whether the `value` (after deducting `received`) signed by the `sender` exceeds this. If it does, the microtransaction can be rejected.

Deploying and Interacting with Contracts

Setting up the Environment

Installing Node.js and npm

Node.js is a JavaScript interpreter and `npm` is its package manager. We will need them to install and run `truffle` as it is written in JavaScript.

Windows and MacOS

First, download the current version of Node.js Windows/MacOS Installer from here. Next, install Node.js. `npm` will also be installed along with it.

Installing truffle

`truffle` is the development framework we are going to be using throughout this book. It will take care of creating initial boilerplate code, deployment and testing for our smart contracts.

```
npm install -g truffle
```

JavaScript Promises

Before writing deployment and interaction scripts for smart contracts, we must know what JavaScript promises are and how they work as we will use them extensively. For those who know about JavaScript promises and `async` and `await` keywords, this section can be skipped.

Promises are used wherever asynchronous operations take place such as HTTP requests, timeouts and I/O. It is an alternative to callback functions. Promises are objects returned by many such functions. At first, a promise will be pending, which means the task is not done yet. Eventually it is going to be either fulfilled, where the task has been completed, or rejected, where the task has failed. We can attach `.then()` for handling fulfilled promises and `.catch()` for handling rejection.

```
getDataFromServer("username").then(function(username) {
    console.log("Hello " + username + "!");
}).catch(function() {
    console.log("Could not connect to server :(");
});
```

Here, `getDataFromServer` returns a promise that is fulfilled when the server responds with `username`. When it does, the function passed into `.then()` is called with the `username` as argument, outputting `Hello asifmallik!`.

However, if there is no internet connection, the HTTP request fails and as a result, the function passed into `.catch()` is called. `.catch()` and `.then()` can be chained together as calling one returns another promise.

Chaining

```
getDataFromServer("username").then(function(username) {
    console.log("Hello " + username + "!");
    getDataFromServer("user_data", username).then(function(data) {
        console.log("Your age is " + data.age);
    });
});
```

When an asynchronous function depends on another one, we can carry it out like above. However, this will become problematic when there are many consecutive asynchronous functions to call. Instead we will use chaining, where we return a promise to our `.then()` function.

```
getDataFromServer("username").then(function(username) {
    console.log("Hello " + username + "!");
    return getDataFromServer("user_data", username);
}).then(function(data) {
    console.log("Your age is " + data.age);
});
```

The second `.then()` is called on the promise returned by `getDataFromServer("user_data", username)`, so it is only fulfilled when the server responds with user data.

Running an Ethereum client

Here we are going to take a look at Ethereum clients for different networks.

truffle develop

`truffle` can be used to create a private test network for Ethereum which exists only on one computer. In order to do this, open the CLI and run

```
truffle develop
```

Note that this command can only be run inside a `truffle` project directory. We are going to learn how to create one in the next section.

Deploying Smart Contracts

Deploying with Truffle

First, we will a CLI, navigate to our development directory and run

`truffle init`

This will create the following files

```
├── contracts
|   └── Migrations.sol
├── migrations
|   └── 1_initial_migration.js
├── test
├── truffle.js
└── truffle-config.js
```

Now that the boilerplate has been generated, we are going to place our code. First, we are going to replace the contents of `truffle.js` with our own configuration.

```
module.exports = {
    networks: {
        development: {
            host: "localhost",
            port: 9545,
            network_id: "*" // Match any network id
        }
    }
```

```
};
```

This defines the port in which the Ethereum client is running and the network type (whether it is live network or test network). Next, we will create a new solidity file `Greeting.sol` in the `contracts` folder and paste the following code

```
pragma solidity ^0.4.17;

contract Greeting {
    string message;
    address public owner;

    function Greeting() {
        owner = msg.sender;
    }

    function setMessage(string _message) {
        message = _message;
    }

    function getMessage() constant returns (string) {
        return message;
    }
}
```

Now that we have the code for our smart contract, we can go ahead and compile it to artifacts using the command

`truffle compile`

Note that the above command must be run from the `contract` folder. After this, create a new file `migrations/2_deploy_contracts.js` which will contain the code for deploying the contract and replace it with

```
var Contract = artifacts.require("./Greeting.sol");

module.exports = function(deployer) {
  deployer.deploy(Contract);
};
```

Basically, this code imports the artifact that has been compiled from `Greeting.sol` in the `contracts` folder in the previous step and deploys it to the blockchain. Now, to run this, simply use the command

`truffle migrate`

Note that our Ethereum client must be running when we run the above command. This should output something like this

```
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ...
0x2f3166590e022089e8f759640647d90a285772bded90f837ba71228648919b70
  Migrations: 0x9b7269c2093f286ef5b6291acc8d1e86aae84ef1
Saving successful migration to network...
  ...
0x98b177ba6f32d76964b445810c83a233e3f58790b092e72acb5fbdff61481306
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Greeting...
```

```
  ...
0x1262769ccfa6087661d5ca3f251fff53687070ae320ae818ea5e07eb4119da71
  Greeting: 0x08848398f488ffA565B903D84825214168A8B76b
Saving successful migration to network...
  ...
0x4d9cd7f52d7191752bb8b8621060adb1122e807f88c0733d3c1cf9d517160fd3
Saving artifacts...
```

Right after `Greeting:` we will find the address of the deployed contract account of the `Greeting` contract. In this case, it is `0x08848398f488ffA565B903D84825214168A8B76b` which we are going to copy and save somewhere for later use.

Lastly, remember to use `truffle migrate --reset` instead when deploying the same contract after editing it.

Constructor Arguments

Next, we are going to replace `Greeting.sol` with our latest `Ballot.sol`.

The constructor function accepts two arguments, the first is a `string` and the other one is a `uint`. In order to deploy a contract with arguments, the deployment script above must be slightly modified

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

module.exports = function(deployer) {
  deployer.deploy(Ballot, "Should Ethereum implement PoS?", 10);
};
```

Multiple Contracts

Now, we are going to deploy `ShopTokenICO` and for that, we will also need to deploy `ShopToken` in order to get pass its `address` to `ShopTokenICO`. This can be done with one deployment script.

```
var ShopToken = artifacts.require("./ShopToken.sol");
var ShopTokenICO = artifacts.require("./ShopTokenICO.sol");

module.exports = function(deployer) {
  deployer.deploy(ShopToken, "1000000000000000000").then(function(){
      return deployer.deploy(ShopTokenICO, ShopToken.address, 100,
"50000000000000000000", "30000000000000000000", "1539475200");
  });
};
```

`deployer.deploy()` returns a promise for the deployment. After deployment is done, `ShopToken.address` is set to the `address` of the deployed `ShopToken`. Now, this can be passed to the constructor of `ShopTokenICO`. The promise must be returned so that truffle can save the address of the ShopTokenICO to the artifacts.

Converting between units of Ether

Above, we have used large numbers such as `"50000000000000000000"` and `"30000000000000000000"`. However, this makes the code less readable as it is not immediately obvious the amount of Ether to be raised. To make the code easier to read, we will instead use

```
var ShopToken = artifacts.require("./ShopToken.sol");
var ShopTokenICO = artifacts.require("./ShopTokenICO.sol");

module.exports = function(deployer) {
  deployer.deploy(ShopToken, "1000000000000000000").then(function(){
      return deployer.deploy(ShopTokenICO, ShopToken.address, 100,
web3.toWei(500, "ether"), web3.toWei(300, "ether"), "1539475200");
  });
};
```

`web3.toWei(500, "ether")`

This converts 500 Ether to wei by adding 18 zeros after it. Other units can be converted to wei too by changing the second parameter such as, to `"finney"`. `web3.fromWei()` can be used to do the opposite, converting wei to the desired unit.

Inheritance

The deployment script for contracts with inheritance is identical to those without it. This is because only a single contract is being deployed. All the code from the parent contracts are copied onto the child contract.

Interacting with Smart contracts

With Truffle and JavaScript

Now, we are going to see whether the deployment has taken place. Run the command

`truffle console`

to start interacting with the deployed smart contract. `truffle` uses JavaScript, so all interaction here will take place in it. First of all, we will run

```
Greeting.deployed().then(function (greeting) {
    return greeting.setMessage("Hello World!")
}).then(function () {
    return greeting.getMessage();
}).then(function (message) {
    console.log(message);
});
```

`Greeting.deploy()` returns a promise for the deployed `Greeting` contract instance. The `greeting` object passed into the function is the instance of `Greeting`. All the functions in the contract can be accessed as methods on the instance. When `greeting.setMessage("Hello World!")` is called, it sends a request to the Ethereum client to call the corresponding function in the contract account with the argument `"Hello World!"`. Then, it returns a promise which will be fulfilled once the contract function is called successfully. Once `setMessage` is called and `message` is set to `"Hello`

`World!"`, we will call `greeting.getMessage()` which will again return a promise. This promise, once fulfilled, passes the `message` from the contract, which is then logged.

Alternatively, we can also create a separate file `message.js` and wrap the above code to get the following

```
module.exports = function(callback) {
    var Greeting = artifacts.require("./Greeting.sol");
    Greeting.deployed().then(function (greeting) {
        return greeting.setMessage("Hello World!")
    }).then(function () {
        return greeting.getMessage();
    }).then(function (message) {
        console.log(message);
    });
    callback();
}
```

and then run

`truffle exec message.js`

Whichever method is used, the final output ought to be

`Hello World!`

Now, if we run the same code again without `setMessage` like this

```
Greeting.deployed().then(function (greeting) {
    return greeting.getMessage();
}).then(function (message) {
    console.log(message);
```

```
});
```

the output will still be the same. This is to demonstrate that the state of the smart contract is persistent.

Accessing Public Variables

Since `owner` is a `public` variable, a getter function of the same name is generated. So, we can access it with the following script.

```
Greeting.deployed().then(function (greeting) {
    return greeting.owner();
}).then(function(owner) {
    console.log(owner);
});
```

Accounts

Accounts addresses can be access using `web3.eth.accounts`. It is an array with all the account addresses available. `web3.eth.accounts[0]` is the default account used to deploy and call functions unless specified otherwise.

```
Greeting.deployed().then(function (greeting) {
    return greeting.owner();
}).then(function(owner) {
    console.log(owner == web3.eth.accounts[0]);
});
```

This is going to log `true`.

Constant and non-constant functions

It is important to distinguish between the two different types of functions. `getMessage` is a `constant` function, which means that, when called, no changes are made to the state variables, hence the blockchain. This means, when called, no transaction takes place and costs no gas, as it is only run in user's Ethereum client. On the other hand, when `setMessage` is called, `message` is modified and as a result, a transaction is sent. Thus, `getMessage` will be processed immediately and the callback function will be fired soon but with `setMessage`, the callback function is fired with some delay as the transaction needs to be verified. When a transaction is sent, the value from the function is not passed into the callback function, rather the transaction data is.

```
ShopToken.deployed().then(function (shopToken) {
    return shopToken.approve(web3.eth.accounts[1], 1000, {
        from: web3.eth.accounts[0]
    });
}).then(function (transactionData) {
    console.log(transactionData);
});
```

This will log out something like

```
{ tx:
'0x0d8e94248f0827734ee8d44e19ad06d343b1e980c6960e2123f5a46838181f1b',
  receipt:
   { transactionHash:
'0x0d8e94248f0827734ee8d44e19ad06d343b1e980c6960e2123f5a46838181f1b',
     transactionIndex: 0,
```

```
     blockHash:
'0x181162fa9bdc1ba6bb7485b24e80d01f4a0d4fe88b1dbc72c61ca96b3932f094',
     blockNumber: 15,
     gasUsed: 45289,
     cumulativeGasUsed: 45289,
     contractAddress: null,
     logs: [ [Object] ] },
  logs:
   [ { logIndex: 0,
       transactionIndex: 0,
       transactionHash:
'0x0d8e94248f0827734ee8d44e19ad06d343b1e980c6960e2123f5a46838181f1b',
       blockHash:
'0x181162fa9bdc1ba6bb7485b24e80d01f4a0d4fe88b1dbc72c61ca96b3932f094',
       blockNumber: 15,
       address: '0xf9cc04a3cd5419368d9e959ab174835e9746e8b1',
       type: 'mined',
       event: 'Approval',
       args: [Object] } ] }
```

`transactionData.tx` gives the transaction hash which is used to identify the transaction. `transactionData.receipt` gives other data about the transaction such as `gasUsed` and `blockNumber`. `transactionData` gives an array of logs, each of which is an object with `event` and `args` properties.

While in the above method, the value `true` is not returned, there is one way to get the value returned from a non-constant function. This is by calling the `.call()` method instead.

```
ShopToken.deployed().then(function (shopToken) {
    return shopToken.transfer.call(web3.eth.accounts[0], 1000, {
        from: web3.eth.accounts[1]
```

```
    });
}).then(function (success) {
    if (success) {
        console.log("1000 Shop Tokens can be transferred.");
    }else{
        console.log("Shop Tokens transfer will fail.");
    }
});
```

In this case, no transaction is sent and the `transfer` function is only executed locally. Thus, the blockchain is not permanently modified and the callback function is fired immediately. If the second account does not have 1000 Shop Tokens, the script will log

```
Shop Tokens transfer will fail.
```

Within Deployment Script

Often, it is necessary to call contract functions in the deployment script as part of configuring the contract. Now, we are going to modify the `ShopTokenICO` migration script in order to transfer shop tokens to the ICO and then start it.

```
var ShopToken = artifacts.require("./ShopToken.sol");
var ShopTokenICO = artifacts.require("./ShopTokenICO.sol");

module.exports = function(deployer) {
  deployer.deploy(ShopToken, 50000).then(function(){
      return deployer.deploy(ShopTokenICO, ShopToken.address, 100,
web3.toWei(500, "ether"), web3.toWei(300, "ether"), "1539475200");
  }).then(function () {
      return ShopToken.deployed();
```

```
  }).then(function (shopToken) {
      return shopToken.transfer(ShopTokenICO.address, 50000);
  }).then(function () {
      return ShopTokenICO.deployed();
  }).then(function (shopTokenICO) {
      return shopTokenICO.begin();
  });
};
```

After deployment of both `ShopToken` and `ShopTokenICO`, the instance of `ShopToken` is requested by `ShopToken.deployed()`. Next, using the instance of `ShopToken` we transfer 500 shop tokens to the `ShopTokenICO` by passing its `address` to the `transfer` function. Next, we request the instance of `ShopTokenICO` and call the `being` function in order to start the ICO.

Sending Ether to a function

```
ShopTokenICO.deployed().then(function(shopTokenICO) {
    return shopTokenICO.buy(web3.eth.accounts[0], {value:
"1000000000000000000"});
});
```

In order to send Ether to a function, after the arguments for the contract function, another argument is passed. This argument is an object with a property `value` to which the amount of Ether to transfer in wei is assigned. Here, it sends 1 Ether.

To call any function from another account, we specify another property `from` to which the `address` of the account to call from is assigned. Note that only accounts unlocked in the Ethereum client can be used to call functions and transfer Ether.

```
ShopTokenICO.deployed().then(function(shopTokenICO) {
    return shopTokenICO.buy(web3.eth.accounts[1], {
        value: "1000000000000000000",
        from: web3.eth.accounts[1]
    });
});
```

This sends 1 Ether to the `buy` function of `ShopTokenICO` from the second account to which 100 shop tokens are sent.

Catching Errors

When calling a contract function, errors may arise in many forms such as when `require` conditions fail or when all gas runs out. When this happens, the callback function passed into `.then()` is not fired, rather, the callback function passed into `.catch()` is fired, passing the error message into it.

```
ShopTokenICO.deployed().then(function(shopTokenICO) {
    return shopTokenICO.sell(web3.eth.accounts[1], 100, {
        from: web3.eth.accounts[1]
    });
}).catch(function (err) {
    console.log(err);
});
```

After deploying `ShopTokenICO` using our latest deployment script, running this will fail as `deadline` is not reached yet. As a result, it logs the error

`Error: VM Exception while processing transaction: invalid opcode`

due to `require(deadline >= now);` in `beforeDeadline` modifier.

Big number

JavaScript only supports integers up to 9007199254740991. However, this is not enough for working with smart contracts as, in Solidity, integers can be much larger. As a result, when integers are returned from smart contracts, they are represented by a data type called `BigNumber` which can store and manipulate arbitrarily large integers.

```
ShopToken.deployed().then(function(shopToken) {
    shopToken.balanceOf(ShopTokenICO.address).then(function(balance) {
        console.log(balance); //BigNumber format
        console.log(balance.valueOf()); //String format

        var anotherBigNumber = balance.plus(balance); //adding
BigNumbers
        console.log("New BigNumber: " + anotherBigNumber.valueOf());
    });
});
```

This will log out

```
{ [String: '50000'] s: 1, e: 4, c: [ 50000 ] }
50000
New BigNumber: 100000
```

```
var anotherBigNumber = balance.plus(balance);
```

This is how we add `BigNumber`. It is the equivalent of `var anotherNumber = balance + balance;` in terms of regular `Number`. Below, there is a list of methods available for `BigNumber` under "BigNumber.prototype methods"

BigNumber.prototype methods

absoluteValue	abs	isNegative	isNeg	toDigits	
ceil		isZero		toExponential	
comparedTo	cmp	lessThan	lt	toFixed	
decimalPlaces	dp	lessThanOrEqualTo	lte	toFormat	
dividedBy	div	minus	sub	toFraction	
dividedToIntegerBy	divToInt	modulo	mod	toJSON	
equals	eq	negated	neg	toNumber	
floor		plus	add	toPower	pow
greaterThan	gt	precision	sd	toPrecision	
greaterThanOrEqualTo	gte	round		toString	
isFinite		shift		truncated	trunc
isInteger	isInt	squareRoot	sqrt	valueOf	
isNaN		times	mul		

BigNumber.config properties		BigNumber methods		BigNumber properties	
DECIMAL_PLACES	20	another		ROUND_UP	0
ROUNDING_MODE	4	config	set	ROUND_DOWN	1
EXPONENTIAL_AT	[-7, 20]	max		ROUND_CEIL	2
RANGE	1e+7	min		ROUND_FLOOR	3
ERRORS	true	random		ROUND_HALF_UP	4
CRYPTO	false			ROUND_HALF_DOWN	5
MODULO_MODE	1			ROUND_HALF_EVEN	6
POW_PRECISION	0			ROUND_HALF_CEIL	7
FORMAT	{}			ROUND_HALF_FLOOR	8
				EUCLID	9

Writing Tests for Smart Contracts

Unit testing is an essential part of programming. It is especially important for smart contract development because

- updating smart contracts are expensive and difficult if not impossible, so it must be ensured they work properly when deployed the first time as much as possible.
- smart contracts involves value transfer in the form of Ether and other tokens. Moreover, its decentralized nature means all of its data are permanent and cannot be fixed by a central authority (unless preconfigured). Therefore, a bug can cause devastating effects by allowing malicious users to manipulate information and steal funds.

Proper unit testing ensures the smart contract works as expected automatically. Therefore, it allows rapid development, as developers can easily check if the smart contract still works as expected after introducing changes to it.

We are going to be using the `truffle` framework for running our unit tests. Tests can be written in either JavaScript or Solidity. In this chapter, we will cover both of these extensively by writing tests for `Crowdfund`.

Unit Testing with JavaScript

For this, we are going to first go to the `test` directory of our project folder. Here, we will create a new file `crowdfund.js` where we are going to write our unit tests.

```
var Crowdfund = artifacts.require("Crowdfund");

contract("Crowdfund", function (accounts) {
    it("should have verifier set correctly", function () {
        var crowdfund;
        return Crowdfund.deployed().then(function (instance) {
            crowdfund = instance;
            return crowdfund.verifier();
        }).then(function (verifier) {
            assert.equal(verifier, accounts[0], "The first account was
not the verifier");
        });
    });
});
```

This is a basic very basic test checking whether `verifier` and `owner` is set to the correct addresses.

`contract("Crowdfund", function (accounts) { ... });`

The `contract` function is similar to the `describe` function in mocha. The first argument describes what is being tested, which in this case is the `Crowdfund` contract. The second argument is a function which contains all the tests for it. Every time `contract` is called, the deployment script is run again so that the tests within the callback can work with a fresh deployment of the contract. `accounts` is passed into the callback function which is an array of the account addresses that can be accessed by the Ethereum client.

```
it("should have verifier set correctly", function () { ... });
```

The `it` function describe individual test within `contract` callback. The first argument describes the particular test, whereas the second argument is a callback function which contains the actual logic of the test. This particular test will check whether the `verifier` is set correctly to `accounts[0]`.

```
var crowdfund;
return Crowdfund.deployed().then(function (instance) { ...
}).then(function (verifier) { ... });
```

`Crowdfund.deployed()` is called in order to get the deployed instance of `Crowdfund`. The promise is returned to the callback function because this test is asynchronous. As a result, the mocha framework needs to know when the test is complete so that it can proceed to the next.

```
assert.equal(verifier, accounts[0], "The first account was not the
verifier");
```

After getting the `verifier` variable by called `crowdfund.verfier()`, we check if this is equal to `accounts[0]` by passing both of them to `assert.equal`. Finally, we pass a third variable which is an error message that is logged when the test is run and the assertion fails. It lets the developer know what is wrong.

Next, we will run the following in the command line in order to run the tests

```
truffle test
```

The output will be similar to

```
Contract: Crowdfund
    √ should have verifier set correctly


  1 passing (117ms)
```

indicating all the tests have been passed in 117 milliseconds. Next, we are going to change second argument of `assert.equal` to `accounts[1]` and re-run the test to see what happens when the test fails

```
Contract: Crowdfund
   1) should have verifier set correctly
   > No events were emitted


 0 passing (57ms)
 1 failing

 1) Contract: Crowdfund should have verifier set correctly:
    AssertionError: The first account was not the verifier: expected
'0x47adc0faa4f6eb42b499187317949ed99e77ee85' to equal
'0x4ef9e4721bbf02b84d0e73822ee4e26e95076b9d'
     at E:\Programming\ethereum-book\crowdfund\test\crowdfund.js:10:20
     at <anonymous>
     at process._tickCallback (internal/process/next_tick.js:188:7)
```

It outputs the error message `Crowdfund should have verifier set correctly` along with the values expected and received.

```
var Crowdfund = artifacts.require("Crowdfund");

contract("Crowdfund", function (accounts) {
    it("should have verifier and receiver set correctly", function ()
{
        var crowdfund;
        return Crowdfund.deployed().then(function (instance) {
            crowdfund = instance;
            return crowdfund.verifier();
        }).then(function (verifier) {
            assert.equal(verifier, accounts[0], "The first account was
not the verifier");
            return crowdfund.receiver();
        }).then(function (receiver) {
            assert.equal(receiver, accounts[1], "The second account
was not the receiver");
        });
    });
});
```

Here we have added another `assert.equal` for `owner` to check if that has been set properly as well. This illustrates that we can have multiple assertion in a single test.

```
var Crowdfund = artifacts.require("Crowdfund");

contract("Crowdfund", function (accounts) {
    it("should have verifier and receiver set correctly", function ()
{
```

```
        var crowdfund;
        return Crowdfund.deployed().then(function (instance) {
            crowdfund = instance;
            return crowdfund.verifier();
        }).then(function (verifier) {
            assert.equal(verifier, accounts[0], "The first account was
not the verifier");
            return crowdfund.receiver();
        }).then(function (receiver) {
            assert.equal(receiver, accounts[1], "The second account
was not the receiver");
        });
    });

    it("should keep track of funds", function () {
        var crowdfund;
        return Crowdfund.deployed().then(function (instance) {
            crowdfund = instance;
            return crowdfund.fund({
                from: accounts[3],
                value: web3.toWei(1, "ether")
            });
        }).then(function () {
            return crowdfund.funds(accounts[3]);
        }).then(function (fund) {
            assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
        });
    });
});
```

Here, we introduced another test which first calls the `fund` function with 1 Ether and later checks whether the contract has recorded the 1 Ether contributed by that account. This is done by calling the `funds` function which accesses

```
mapping (address => uint) public funds;
```

It is important the contract keeps track of funds properly because it will be used to refund the users if the crowdfund fails.

```
assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The contract did not keep track of funds properly");
```

Here, we passed `fund.valueOf()` instead of passing `fund` itself even though the latter works because when the assertion fails, it would output a `BigNumber` object which is not in a human readable format.

However, will have many different tests for the `Crowdfund` contract, so it does not make sense to call `Crowdfund.deployed()` each time. This is where the `before` function comes in.

```
var Crowdfund = artifacts.require("Crowdfund");

contract("Crowdfund", function (accounts) {
    var crowdfund;
    before(function () {
        return Crowdfund.deployed().then(function (instance) {
            crowdfund = instance;
        });
    });

    it("should have verifier and receiver set correctly", function ()
{
        return crowdfund.verifier().then(function (verifier) {
            assert.equal(verifier, accounts[0], "The first account was
not the verifier");
            return crowdfund.receiver();
```

```
        }).then(function (receiver) {
            assert.equal(receiver, accounts[1], "The second account
was not the receiver");
        });
    });

    it("should keep track of funds", function () {
        return crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        }).then(function () {
            return crowdfund.funds(accounts[3]);
        }).then(function (fund) {
            assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
        });
    });
});
```

The callback function passed into `before` function is fired once before running any test. Here, we declare a variable `crowdfund` directly under the `contract` callback function so that it can be accessed from each test instead. In the `before` function we return a promise which is resolved after `crowdfund` is set to the deployed instance, so that mocha can proceed to running the tests.

async and await

Currently, we are using `then` method for dealing with promises. However, this makes the code less readable and large. This can be solved by the `async` and `await` keywords. These two keywords have been introduced by

ECMAScript 6 standard and it has been already implemented in Node.js. They make it easier to work with promises.

```
async function getX () {
    return 100;
}

getX().then(function (x) {
    console.log(x);
});
```

Functions can be prepended by `async` to make them asynchronous. `async` functions return a promise that is resolved when the function returns a value. Hence, the above `getX` is equivalent to

```
function getX () {
    return new Promise(function (resolve) {
        resolve(100);
    });
}
```

However, `async` is not very useful without the `await` keyword. `await` can only be used within an `async` function before a promise. This will pause the function until the promise is resolved and evaluate to the value resolved by the promise.

```
async function addToX (y) {
    var x = await getX();
    return x + y;
}
```

```
addToX(10).then(function (val) {
    console.log(val);
});
```

This will log out 110. In terms of promises, the above `addToX` will be equivalent to

```
function addToX (y) {
    return new Promise(function (resolve) {
        getX().then(function (x) {
            resolve(x + y);
        });
    });
}
```

With `async` and `await` we can make our tests much smaller and more readable.

```
var Crowdfund = artifacts.require("Crowdfund");

contract("Crowdfund", function (accounts) {
    var crowdfund;
    before(async function () {
        crowdfund = await Crowdfund.deployed();
    });

    it("should have verifier and receiver set correctly", async
function () {
        var verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not
the verifier");
```

```
        var receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was
not the receiver");
    });

    it("should keep track of funds", async function () {
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });

        var fund = await crowdfund.funds(accounts[3]);
        assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
    });
});
```

Lambda functions, const and let

Since we are using `async` and `await` which are part of ECMAScript 6 standard, it is also conventional to use lambda functions, `const` and `let` along with them. While there are differences in way each works we will not get into the details here. `const` and `let` can be thought of as alternatives for `var` . `const` variables are immutable and `let` is used in cases where the variable needs to be changed.

```
let add = function (a, b) {
    return a + b;
}

let addInLamba = (a, b) => {
    return a + b;
```

```
}
```

The above demonstrates the syntax of lambda functions which are not very different from `function`. Now, we are going to modify out code to incorporate these.

```
const Crowdfund = artifacts.require("Crowdfund");

contract("Crowdfund", (accounts) => {
    let crowdfund;
    before(async () => {
        crowdfund = await Crowdfund.deployed();
    });

    it("should have verifier and receiver set correctly", async () => {
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not the verifier");
        let receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was not the receiver");
    });

    it("should keep track of funds", async () => {
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });

        let fund = await crowdfund.funds(accounts[3]);
        assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The contract did not keep track of funds properly");
```

```
    });
});
```

Testing balance

Next, we are going to test the `approve` function. For this we are going to have to ensure all the balance in Ether of the `Crowdfund` is transferred to the `receiver`. This can be done using the `web3.eth.getBalance` function. However, this function does not return a promise, rather it takes a callback function as an argument to which the balance is passed in. So, we are going to wrap it into a separate function `getBalance` which will return a promise for the balance.

```
const Crowdfund = artifacts.require("Crowdfund");

let getBalance = (account) => {
    return new Promise((resolve, error) => {
        web3.eth.getBalance(account, (err, balance) => {
            if (err) {
                error(err);
            } else {
                resolve(balance);
            }
        });
    });
};

contract("Crowdfund", (accounts) => {
    let crowdfund;
    before(async () => {
        crowdfund = await Crowdfund.deployed();
    });
```

```
    it("should have verifier and receiver set correctly", async () =>
{
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not
the verifier");
        let receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was
not the receiver");
    });

    it("should keep track of funds", async () => {
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });

        let fund = await crowdfund.funds(accounts[3]);
        assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
    });

    it("should release funds during approval", async () => {
        let initialReceiverBalance = await getBalance(accounts[1]);
        let initialContractBalance = await
getBalance(crowdfund.address);

        await crowdfund.approve(true, {
            from: accounts[0]
        });

        let finalReceiverBalance = await getBalance(accounts[1]);
        let finalContractBalance = await
getBalance(crowdfund.address);

        assert.equal(finalContractBalance.valueOf(), 0, "The full
```

```
balance of the contract was not transferred");

assert.equal(finalReceiverBalance.minus(initialReceiverBalance).valueO
f(), initialContractBalance.valueOf(), "The receiver did not receive
the funds from the contract");
    });
});
```

In the new test we introduced above, we have first saved the initial balances of both the contract account and the `receiver`. Then, we called the `approve` function from the `verifier` account with `true` as the argument. As a result, if `Crowdfund` works correctly, it should transfer the funds to the `receiver`. After this transaction, we again called `getBalance` for both accounts. Finally, we made two assertions. The first one ensures that no Ether is left in the contract account. The second one checks whether all initial balance of the contract is equal to the difference in the initial and final balances of the `receiver` account. This ensures that the balance is transferred to the `receiver` and not another account.

Deploying new instance

Now, we would like to test the `refund` function of the contract. However, the `Crowdfund` has already been approved and there is no way of undoing this. As a result, new contract must be deployed. In fact, it is a good practice to deploy a new contract for every test to ensure that they are isolated and not interdependent on each other. Otherwise, changing the order or the tests themselves may lead to the tests breaking.

```
const Crowdfund = artifacts.require("Crowdfund");
```

```
let getBalance = (account) => {
    return new Promise((resolve, error) => {
        web3.eth.getBalance(account, (err, balance) => {
            if (err) {
                error(err);
            } else {
                resolve(balance);
            }
        });
    });
};

contract("Crowdfund", (accounts) => {

    it("should have verifier and receiver set correctly", async () => {
        let crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not the verifier");
        let receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was not the receiver");
    });

    it("should keep track of funds", async () => {
        let crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });
        let fund = await crowdfund.funds(accounts[3]);
        assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The contract did not keep track of funds properly");
    });
```

```
    it("should release funds during approval", async () => {
        let crowdfund = await Crowdfund.new(accounts[0], accounts[1]);

        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });

        let initialReceiverBalance = await getBalance(accounts[1]);
        let initialContractBalance = await
getBalance(crowdfund.address);

        await crowdfund.approve(true, {
            from: accounts[0]
        });

        let finalReceiverBalance = await getBalance(accounts[1]);
        let finalContractBalance = await
getBalance(crowdfund.address);

        assert.equal(finalContractBalance.valueOf(), 0, "The full
balance of the contract was not transferred");

assert.equal(finalReceiverBalance.minus(initialReceiverBalance).valueO
f(), initialContractBalance.valueOf(), "The receiver did not receive
the funds from the contract");
    });

    it("should allow refunds when not approved", async () => {
        let crowdfund = await Crowdfund.new(accounts[0], accounts[1]);

        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });
```

```
        await crowdfund.fund({
            from: accounts[4],
            value: web3.toWei(2, "ether")
        });

        let initialBalance = await getBalance(accounts[5]);
        await crowdfund.approve(false, {
            from: accounts[0]
        });
        await crowdfund.refund(accounts[5], {
            from: accounts[3]
        });
        let finalBalance = await getBalance(accounts[5]);

        assert.equal(finalBalance.minus(initialBalance).valueOf(),
web3.toWei(1, "ether").valueOf(), "The funder did not receive the
correct amount of refund");
    });
});
```

Here, we have removed the `before` function and instead, we deploy a new `Crowdfund` instance in every test. So, the contract must be funded each time in both `should keep track of funds` and `should release funds during approval`. Moreover, we have introduced another test to test `refund` function. It does so by funding the contract with two different accounts followed by calling the `refund` function one of two accounts and checking if the correct amount has been refunded.

beforeEach

However, this means there is too much code duplication as we have

```
let crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
```

in every test. To solve this, we are going to use the `beforeEach` function. The callback passed into this function will be fired before every single test. So, inside this callback, we can deploy a new `Crowdfund` instance and assign it to `crowdfund` which is again declared directly in the `contract` callback function.

```
const Crowdfund = artifacts.require("Crowdfund");

let getBalance = (account) => {
    return new Promise((resolve, error) => {
        web3.eth.getBalance(account, (err, balance) => {
            if (err) {
                error(err);
            } else {
                resolve(balance);
            }
        });
    });
};

contract("Crowdfund", (accounts) => {
    let crowdfund;
    beforeEach(async () => {
        crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
    });

    it("should have verifier and receiver set correctly", async () =>
{
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not
the verifier");
        let receiver = await crowdfund.receiver();
```

```
        assert.equal(receiver, accounts[1], "The second account was
not the receiver");
    });

    it("should keep track of funds", async () => {
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });
        let fund = await crowdfund.funds(accounts[3]);
        assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
    });

    it("should release funds during approval", async () => {
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });

        let initialReceiverBalance = await getBalance(accounts[1]);
        let initialContractBalance = await
getBalance(crowdfund.address);

        await crowdfund.approve(true, {
            from: accounts[0]
        });

        let finalReceiverBalance = await getBalance(accounts[1]);
        let finalContractBalance = await
getBalance(crowdfund.address);

        assert.equal(finalContractBalance.valueOf(), 0, "The full
balance of the contract was not transferred");

assert.equal(finalReceiverBalance.minus(initialReceiverBalance).valueO
```

```
f(), initialContractBalance.valueOf(), "The receiver did not receive
the funds from the contract");
    });

    it("should allow refunds when not approved", async () => {
        await crowdfund.fund({
            from: accounts[3],
            value: web3.toWei(1, "ether")
        });
        await crowdfund.fund({
            from: accounts[4],
            value: web3.toWei(2, "ether")
        });

        let initialBalance = await getBalance(accounts[5]);
        await crowdfund.approve(false, {
            from: accounts[0]
        });
        await crowdfund.refund(accounts[5], {
            from: accounts[3]
        });
        let finalBalance = await getBalance(accounts[5]);

        assert.equal(finalBalance.minus(initialBalance).valueOf(),
web3.toWei(1, "ether").valueOf(), "The funder did not receive the
correct amount of refund");
    });
});
```

context

However, there are still a lot of code duplication. We have written the code for funding the contract function in three different tests. This can be solved by using `context`. `context` allows us to create a group of tests and apply `before`, `after`, `beforeEach` and `afterEach` to the group of tasks.

```
const Crowdfund = artifacts.require("Crowdfund");

let getBalance = (account) => {
    return new Promise((resolve, error) => {
        web3.eth.getBalance(account, (err, balance) => {
            if (err) {
                error(err);
            } else {
                resolve(balance);
            }
        });
    });
};

contract("Crowdfund", (accounts) => {
    let crowdfund;
    beforeEach(async () => {
        crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
    });

    it("should have verifier and receiver set correctly", async () =>
{
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not
the verifier");
        let receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was
not the receiver");
    });
```

```
    context("fund contract", () => {
        beforeEach(async () => {
            await crowdfund.fund({
                from: accounts[3],
                value: web3.toWei(1, "ether")
            });
            await crowdfund.fund({
                from: accounts[4],
                value: web3.toWei(2, "ether")
            });
        });

        it("should keep track of funds", async () => {
            let fund = await crowdfund.funds(accounts[3]);
            assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
        });

        it("should release funds during approval", async () => {
            let initialReceiverBalance = await
getBalance(accounts[1]);
            let initialContractBalance = await
getBalance(crowdfund.address);

            await crowdfund.approve(true, {
                from: accounts[0]
            });

            let finalReceiverBalance = await getBalance(accounts[1]);
            let finalContractBalance = await
getBalance(crowdfund.address);

            assert.equal(finalContractBalance.valueOf(), 0, "The full
balance of the contract was not transferred");
```

```
assert.equal(finalReceiverBalance.minus(initialReceiverBalance).valueO
f(), initialContractBalance.valueOf(), "The receiver did not receive
the funds from the contract");
        });

        it("should allow refunds when not approved", async () => {
            let initialBalance = await getBalance(accounts[5]);
            await crowdfund.approve(false, {
                from: accounts[0]
            });
            await crowdfund.refund(accounts[5], {
                from: accounts[3]
            });
            let finalBalance = await getBalance(accounts[5]);

            assert.equal(finalBalance.minus(initialBalance).valueOf(),
web3.toWei(1, "ether").valueOf(), "The refund receiver did not receive
the correct amount of refund");
        });
    })

});
```

The first argument of `context` describes the context of the tests in it. Here, for all the tests, we are funding the contract so we named it `fund contract`. The next argument is the callback function that contains the tests. Inside this, we have a `beforeEach` function with which we fund the contract instance. As a result, for each test in this context, we are not only creating a new instance of the `Crowdfund` contract but also calling the `fund` function from two different accounts with 1 Ether and 2 Ether before running the actual tests.

We can also have `context` inside another `context`. For each test, the `beforeEach`, `before`, `afterEach` and `after` function in all parent `context` all the way up to `contract` applies, starting from the top to the bottom.

Testing error

At times, we expect a contract to throw an error. For example, when any account other than the `verifier` attempts to `approve` the `Crowdfund`, it should be prevented. When we run `await promise;` it throws an error if the promise leads to an error. Therefore we are going to use `try { ... } catch (err) { ... }` to deal with this.

```
const Crowdfund = artifacts.require("Crowdfund");

let getBalance = (account) => {
    return new Promise((resolve, error) => {
        web3.eth.getBalance(account, (err, balance) => {
            if (err) {
                error(err);
            } else {
                resolve(balance);
            }
        });
    });
};

contract("Crowdfund", (accounts) => {
    let crowdfund;
    beforeEach(async () => {
        crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
    });
```

```
    it("should have verifier and receiver set correctly", async () =>
{
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not
the verifier");
        let receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was
not the receiver");
    });

    context("fund contract", () => {
        beforeEach(async () => {
            await crowdfund.fund({
                from: accounts[3],
                value: web3.toWei(1, "ether")
            });
            await crowdfund.fund({
                from: accounts[4],
                value: web3.toWei(2, "ether")
            });
        });

        it("should keep track of funds", async () => {
            let fund = await crowdfund.funds(accounts[3]);
            assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
        });

        it("should prevent random accounts from calling approve",
async () => {
            try {
                await crowdfund.approve(true, {
                    from: accounts[1]
                });
            } catch (error) {
                let invalidOpcode = error.message.search("invalid
```

```
opcode") >= 0;
                let outOfGas = error.message.search("out of gas") >=
0;
                assert(invalidOpcode || outOfGas, "Expected throw but
got: " + error);
                return;
            }
            assert(false, "Expected throw not received");
        });

        it("should release funds during approval", async () => {
            let initialReceiverBalance = await
getBalance(accounts[1]);
            let initialContractBalance = await
getBalance(crowdfund.address);

            await crowdfund.approve(true, {
                from: accounts[0]
            });

            let finalReceiverBalance = await getBalance(accounts[1]);
            let finalContractBalance = await
getBalance(crowdfund.address);

            assert.equal(finalContractBalance.valueOf(), 0, "The full
balance of the contract was not transferred");

assert.equal(finalReceiverBalance.minus(initialReceiverBalance).valueO
f(), initialContractBalance.valueOf(), "The receiver did not receive
the funds from the contract");
        });

        it("should allow refunds when not approved", async () => {
            let initialBalance = await getBalance(accounts[5]);
            await crowdfund.approve(false, {
                from: accounts[0]
```

```
            });
            await crowdfund.refund(accounts[5], {
                from: accounts[3]
            });
            let finalBalance = await getBalance(accounts[5]);

            assert.equal(finalBalance.minus(initialBalance).valueOf(),
web3.toWei(1, "ether").valueOf(), "The refund receiver did not receive
the correct amount of refund");
        });
    })

});
```

Inside the `try` block, we attempt to call `approve` from `accounts[1]` which is the `receiver` account. If it works, the `assert` function at the very bottom is fired, which has `false` as its first argument. As a result, if it reaches there, the assertion fails and so does the test.

However, if the attempt at calling `approve` fails, the `catch` block is fired, with `error` which contains information about the error.

```
let invalidOpcode = error.message.search("invalid opcode") >= 0;
let outOfGas = error.message.search("out of gas") >= 0;
```

The `search` method of string returns the index of the first match found of the substring passed into it. If there are no matches, -1 is returned. So, we check whether `invalid opcode` and `out of gas` is present in the error message and save them in the corresponding variables.

```
assert(invalidOpcode || outOfGas, "Expected throw but got: " + error);
return;
```

This line asserts that either the error is due to `invalid opcode` or `out of gas`. If not, `invalidOpcode || outOfGas` will evaluate to `false` and the test will fail, logging out the actual error received. This is because, when an error is thrown by `require` or `assert`, either of these message must be present. Otherwise, the error is due to another reason which is not expected in this test. After this, the `return` ends the test so that it does not reach the final assert statement below it.

However, this is not an optimal solution, as it involves at least 7 lines of code just to expect an error each time. So, we are going to wrap these code into a function.

```
let expectThrow = async (promise) => {
    try {
        await promise;
    } catch (error) {
        let invalidOpcode = error.message.search("invalid opcode") >=
0;
        let outOfGas = error.message.search("out of gas") >= 0;
        assert(invalidOpcode || outOfGas, "Expected throw but got: " +
error);
        return;
    }
    assert(false, "Expected throw not received");
}

let getBalance = (account) => {
    return new Promise((resolve, error) => {
```

```
        web3.eth.getBalance(account, (err, balance) => {
            if (err) {
                error(err);
            } else {
                resolve(balance);
            }
        });
    });
};

module.exports = {
    expectThrow, getBalance
};
```

Here, we have taken the two functions, `getBalance` and `expectThrow` to package them into one module. We saved it as `utils.js` in the `helper` folder under `tests`. While not absolutely necessary, this makes code easier to work there can be multiple test files and each will require common functions such as `getBalance` and `expectThrow`. `expectThrow` basically takes a promise as an argument which it awaits inside a `try` block just like in our previous example. In this case, it works with calling any function, not just `approve`, allowing it to be reused elsewhere. Finally, the functions are exported by

```
module.exports = {
    expectThrow, getBalance
};
```

Now, we can simply access these functions from `crowdfund.js` by calling

```
const utils = require("./helper/utils");
```

and now, we can simply expect a throw by using

```
utils.expectThrow(crowdfund.approve(true, {
    from: accounts[1]
}));
```

This is the final version of `crowdfund.js`

```
const Crowdfund = artifacts.require("Crowdfund");
const utils = require("./helper/utils");

contract("Crowdfund", (accounts) => {
    let crowdfund;
    beforeEach(async () => {
        crowdfund = await Crowdfund.new(accounts[0], accounts[1]);
    });

    it("should have verifier and receiver set correctly", async () =>
{
        let verifier = await crowdfund.verifier();
        assert.equal(verifier, accounts[0], "The first account was not
the verifier");
        let receiver = await crowdfund.receiver();
        assert.equal(receiver, accounts[1], "The second account was
not the receiver");
    });

    context("fund contract", () => {
        beforeEach(async () => {
            await crowdfund.fund({
```

```
                from: accounts[3],
                value: web3.toWei(1, "ether")
            });
            await crowdfund.fund({
                from: accounts[4],
                value: web3.toWei(2, "ether")
            });
        });

        it("should keep track of funds", async () => {
            let fund = await crowdfund.funds(accounts[3]);
            assert.equal(fund.valueOf(), web3.toWei(1, "ether"), "The
contract did not keep track of funds properly");
        });

        it("should prevent random accounts from calling approve",
async () => {
            utils.expectThrow(crowdfund.approve(true, {
                from: accounts[1]
            }));
        });

        it("should release funds during approval", async () => {
            let initialReceiverBalance = await
utils.getBalance(accounts[1]);
            let initialContractBalance = await
utils.getBalance(crowdfund.address);

            await crowdfund.approve(true, {
                from: accounts[0]
            });

            let finalReceiverBalance = await
utils.getBalance(accounts[1]);
            let finalContractBalance = await
utils.getBalance(crowdfund.address);
```

```
            assert.equal(finalContractBalance.valueOf(), 0, "The full
balance of the contract was not transferred");

assert.equal(finalReceiverBalance.minus(initialReceiverBalance).valueO
f(), initialContractBalance.valueOf(), "The receiver did not receive
the funds from the contract");
        });

        it("should allow refunds when not approved", async () => {
            let initialBalance = await utils.getBalance(accounts[5]);
            await crowdfund.approve(false, {
                from: accounts[0]
            });
            await crowdfund.refund(accounts[5], {
                from: accounts[3]
            });
            let finalBalance = await utils.getBalance(accounts[5]);

            assert.equal(finalBalance.minus(initialBalance).valueOf(),
web3.toWei(1, "ether").valueOf(), "The refund receiver did not receive
the correct amount of refund");
        });
    })

});
```

Testing events

Now, we are going to look at how to test events. Since `Crowdfund` does not fire any event, we are going to use `ShopToken` instead.

```
const ShopToken = artifacts.require("ShopToken");
```

```
contract("ShopToken", (accounts) => {
    it("should fire Transfer event when transfer is called", async ()
=> {
        let shopToken = await ShopToken.new("10000");
        let tx = await shopToken.transfer(accounts[1], "100", {
            from: accounts[0]
        });
        assert.equal(tx.logs.length, 1, "Expected one event to be
fired");
        let log = tx.logs[0];
        assert.equal(log.event, "Transfer", "Transfer event was not
fired");
        assert.equal(log.args._from, accounts[0], "_from argument is
not set correctly");
        assert.equal(log.args._to, accounts[1], "_to argument is not
set correctly");
        assert.equal(log.args._value, 100, "_value arugment is not set
correctly");
    });
});
```

We are going to create this file in the `test` folder for the `ShopToken` project directory. This test will only test the `transfer` function and check whether it logs `Transfer` event correctly.

```
let tx = await shopToken.transfer( ... );
```

Since `transfer` is not a constant function, it returns an object with transaction data. The `log` property of `tx` is an array with all the events fired during the course of this transaction.

```
assert.equal(tx.logs.length, 1, "Expected one event to be fired");
```

This line ensures that only one event has been fired by ensuring the array length is equal to 1.

```
let log = tx.logs[0];
assert.equal(log.event, "Transfer", "Transfer event was not fired");
```

The `event` property of a log indicates the type of event that is fired, so this ensures that `Transfer` event is fired. After this, we check that the arguments which is stored as properties of `log.args` are correctly set.

Testing time-dependent functions

In `ShopTokenICO` we have introduced a deadline after which `buy` function cannot be called. However, with what we have learnt so far we will not be able to test this automatically as we cannot manipulate time in the EVM. `now` in Solidity, returns the timestamp of the latest block, so, if we are able to change this, we will be test the feature. While this is not possible in the public testnets or the mainnet, we can do this with `truffle develop`.

```
const Inheritable = artifacts.require("Inheritable");
const utils = require("./helper/utils.js");

contract("Inheritable", (accounts) => {
    let inheritable;
    beforeEach(async () => {
        inheritable = await Inheritable.new(30, accounts[1], {
            value: web3.toWei(1, "ether")
        });
    });
    it("should throw error when calling inherit before 30 days", async
```

```
() => {
        utils.expectThrow(inheritable.inherit({
            from: accounts[1]
        }));
    });

    it("should allow calling inherit after 30 days", async () => {
        await utils.increaseTime(31 * 24 * 60 * 60);
        await utils.advanceBlock();
        await inheritable.inherit({
            from: accounts[1]
        });
    });
});
```

We have implemented two tests here. The first one ensures that it is not possible to call the `inherit` function immediately after its creation. The second test goes forward 31 days in the future and attempts to call `inherit`. It expects this transaction to be successful.

```
let increaseTime = (addSeconds) => {
    return new Promise((resolve, reject) => {
        web3.currentProvider.sendAsync({
            jsonrpc: "2.0",
            method: "evm_increaseTime",
            params: [addSeconds],
            id: 0
        }, (error, result) => {
            if (error) {
                reject(error);
            } else {
                resolve(result);
            }
```

```
        });
    });
};

let advanceBlock = () => {
    return new Promise((resolve, reject) => {
        web3.currentProvider.sendAsync({
            jsonrpc: "2.0",
            method: "evm_mine",
            params: [],
            id: 0
        }, (error, result) => {
            if (error) {
                reject(error);
            } else {
                resolve(result);
            }
        });
    });
};
```

We have introduced two new functions to `utils.js` module. `increaseTime` takes `seconds` as an argument which is the number of seconds by which we will increase the next block's time. Usually, we use `web3` methods such as `web3.eth.getBalance` to interact with our Ethereum client. However, behind the scene, `web3` uses a protocol known as JSON-RPC to interact with the Ethereum client. The methods `evm_increaseTime` and `evm_mine` are not part of the `web3` standard. They have been introduced by `truffle` and `testrpc` to enable developers test time-related functions. As a result, we need to send JSON-RPC custom requests using `web3.currentProvider.sendAsync` . The method `evm_increaseTime` takes an integer as parameter. This integer is added to the sum of previous integers with which `evm_increaseTime` was called and stored as the new sum. As a result, this method, on its own does not increase the time in the EVM. The function returns a promise which is resolved when the callback to `web3.currentProvider.sendAsync` is fired.

`advanceBlock` sends another JSON-RPC request to the Ethereum client in order to manually issue a new block using `evm_mine`. When a new block is mined, usually the block's time is set to OS time at that moment. However, `testrpc` and `truffle` adds the sum calculated above to the OS time, ensuring the blockchain time is increased.

In `beforeEach`, we deploy an `Inheritable` contract and assign the second account as its child and 30 days as interval. As a result, the `inherit` function can be called only 30 days after the time of creation. The first test expects calling `inherit` will throw an error as no time has passed before it.

```
await utils.increaseTime(31 * 24 * 60 * 60);
await utils.advanceBlock();
```

These two functions collectively increases the EVM time by 31 days. `31 * 24 * 60 * 60` calculates the number of seconds in 31 days as `increaseTime` takes seconds as an argument. `utils.advanceBlock` simply mines a new block with the new increased time. Now, the test can expect `inherit` to be called without throwing an error.

Testing in Solidity

Tests can also be written in Solidity in the form of smart contracts. These contracts also share the `test` directory with the JavaScript tests. We are going to create a new file `TestCrowdfund.sol` for this test contract. The name of any test contract should start with `Test` so that `truffle` can identify, deploy and run them.

```
pragma solidity ^0.4.17;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";

import "../contracts/Crowdfund.sol";

contract TestCrowdfund {
    function testVerifier() {
        Crowdfund crowdfund =
Crowdfund(DeployedAddresses.Crowdfund());
        Assert.equal(crowdfund.verifier(), msg.sender, "The verifier
is not set correctly");
    }
}
```

This is an example of a basic test in Solidity.

```
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
```

```
import "../contracts/Crowdfund.sol";
```

These are two libraries we are going to use for testing. `Assert` is just like `assert` in mocha and logs events when assertions fail. These events are parsed by `truffle` which then fails the test and display the error to the developer. `DeployedAddresses` can be used to access the latest address to which the contracts have been deployed to by the deployment script. Just like in the JavaScript tests, before every test contract is deployed, the deployment script is run again to give the tests a clean slate to work on. The `Crowdfund` contract is also imported as an interface is necessary to interact with the deployed address.

```
function testVerifier() { ... }
```

Every function starting with `test` will be called by `truffle` as an individual test.

```
Crowdfund crowdfund = Crowdfund(DeployedAddresses.Crowdfund());
```

`DeployedAddresses.Crowdfund()` returns the address of the deployed (by the deployment script) `Crowdfund` contract which is then passed into `Crowdfund` so that the contract can interact with its functions.

```
Assert.equal(crowdfund.verifier(), msg.sender, "The verifier is not set correctly");
```

The syntax for assertion is same in Solidity as in mocha, with the exception of `Assert` in place of `assert`. The test functions are called by the first account which is also set as the `verifier` in the deployment script. As a result, the `msg.sender` should be equal to the `verifier` of the deployed `Crowdfund` contract.

Next, we would like to test if `Crowdfund` keeps track of the funds.

```
pragma solidity ^0.4.17;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";

import "../contracts/Crowdfund.sol";

contract TestCrowdfund {
    uint public initialBalance = 1 ether;

    function testVerifierSet() {
        Crowdfund crowdfund =
Crowdfund(DeployedAddresses.Crowdfund());
        Assert.equal(crowdfund.verifier(), msg.sender, "The verifier
is not set correctly");
    }

    function testTrackFund() {
        Crowdfund crowdfund =
Crowdfund(DeployedAddresses.Crowdfund());
        crowdfund.fund.value(1 ether)();
        Assert.equal(crowdfund.funds(this), 1 ether, "The contract did
not keep track of funds properly");
    }
}
```

To call the `fund` function of `Crowdfund` we would need some Ether balance in the `TestCrowdfund` contract. In order to get this, we have to set a `public` variable `initialBalance` which will be accessed by `truffle` and the amount of Ether will be sent to the contract. After this, we create a new function `testTrackFund`. After initializing `crowdfund` we then call the `fund` function with 1 Ether. Then by calling `funds` with `this` address as parameter we check if the contribution has been tracked properly. However, there are two issues with this currently. Firstly, we are not deploying a new `Crowdfund` contract with each test which is necessary for reasons discussed previously. Secondly, we cannot test whether `receiver` has been set correctly as the test contract does not have access to the addresses except that of the first account.

beforeEach

```
pragma solidity ^0.4.17;

import "truffle/Assert.sol";

import "../contracts/Crowdfund.sol";

contract TestCrowdfund {
    uint public initialBalance = 1 ether;
    Crowdfund crowdfund;

    function beforeEach() {
        crowdfund = new Crowdfund(this, msg.sender);
    }

    function testVerifierAndReceiverSet() {
        Assert.equal(crowdfund.verifier(), this, "The verifier is not
set correctly");
```

```
        Assert.equal(crowdfund.receiver(), msg.sender, "The receiver
is not set correctly");
    }

    function testTrackFund() {
        crowdfund.fund.value(1 ether)();
        Assert.equal(crowdfund.funds(this), 1 ether, "The contract did
not keep track of funds properly");
    }
}
```

Both of these can be solved by deploying a new `Crowdfund` contract in the `beforeEach` function. As the name suggests, `truffle` calls this function before every test. Since tests in Solidity are constrained by the limits as smart contracts, `beforeEach` will also have a gas limit. As a result, it may be necessary to break it up into multiple functions. `truffle` will call every function beginning with `beforeEach` before each test. In this function, we create a new `Crowdfund` contract and set its `verifier` as the test function itself and the `receiver` as the `msg.sender` which is equivalent to `account[0]` in JavaScript.

So far, the logic of the test is nearly identical to the JavaScript counterpart. However, this will not be the same for future tests. For one, we are unable to get access to other accounts and addresses. This can be solved with the use of `Proxy` contracts.

Proxy contracts

```
pragma solidity ^0.4.17;
```

```
contract Proxy {
  address public target;
  bytes data;
  uint value;

  function Proxy(address _target) {
    target = _target;
  }

  function() payable {
    data = msg.data;
    value = msg.value;
  }

  function execute() returns (bool) {
    return target.call.value(value)(data);
  }
}
```

These are contracts that acts as proxies for other contract. Here, we will use them as proxy funders for `Crowdfund` project. In the constructor function, we pass in the address of the contract which will be called. In this case, it is the `Crowdfund` contract.

Then the test contract will call the `fund` function on the `Proxy` contract. Since it does not have a `fund` function, this will trigger the fallback function which saves the `msg.data` and `msg.value` to state variables `data` and `value`. `msg.data` contains data about the call such as the function being called and the arguments. After this, the `execute` function which consists of

`return target.call.value(value)(data);`

The `target` is called with the `value` and `data` saved from the fallback function. Depending on whether this call throws an error, it will evaluate to `true` or `false` which is returned to the test contract. Essentially, the whole function call has been forwarded through the `Proxy`, changing the `msg.sender` in the `Crowdfund` contract. We can have multiple `Proxy` contracts such as these, each acting as a different funder account.

```
pragma solidity ^0.4.17;

import "truffle/Assert.sol";

import "../contracts/Crowdfund.sol";
import "./helper/Proxy.sol";

contract TestCrowdfund {
    uint public initialBalance = 5 ether;
    Crowdfund crowdfund;
    Proxy funderOne;
    Proxy funderTwo;

    function beforeEach() {
        crowdfund = new Crowdfund(this, msg.sender);
        funderOne = new Proxy(address(crowdfund));
        funderTwo = new Proxy(address(crowdfund));
    }

    function testVerifierAndReceiverSet() {
        Assert.equal(crowdfund.verifier(), this, "The verifier is not
set correctly");
        Assert.equal(crowdfund.receiver(), msg.sender, "The receiver
is not set correctly");
    }
```

```
    function testTrackFund() {
        Crowdfund(address(funderOne)).fund.value(1 ether)();
        funderOne.execute();
        Assert.equal(crowdfund.funds(address(funderOne)), 1 ether,
"The contract did not keep track of funds properly");
    }

    function testReleaseFundAfterApproval() {
        Crowdfund(address(funderOne)).fund.value(1 ether)();
        funderOne.execute();

        uint initialBalance = msg.sender.balance;
        crowdfund.approve(true);
        uint finalBalance = msg.sender.balance;

        Assert.equal(finalBalance - initialBalance, 1 ether, "The
receiver did not receive the funds from the contract");
    }

    function testAllowRefundsWhenNotApproved() {
        Crowdfund(address(funderOne)).fund.value(1 ether)();
        Crowdfund(address(funderTwo)).fund.value(2 ether)();
        funderOne.execute();
        funderTwo.execute();

        crowdfund.approve(false);

        uint initialBalance = address(funderOne).balance;
        Crowdfund(address(funderOne)).refund(address(funderOne));
        funderOne.execute();
        uint finalBalance = address(funderOne).balance;

        Assert.equal(finalBalance - initialBalance, 1 ether, "The
funder did not receive the correct amount of refund");
    }
}
```

First of all, we have imported the `Proxy` contract at the very top. Next, we have introduced two new state variables `funderOne` and `funderTwo` which are both `Proxy` contracts. We have to deploy two new `Proxy` contract each time in `beforeEach` function as new `Crowdfund` contract is deployed, hence, the `target` is different. When passing `crowdfund` we have to convert it to `address` so we pass in `address(crowdfund)`. In the function `testTrackFund`, instead of directly calling `fund` from `crowdfund`, we do it through `funderOne`.

```
Crowdfund(address(funderOne)).fund.value(1 ether)();
```

While this may seem complicated at a glance, it can be broken down. First of all, by calling `address(funderOne)` we are getting the address of `funderOne` proxy. We then pass it into `Crowdfund`. Now, we have an interface for `Crowdfund` attached to a `Proxy` contract. We then call the function `fund` with 1 Ether. However, `funderOne` being a `Proxy` contract does not have a `fund` function. Instead it triggers the fallback function which saves the `msg.data` and `msg.value`. In the next line, we call

```
funderOne.execute();
```

and with this the `fund` function call is forwarded to the `Crowdfund` contract. Now, calling `crowdfund.funds(address(funderOne))` must return 1 Ether as to the `Crowdfund` `fund` has been called by `funderOne`.

This same method is used to fund `Crowdfund` in `testReleaseFundAfterApproval` and `testAllowRefundsWhenNotApproved`. Other than the use of `Proxy` for doing this, the logic of the tests are identical to the

JavaScript counterpart. One distinction is that when calling `refund` in `testAllowRefundsWhenNotApproved` we used

```
Crowdfund(address(funderOne)).refund(address(funderOne));
funderOne.execute();
```

Here, `address` of `funderOne` is passed as an argument, whereas in JavaScript, the account from which the `refund` function was called was different from the account to which the Ether was sent to. This is because of the gas cost of sending the transaction makes it difficult to calculate the Ether received by the account. However, `funderOne` here does not pay the gas cost, it is paid by the account from which the transaction originated. Thus, it is not necessary to use a different account to receive the Ether.

Testing error

With the use of `Proxy`, testing for error takes very little additional code. The `Proxy` contract was designed with the need to catch errors in mind. That is exactly why instead of having the fallback function call the target, it uses a separate function `execute`. When an error is thrown in `Crowdfund`, the `execute` function returns `false`. This would not be possible with just the fallback function as fallback functions are unable to return data. So, for our last test, we will add the following function

```
function testPreventRandomAccountsFromApproving() {
        Crowdfund(address(funderOne)).approve(true);
        bool success = funderOne.execute();
```

```
        Assert.equal(success, false, "Excepted approve to throw
error");
    }
```

Here, after we call `approve` through `funderOne`, we store the returned `bool` as `success`. Then we assert it to be `false`, thereby ensuring `funderOne` fails to call `approve` as it is not the `verifier`.

Testing in JavaScript vs Solidity

As mentioned previously, testing in Solidity is constrained by the same limits faced by smart contracts. These include

- Being unable to test events as smart contracts do not have access to event data
- Being unable to test time related features as smart contracts cannot send request outside the blockchain to trigger `truffle` or `testrpc` to manipulate time
- Does not have access to externally owned accounts and as such must resort to the use of proxy contracts
- Subject to gas limits

These limitations make testing in Solidity unattractive as opposed to testing in JavaScript. However, it is not without its merits. When we are primarily testing how smart contracts interact with each other instead of EOA with smart contracts, it is much easier to do so in Solidity than JavaScript, as smart contracts are written in Solidity. Moreover, while it is less powerful in certain situations, many prefer testing in Solidity on more philosophical grounds, namely, tests should be written in the same language as the program being tested.

Developing DApp Frontend

The frontend for DApp will be written using HTML, CSS and JavaScript. It will be used as a graphical interface for ordinary users to interact with the smart contracts. Web3, a JavaScript library, will be used to interact with the smart contracts through the Ethereum client such as Geth, Parity and Truffle for development.

In this chapter we are going to build the frontend for the Shop Token ICO.

First, we are going to create a new directory for our project. Here, we will open the command-line and run

```
truffle unbox webpack
```

This will create the following files

```
├── app
│   ├── javascripts
│   │   └── app.js
│   ├── stylesheets
│   │   └── app.css
│   └── index.html
├── contracts
│   ├── ConvertLib.sol
│   ├── MetaCoin.sol
│   └── Migrations.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_deploy_contracts.js
```

```
├── node_modules
│   └── ...
├── test
│   ├── metacoin.js
│   └── TestMetacoin.sol
├── package.json
├── truffle.js
├── webpack.config.js
└── ...
```

Some of the files were omitted as they are will not be covered in this chapter.

The HTML and CSS

First, we are going to build the interface. For that, we are going to replace the contents of `app/index.html` with

```
<!DOCTYPE html>
<html>

<head>
    <title>Shop Token ICO</title>
    <link href='https://fonts.googleapis.com/css?
family=Open+Sans:400,700' rel='stylesheet' type='text/css'>
</head>

<body>
    <div class="intro">
        <h1>Shop</h1>
        <p>Over the course of the next two years, we will develop
Shop, a DApp for buying and selling goods in a decentralized manner.
All transactions will take place using shop tokens, so to buy things
from Shop, one must own shop tokens. Anyone can become a buyer or
seller in this DApp. This will dIsRupT tHe oNlInE sHopPinG iNduStrY.
</p>
        <h2>Shop ICO</h2>
        <p>The best way to get shop tokens is by buying them right now
from the ICO while the price is low. The fund raised here will be used
to <s>make me rich</s> fund the development of Shop. The deadline for
the ICO is 14th October 2018. If the minimum funding goal of 300 Ether
is not raised by then, you can get a refund from here. Otherwise, it
will be withdrawn by the team to develop the DApp. The maximum funding
is 500 Ether. One Shop Token costs 0.01 Ether.</p>
    </div>
```

```
    <div class="balance">
        <p>Your current account holds <span id="shop-token-balance">
</span> Shop Tokens</p>
        <p>The ICO holds <span id="ico-shop-token-balance"></span>
Shop Tokens</p>
    </div>
    <div class="funding">
        <p>ICO is ongoing, you can buy Shop Tokens with Ether!</p>
        <div>
            <div>
                <input placeholder="Amount" id="amount" />
            </div>
            <div>
                <input placeholder="Address to send tokens to (leave
blank to use current account)" id="send-token-address" />
            </div>
        </div>
        <div class="button-container">
            <button id="buy">Buy</button>
        </div>
    </div>
    <div class="refund">
        <p>The ICO has failed. Use the form below to recover your
Ether.</p>
        <input placeholder="Address to refund to (leave blank to use
current)" id="refund-address" />
        <button id="refund">Refund</button>
    </div>
    <div class="success">
        <p>The ICO was successful! Thank you for your cooperation.</p>
    </div>
    <script
src="//ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
</script>
    <script src="./app.js"></script>
</body>
```

```
</html>
```

The page can be broken down into 5 sections, `intro` , `balance` , `funding` , `success` and `refund` .

- `intro` contains the description for the upcoming Shop DApp and its ICO.
- `balance` gives the Shop Token balance of the current account and the tokens left in the ICO.
- `funding` has the interface to contribute. This includes an input where users will type in the number of Shop Tokens to buy. There is another input where users can specify which account to send the Shop Tokens to. Below, the cost in ETH and USD will be shown.
- `refund` has the interface to sell the Shop Tokens including an input where the address to send the Ether to can be specified.
- `success` displays a simple message that the ICO has been successful While all `funding` , `refund` and `success` sections are shown right now, once the DApp is complete, it will only show one of them depending on whether funding goal has been reached and whether deadline has passed yet.

At the end we added jQuery so that we can use it to easily manipulate the DOM.

Next, we will replace the content of `app/stylesheets/app.css` with the following

```
body {
```

```css
    margin-left: 25%;
    margin-right: 25%;
    margin-top: 5%;
    font-family: "Open Sans", sans-serif;
}

input {
    padding: 5px;
    font-size: 16px;
    width: 100%;
    margin: 5px;
}

#refund-address {
    width: 80%;
}

select {
    padding: 5px;
    font-size: 16px;
}

.button-container {
    text-align: center;
    padding: 10px;
}

button {
    font-size: 16px;
    padding: 5px;
    margin: 0px auto;
}

h1,
h2 {
    display: inline-block;
```

```
    vertical-align: middle;
    margin-top: 0px;
    margin-bottom: 10px;
}

h2 {
    color: #AAA;
    font-size: 32px;
}

.funding,
.refund,
.success {
    display: none;
}
```

The above code is self-explanatory. Next, we will write the script for interacting with the smart contract in `app/javascripts/app.js`

JavaScript

```
import "../stylesheets/app.css";

import {default as Web3} from 'web3';
import {default as contract} from 'truffle-contract';

import shopTokenArtifacts from '../../build/contracts/shopToken.json';
import shopTokenICOArtifacts from
'../../build/contracts/shopTokenICO.json';

var ShopToken = contract(shopTokenArtifacts);
var ShopTokenICO = contract(shopTokenICOArtifacts);
var accounts;

window.App = {
    start: function () {
        ShopToken.setProvider(web3.currentProvider);
        ShopTokenICO.setProvider(web3.currentProvider);
        web3.eth.getAccounts(function (err, accs) {
            console.log(accs)
            if (err != null) {
                alert("There was an error fetching your accounts.");
                return;
            }
            if (accs.length == 0) {
                alert("Couldn't get any accounts! Make sure your
Ethereum client is configured correctly.");
                return;
            }
            accounts = accs;
        });
```

```
    }
};

window.addEventListener('load', function () {
    if (typeof web3 !== 'undefined') {
        window.web3 = new Web3(web3.currentProvider);
    } else {
        console.warn("No web3 detected. Falling back to
http://localhost:8545. You should remove this fallback when you deploy
live, as it's inherently insecure. Consider switching to Metamask for
development. More info here:
http://truffleframework.com/tutorials/truffle-and-metamask");
        window.web3 = new Web3(new
Web3.providers.HttpProvider("http://localhost:8545"));
    }
    App.start();
});
```

This is the basic script on which we will build on. It imports the necessary utilities and stylesheets and sets up `web3` and the interface to the smart contract.

`window.addEventListener('load', function() { ... }`

This function is called when the page is fully loaded with all the JavaScript libraries and stylesheets.

```
if (typeof web3 !== 'undefined') {
    window.web3 = new Web3(web3.currentProvider);
}
```

This checks whether `web3` is defined. If it is, it means MetaMask is installed and it has injected `web3`. This is then passed into `new Web3()` which instantiates the interface to the Ethereum client and sets it to `window.web3`. If not, it assumes that this is run in developer's browser and so connects insecurely to the Ethereum client which in this case is TestRPC. It then calls `App.start()`.

```
ShopTokenICO.setProvider(web3.currentProvider);
```

This connects the interface to the contract to the Ethereum client so that it can communicate with the contract.

```
web3.eth.getAccounts(function (err, accs) { ... }
```

This passes all the user's accounts saved in the Ethereum client to the function as an array. If there is no error, `accs` is then assigned to `accounts` which will be accessible from any other function from the `App` object. This is so that the accounts array does not need to be fetched every time.

Next, we are going to create a new method `refreshBalance` which is going to populate the balance, in Shop Tokens, of the ICO and the current account.

```
window.App = {
    start: function () {
        ShopToken.setProvider(web3.currentProvider);
        ShopTokenICO.setProvider(web3.currentProvider);
        web3.eth.getAccounts(function (err, accs) {
            console.log(accs)
            if (err != null) {
                alert("There was an error fetching your accounts.");
                return;
            }
```

```
            if (accs.length == 0) {
                alert("Couldn't get any accounts! Make sure your
Ethereum client is configured correctly.");
                return;
            }
            accounts = accs;
            App.refreshBalance();
        });
    },
    refreshBalance: function () {
        ShopToken.deployed().then(function (shopToken) {
            return shopToken.balanceOf(accounts[0]);
        }).then(function (balance) {
            $("#shop-token-balance").html(balance.valueOf());
        });
        ShopTokenICO.deployed().then(function (shopTokenICO) {
            return shopTokenICO.getTokensLeft();
        }).then(function (balance) {
            $("#ico-shop-token-balance").html(balance.valueOf());
        });
    }
};
```

`App.refreshBalance()` is called in the callback function of `web3.eth.getAccounts` because `accounts` array is needed to get the balance of the current account in

`shopToken.balanceOf(accounts[0]).then(function (balance) { ... });`

As evident, the same APIs are used in a DApp to interact with smart contracts as in `truffle console` and `truffle exec`.

`$("#shop-token-balance").html(balance.valueOf());`

`balance.valueOf()` converts the `BigNumber` object into a `string` so that it can be displayed in the UI. Then the `string` replaces the inner HTML of the `span` with id `shop-token-balance`.

Next, we are going to modify `App.start` to check the stage the ICO is in and display the sections accordingly.

```
var accounts, shopToken, shopTokenICO;

window.App = {
    start: function () {
        ShopToken.setProvider(web3.currentProvider);
        ShopTokenICO.setProvider(web3.currentProvider);
        web3.eth.getAccounts(function (err, accs) {
            console.log(accs)
            if (err != null) {
                alert("There was an error fetching your accounts.");
                return;
            }
            if (accs.length == 0) {
                alert("Couldn't get any accounts! Make sure your
Ethereum client is configured correctly.");
                return;
            }
            accounts = accs;
            ShopToken.deployed().then(function (instance) {
                shopToken = instance;
                return ShopTokenICO.deployed();
            }).then(function (instance) {
                shopTokenICO = instance;
                App.refreshBalance();
                return shopTokenICO.deadline();
            }).then(function (deadline) {
                var currentTime = Math.floor(Date.now() / 1000);
```

```
                if (currentTime < deadline.toNumber()) {
                    return "funding";
                } else {
                    return shopTokenICO.refundingAllowed();
                }
            }).then(function (refundingAllowed) {
                if (refundingAllowed == "funding") {
                    $(".funding").show();
                } else if (refundingAllowed) {
                    $(".refund").show();
                } else {
                    $(".success").show();
                }
            });
        });
    },
    refreshBalance: function () {
        shopToken.balanceOf(accounts[0]).then(function (balance) {
            $("#shop-token-balance").html(balance.valueOf());
        });
        shopTokenICO.getTokensLeft().then(function (balance) {
            $("#ico-shop-token-balance").html(balance.valueOf());
        });
    }
};
```

Since we are going to interact with `ShopToken` and `ShopTokenICO` instances in different functions, it does not make sense to call `Contract.deployed()` every time to get the instances. Instead, here, we call `Contract.deployed()` once for both the contracts in `App.start` and set them to `shopToken` and `shopTokenICO` which were declared outside the `App` so that the instance can be accessed from any function.

After this is set, `App.refreshBalance` is called, which uses `shopToken` and `shopTokenICO` set above to get the balances. Next, `shopTokenICO.deadline` is called which is returned. In the next chain, we access the `deadline` then compare it with the `currentTime`.

```
var currentTime = Math.floor(Date.now() / 1000);
if (currentTime < deadline.toNumber()) {
    return "funding";
} else {
    return shopTokenICO.refundingAllowed();
}
```

`Date.now()` returns a UNIX timestamp in microseconds, so it must be converted to seconds before comparing it to deadline timestamp, hence it is divided by `1000`. Then if the current time exceeds the `deadline`, `"funding"` is returned. Otherwise, `shopTokenICO.refundingAllowed()` promise is returned, which will resolve with either `true` or `false` depending on whether minimum funding has been reached. Hence, in the next chain, we will either get `"funding"`, `true` or `false`.

```
if (refundingAllowed == "funding") {
    $(".funding").show();
} else if (refundingAllowed) {
    $(".refund").show();
} else {
    $(".success").show();
}
```

Lastly, depending on the value of `refundingAllowed`, different section of the UI is shown. The next step would be to introduce new methods to the `App` which are called when the respective buttons are pressed.

```
var accounts, shopToken, shopTokenICO, shopTokenBalance;

window.App = {
    start: function () {
        ShopToken.setProvider(web3.currentProvider);
        ShopTokenICO.setProvider(web3.currentProvider);
        web3.eth.getAccounts(function (err, accs) {
            console.log(accs)
            if (err != null) {
                alert("There was an error fetching your accounts.");
                return;
            }
            if (accs.length == 0) {
                alert("Couldn't get any accounts! Make sure your
Ethereum client is configured correctly.");
                return;
            }
            accounts = accs;
            ShopToken.deployed().then(function (instance) {
                shopToken = instance;
                return ShopTokenICO.deployed();
            }).then(function (instance) {
                shopTokenICO = instance;
                App.refreshBalance();
                return shopTokenICO.deadline();
            }).then(function (deadline) {
                var currentTime = Math.floor(Date.now() / 1000);
                if (currentTime < deadline.toNumber()) {
                    return "funding";
                } else {
```

```
                return shopTokenICO.refundingAllowed();
            }
        }).then(function (refundingAllowed) {
            if (refundingAllowed == "funding") {
                $(".funding").show();
            } else if (refundingAllowed) {
                $(".refund").show();
            } else {
                $(".success").show();
            }
        });
    });
    $("#buy").click(App.buy);
    $("#approve").click(App.approve);
    $("#refund").click(App.refund);
},
refreshBalance: function () {
    shopToken.balanceOf(accounts[0]).then(function (balance) {
        shopTokenBalance = balance;
        $("#shop-token-balance").html(balance.valueOf());
    });
    shopTokenICO.getTokensLeft().then(function (balance) {
        $("#ico-shop-token-balance").html(balance.valueOf());
    });
},
buy: function () {
    var amount = parseInt($("#amount").val());
    var receiver = $("#send-token-address").val();
    if (receiver.trim() == "") {
        receiver = accounts[0];
    }
    if (amount <= 0) {
        alert("You must buy at least 1 Shop Token");
        return;
    }
    var value = web3.toWei(amount/100, "ether");
```

```
        shopTokenICO.buy(receiver, {
            value: value,
            from: accounts[0]
        }).catch(function (error) {
            alert(error);
        }).then(App.refreshBalance);
    },
    approve: function() {
        shopToken.approve(shopTokenICO.address, shopTokenBalance, {
            from: accounts[0]
        }).then(function (transaction) {
            if (transaction.logs.length > 0) {
                $(".approve").hide();
                $(".sell").show();
            } else {
                throw new Error("Could not approve transfer to
ShopTokenICO");
            }
        }).catch(alert).then(App.refreshBalance);
    },
    refund: function () {
        var receiver = $("#refund-address").val();
        if (receiver.trim() == "") {
            receiver = accounts[0];
        }
        shopTokenICO.sell(receiver, shopTokenBalance, {
            from: accounts[0]
        }).catch(alert).then(App.refreshBalance);
    }
};

window.addEventListener('load', function () {
    if (typeof web3 !== 'undefined') {
```

We added three new methods to `App`, namely `buy`, `approve` and `refund`.

```
$("#buy").click(App.buy);
$("#approve").click(App.approve);
$("#refund").click(App.refund);
```

These three lines are added to the end of `App.start` to call the corresponding `App` methods when the buttons are pressed. Moreover, we added a single line to `App.refreshBalance` so that the Shop Token balance of the current account can be accessible as `shopTokenBalance` from anywhere within `App`.

```
var amount = parseInt($("#amount").val());
var receiver = $("#send-token-address").val();
if (receiver.trim() == "") {
    receiver = accounts[0];
}
if (amount <= 0) {
    throw new Error("You must buy at least 1 Shop Token");
}
```

In the `App.buy` method, first, the input by the user is parsed and validated. At first, `amount` is set to the value in the Amount input, parsed to integer with the `parseInt` function. `receiver` is set to the address to which the tokens will be sent to. Next, if `receiver` is empty, it is set to the current account address. Then, it checks whether `amount` is negative or 0, if so, the user is alerted and the function call ends.

```
var value = web3.toWei(amount/100, "ether");
shopTokenICO.buy(receiver, {
```

```
        value: value,
        from: accounts[0]
    }).catch(function (error) {
        alert(error);
    }).then(App.refreshBalance);
```

Next, the `value` is calculated. This is the amount of Ether, in wei, sent to the `buy` function of the `ShopTokenICO`. `shopToken.buy` is then called, to which the `receiver` is passed. If the transaction is not successful, the `error` is alerted to the user and finally, the balance is again refreshed.

Once the ICO deadline is reached, the `end` function is called manually and if the minimum funding goal is not reached, the Approve button will be displayed. Clicking this will call the `App.approve` function.

```
    shopToken.approve(shopTokenICO.address, shopTokenBalance, {
        from: accounts[0]
    }).then(function (transaction) {
        if (transaction.logs.length > 0) {
            $(".approve").hide();
            $(".sell").show();
        } else {
            throw new Error("Could not approve transfer to ShopTokenICO");
        }
    }).catch(alert).then(App.refreshBalance);
```

`shopToken.approve` is used to give approval to the `ShopTokenICO` contract account to transfer all the Shop Tokens back to the `owner`. This returns the transaction data to the callback function. Since a successful approval leads to an `Approval` event being logged, we check whether the `transaction.logs` array has at least one item. If so, we can assume that it has been successfully approved, and the UI for refunding is shown. If not, we throw an error. Finally,

```
}).catch(alert);
```

is the same as writing

```
}).catch(function (error) {
    alert(error);
});
```

`App.refund` is fired when the Refund button is pressed and it simply calls the `ShopTokenICO.refund` with a custom receiver if specified.

Watching Events

We can also watch events from contracts and react to them. Now, we are going to modify our DApp so that it watches all `Transfer` events and if it is relevant, the Shop Token balances are refreshed.

```
start: function () {
        ShopToken.setProvider(web3.currentProvider);
        ShopTokenICO.setProvider(web3.currentProvider);
        web3.eth.getAccounts(function (err, accs) {
            console.log(accs)
            if (err != null) {
                alert("There was an error fetching your accounts.");
                return;
            }
            if (accs.length == 0) {
                alert("Couldn't get any accounts! Make sure your
Ethereum client is configured correctly.");
                return;
            }
            accounts = accs;
            ShopToken.deployed().then(function (instance) {
                shopToken = instance;
                return ShopTokenICO.deployed();
            }).then(function (instance) {
                shopTokenICO = instance;
                var transfers = shopToken.Transfer();
                transfers.watch(function (err, event) {
                    if (err) return;
                    if(event.args._from == shopTokenICO.address ||
                       event.args._to == shopTokenICO.address ||
```

```
                         event.args._from == accounts[0] ||
                         event.args._to == accounts[0]) {
                          App.refreshBalance();
                      }
                  });
                  App.refreshBalance();
                  return shopTokenICO.deadline();
              }).then(function (deadline) {
                  var currentTime = Math.floor(Date.now() / 1000);
                  if (currentTime < deadline.toNumber()) {
                      return "funding";
                  } else {
                      return shopTokenICO.refundingAllowed();
                  }
              }).then(function (refundingAllowed) {
                  if (refundingAllowed == "funding") {
                      $(".funding").show();
                  } else if (refundingAllowed) {
                      $(".refund").show();
                  } else {
                      $(".success").show();
                  }
              });
          });
          $("#buy").click(App.buy);
          $("#approve").click(App.approve);
          $("#refund").click(App.refund);
      },
```

```
var transfers = shopToken.Transfer();
transfers.watch(function (err, event) { ... });
```

`shopToken.Transfer()` returns the `Transfer` event to which we can subscribe using the `.watch()` method. This is exactly what we do in the next line, passing a callback function which is fired each time a `Transfer` event is logged. The callback function takes two arguments, `err` which will be `null` if the there are no errors and `event` which will have data passed into the event by `ShopToken`.

```
if (err) return;
if(event.args._from == shopTokenICO.address ||
   event.args._to == shopTokenICO.address ||
   event.args._from == accounts[0] ||
   event.args._to == accounts[0]) {
    App.refreshBalance();
}
```

Inside the callback function, we first check if there is any error, if so, the function call stops right then. Next, we check if the `Transfer` leads to any change in the balances of either `ShopTokenICO` or the user's account. This is done by checking if the `Transfer` is `_to` or `_from` either the user's account or `ShopTokenICO`. These arguments of the `event` are accessed from its property `event.args`.

Moreover, we have also removed `.then(App.refreshBalance)` after calling `approve`, `buy` and `sell` as our callback function for watching `Transfer` events will automatically refresh the balances.

Security and Best Practices

Lazy Evaluation

Lazy evaluation is a concept commonly used in functional programming languages where computations are only done when they are required. This is a very important concept in smart contract development as well, due to the high cost of computation. It does not make sense for a smart contract to send some Ether until the `address` to which it is being sent to requires the Ether. Moreover, it does not make sense to process information until an account needs the processed information. Furthermore, this also means that the ones who require the evaluation are the one who pays for the gas required to make that evaluation.

Let us now look at the `refund` function in the latest version of `Crowdfund`

```
function refund() internal {
        refunded = true;
        for(uint i = 0; i < funders.length; i++){
            address funder = funders[i];
            funder.send(funds[funder]);
        }
    }
```

Here, the function carries out eager evaluation as it is sending Ether to all the users at once. This is bad for three reasons:

1) It will cost a lot for the `verifier` who has no incentive to call the function and spend so much on refunding to each funder 2) Block gas limit might be exceeded and the contract may stall (more on this later) 3) Some `send` may fail and these are not recorded so cannot even be retrieved later

Instead, we should let individual funder call the `refund` function which will only send refund their own fund.

```
pragma solidity ^0.4.17;

contract Crowdfund {
    bool public over;
    bool public refunding;
    bool public funded;
    address public verifier;
    address public receiver;
    mapping (address => uint) public funds;

    modifier onlyVerifier() {
        require(msg.sender == verifier);
        _;
    }

    modifier isNotOver() {
        require(!over);
        _;
    }

    modifier isRefunding() {
        require(refunding);
        _;
    }
```

}

```
    function Crowdfund(address _verifier, address _receiver) {
        verifier = _verifier;
        receiver = _receiver;
    }

    function fund() payable isNotOver external {
        require(msg.value != 0);
        funds[msg.sender] += msg.value;
    }

    function release() internal {
        funded = true;
        bool success = receiver.call.value(this.balance)();
        require(success);
    }

    function startRefund() internal {
        refunding = true;
    }

    function refund(address to) external isRefunding {
        require(funds[msg.sender] > 0);
        to.call.value(funds[msg.sender])();
        funds[msg.sender] = 0;
    }

    function approve(bool completed) onlyVerifier isNotOver external {
        over = true;
        if(completed){
            release();
        }else{
            startRefund();
        }
    }
}
```

Here, when the `Crowdfund` fails, `approve` calls `startRefund` instead of `refund`. The `startRefund` function simply sets `refunding` to true. As a result, now any funder can call `refund`.

```
function refund(address to) external isRefunding {
    require(funds[msg.sender] > 0);
    to.call.value(funds[msg.sender])();
    funds[msg.sender] = 0;
}
```

The `refund` function now takes one argument to which the contract sends the fund. First, it checks if the account calling the function has contributed any fund. Then it sets the account's fund to 0 so that they cannot call `refund` again. Finally, it transfers all the Ether of the funder to `to` while forwarding all the remaining gas to allow it to do necessary processing.

Every funder who has contributed to the `Crowdfund` has an incentive to call the `refund` function as they will get their Ether back, so we can safely assume this will be done eventually.

Reentrancy Attack

This may happen whenever calling an external untrusted contract where that contract calls any function from the original contract, leading to recursion. A specific example would be in `Crowdfund` where, in `refund` function, an external contract is called while sending all the Ether. The fallback function of that contract could run the `refund` function again. This is only possible due to the order of the final two statements in the `refund` function.

```
function refund(address to) external isRefunding {
    require(funds[msg.sender] > 0);
    to.call.value(funds[msg.sender])();
    funds[msg.sender] = 0;
}
```

Now, we will use the following contract `AttackCrowdfund` to `fund` and `refund` from `Crowdfund` .

```
pragma solidity ^0.4.17;

contract AttackCrowdfund {
    Crowdfund crowdfund;

    function AttackCrowdfund(address _crowdfund) {
        crowdfund = Crowdfund(_crowdfund);
    }

    function fund() payable {
```

```
        crowdfund.fund.value(msg.value)();
    }

    function requestRefund() {
        crowdfund.refund(this);
    }

    function() payable {
        crowdfund.refund(this);
    }
}
```

`AttackCrowdfund` needs to be first used to call `fund` in `Crowdfund`. Then once the `Crowdfund` fails and `refunding` begins, `refund` in `AttackCrowdfund` is called which then calls `refund` in `Crowdfund`.

Next, the `to.call.value(funds[msg.sender])();` is run which calls the fallback function of `AttackCrowdfund`. Since the fallback function calls `refund` again and `funds[msg.sender]` is not 0, the funds are sent again and so on, until the balance of `Crowdfund` is drained or gas limit of the transaction is reached. The whole transaction is not reversed because `address.call` only returns `false` when it fails and does not propagate the error.

To fix this, `funds[msg.sender] = 0;` must take place before the external call.

```
function refund(address to) external isRefunding {
    require(funds[msg.sender] > 0);
    uint fund = funds[msg.sender];
    funds[msg.sender] = 0;
    to.call.value(fund)();
```

```
}
```

Now even if `refund` is called again in the external call, the call will fail as `funds[msg.sender]` is 0.

Generally, it is advisable not to use `address.call` as it may be used maliciously and use `address.send` instead which does not forward any gas. However, if it is necessary to use `address.call`, it must be checked for possibility of reentrancy attack.

Another important thing to remember is not to assume that the state of the contract is the same after an external call is made to an untrusted contract as it can change the state of the contract by calling any of its other external functions. Generally speaking, it is best to have any external call at the end of the function.

Overflow and Underflow

`uint` have a range of 0 to 2^{256} - 1 which evaluates to 115792089237316195423570985008687907853269984665640564039457584007913129639935. When a negative value is assigned to an `uint`, it is out of range. As a result, to convert it to valid `uint`, this value is deducted from 2^{256} - 1. This phenomenon is known as underflow.

Similarly, if a number greater than 2^{256} - 1 is assigned to an `uint`, 2^{256} - 1 is deducted from it. This is known as an overflow.

```
contract Flows {
    function underflow() pure returns (uint) {
        uint number = 0;
        return number - 1;
    }

    function overflow() pure returns (uint) {
        uint number = 2**256 - 1;
        return number + 5;
    }
}
```

Here, `underflow()` returns 2^{256} - 1 whereas `overflow()` returns 4. Note that, overflow and underflow only takes place when the values are assigned the variables, not right after every arithmetic operation. `uint num = (2**256 + 2)/2` will evaluate to 2^{255} + 1, not 1.

As for `int`, the range is -2^{255} to $2^{255} - 1$. Underflow and overflow takes place when values outside this range are assigned to `int`. Generally, for any other type, such as `uint16`, any number outside the range of possible values leads to an overflow or an underflow depending on whether it is below or above the range.

In `ShopToken`, when `transfer` takes place, overflow must be checked for.

```
function transfer(address _to, uint _value) returns (bool) {
        if(ledger[msg.sender] >= _value && _value > 0 && _value +
ledger[_to] > ledger[_to]) {
            ledger[msg.sender] -= _value;
            ledger[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        }else{
            return false;
        }
    }
```

The new expression `_value + ledger[_to] > ledger[_to]` checks whether an overflow takes place. If `_value + ledger[_to]` is lower than `ledger[_to]`, an overflow has taken place, so the transfer fails.

Safe Math Library

With this in mind, it is important to check for overflows and underflows during every arithmetic operation to prevent unexpected behavior in the smart contract. This is where the safe math library comes in.

```
pragma solidity ^0.4.11;

library SafeMath {
  function mul(uint256 a, uint256 b) internal constant returns
(uint256) {
    uint256 c = a * b;
    assert(a == 0 || c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal constant returns
(uint256) {
    uint256 c = a / b;
    return c;
  }

  function sub(uint256 a, uint256 b) internal constant returns
(uint256) {
    assert(b <= a);
    return a - b;
  }

  function add(uint256 a, uint256 b) internal constant returns
(uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

For every type of operation, it checks for overflow/underflow depending on the type of operation. For example, for multiplication, we will use `SafeMath.mul` which checks whether `c / a == b` which holds `true` given no overflow has taken place. If it is `false` an error is thrown and the transaction or message is reversed. Similarly, for `SafeMath.sub`, it checks for underflow by ensuring `b <= a` because if `a` is greater than `b`, the resulting number will be negative leading to an overflow since it is assigned to a `uint`. However, `SafeMath.div` does not have any assertions as it never leads to an overflow or underflow. We can use `SafeMath` in contracts in the following manner

```
using SafeMath for uint;

function refund(uint _value, address _to) {
    fund[msg.sender] = fund[msg.sender].sub(_value);
    _to.transfer(_value);
}
```

Here, if `_value` is greater than `fund[msg.sender]`, `SafeMath.sub` will throw an error as an underflow takes place and the transaction will fail.

Contract Stalling

In an earlier version of `Crowdfund`, the refund function was

```
function refund() internal {
        refunded = true;
        for(uint i = 0; i < funders.length; i++){
            address funder = funders[i];
            funder.send(funds[funder]);
        }
    }
```

Here, we are using a `for` loop for refunding each funder. This will become problematic when the number of funders is really high. Since, there is virtually no limit on the number of funders, the gas cost might become very high as well. This will not only lead to high cost of calling `refund` for the `verifier` but might make it impossible. Just like transactions, blocks have a gas limit too. At the time of writing, the block gas limit is about 6500000 and the cost of `address.send` is about 9040 gas units. As a result, if the number of funders exceed 6500000/9040, that is, approximately 719, the contract will be stalled as the verifier cannot call the `refund` function successfully. However, the block gas limit is not fixed and changes over time.

To solve this problem, lazy evaluation can be used which is described above.