Smart Contracts: a Tutorial

October 16, 2023

Since their creation, blockchains have fostered innovation and created new distributed computing paradigms. Among these, we can find smart contracts, which are in essence, computer programs deployed on a blockchain and run by miners. In this document, we explore how to write a smart contract using Solidity, debug it using Truffle/Ganache, and then introduce the ERC-20 tokens on top of Ethereum.

1 My First Smart Contract

In this section, we write our first smart contract. For this contract (and the entire tutorial), we will use Solidity. It is a statically-typed programming language used for developing smart contracts that run on top of Ethereum. Full documentation of this programming language can be found at `https://docs.soliditylang.org/en/latest/`. In this tutorial, we will focus on the basic building blocks of a solidity contract and how to debug it on Remix.

1.1 How to write a Smart Contract?

Generally, a Solidity smart contract looks as follows:

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.1 <0.9.0;
3
4  contract SimpleAuction {
5      function bid() public payable { // Function
6          // ...
7      }
8
9      constructor() public{
10         //...
11     }
12 }
13
14 // Helper function defined outside of a contract
15 function helper(uint x) pure returns (uint) {
16     return x * 2;
17 }
```

Usually, the layout of a contract is composed of the following sections:

- **SPDX License Identifier** (Optional / Encouraged): This is a machine-readable string that indicates the license under which the smart contract is released. This is done to solve the legal issues related to making code available. The compiler doesn't verify the validity of this field.
- **Pragmas**: the pragma keyword is used to enable some compilers/language features and checks. They are local to the file and do not propagate; if you are importing functionality from another contract, the pragmas may or may not be the same. An important pragma to be used is the language version, it indicates which version of the language to use and what language methods and improvements are available to use in the smart contract.
- **Contract Body**: The contract body contains the logic of your contract. It usually has variable definition, local methods, and a constructor [1]

The program may also include import statements (if you are importing functionality from other files), and global helper methods ...

1.2 Using Remix to Debug Our Contracts

One main difference between smart contracts and regular programs is the ability to change and improve the program after it is deployed in production. A smart contract is immutable after it's deployed (the opposite of regular programs), and if an update is needed, the old contract needs to be destroyed, then the new version deployed, incurring extra gas fees. Thus, a smart contract developer needs to exercise due diligence and ensure that a contract is error-free, before deploying it.

Several solutions exist to debug a contract. In this part of the tutorial, we will use remix, which is available at `https://remix.ethereum.org/`.

Generally, its interface looks as follows (Figure **??**):

To understand how Remix Works, we use the same smart contract defined in the course slides:

[1] The constructor may also be implicitly defined if it doesn't perform any action

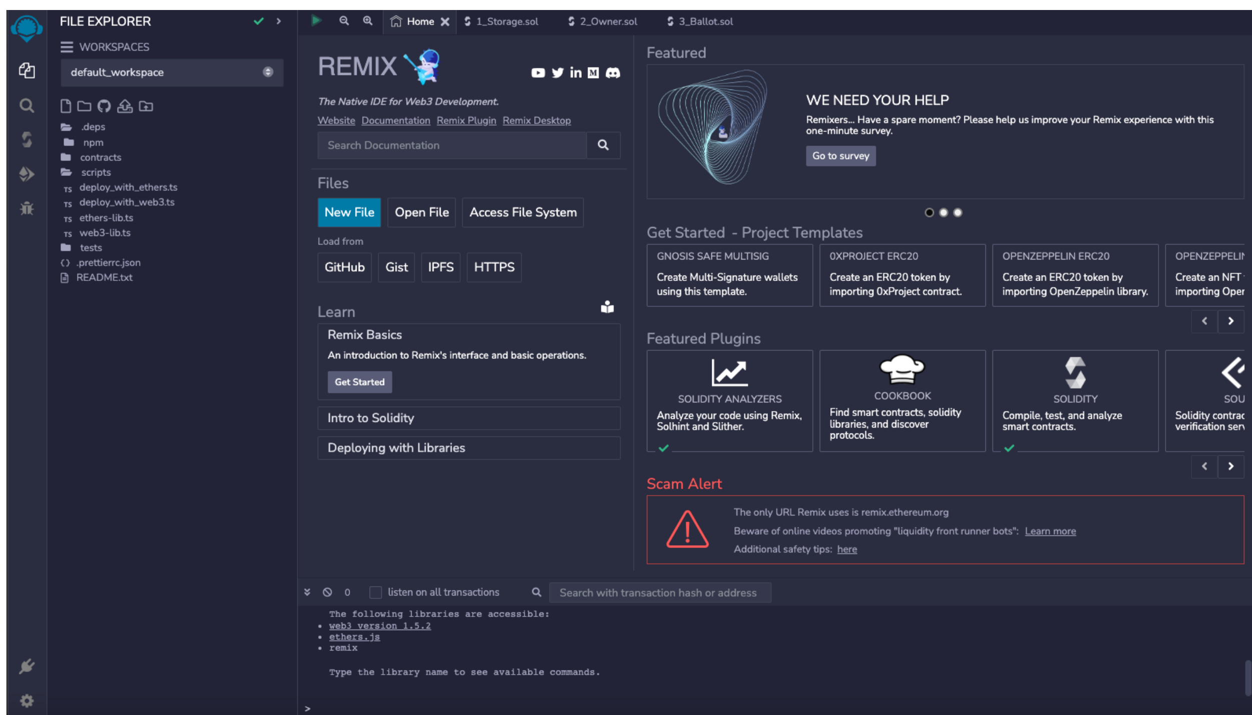

Figure 1: Remix: The Interface

```
pragma solidity >=0.7.0 <0.9.0;

contract Market {
    mapping(uint8 => uint) items;
     address payable market_owner;
    event itemSold(uint8 id);

    /// Initialize contract,
    // we assume that there is only one owner who can list items for sale.
    constructor() public {
        market_owner = payable(msg.sender);
    }

    /// list an item for sale
    function sell(uint8 item_id, uint price) public {
    if (item_id >= 0 && item_id <= 255 && price > 0)
        items[item_id] = price;
    }

    /// Buy an item.
    function buy(uint8 item_id) public payable {

        // check that the item exists and then sell
        if(items[item_id] > 0 && msg.value >= items[item_id]) {

            // mark the item as sold by setting its price to zero
            items[item_id] = 0;
            // transfer the paid currency to the market owner
            market_owner.transfer(msg.value);
        }
        else {
            emit itemSold(item_id);
        }
    }
}
```

For this tutorial, we create a new Blank workspace, and copy the Smart Contract into it (Figure ??)

Figure 2: Smart Contract Written on Remix IDE

Now that we have our smart contract written and ready, we deploy it into a Remix VM, by clicking on the Deploy and Run Transaction tab. If everything is correct, the result of the deployment transaction shows as a green tick in the execution console, and a new entry is recorded under deployed contracts(Figure ??)

Figure 3: Building and Deploying our Contract on Remix

Under deployed contracts, you can find the functions defined in this contract, with text fields allowing you to provide inputs. then click on transact (Indicated in Yellow). If you need to provide a payment with the transaction, you can do so through the value text field (indicated in Cyan) [Figure ??]. A transcript of the transaction with its success status will show in the execution area of the transaction (Figure ??)

Figure 4: Running Transactions in Remix

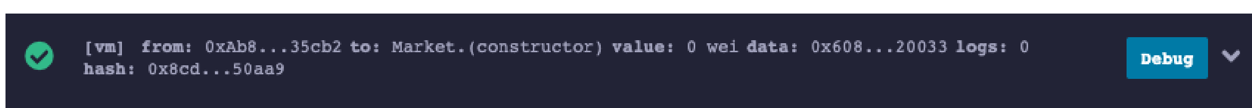

Figure 5: Running Transactions in Remix

To debug a transaction, you can click on the debug button next to it in the output transcript. This opens the debug tab in Remix, it shows different fields (like the callstack, the low level EVM code, the call stack, the state of the contract and the memory) you can have a step-by-step debugging of the transaction execution as well as stepping into functions, out and so forth, like one would do in a regular program. Also, a small cursor walks through your solidity contract to show your current line of execution. For an idea of how the debugging interface looks like, see Figure **??**

Figure 6: Debugging Transactions in Remix

2 Debugging and visualizing smart contracts using Ganache and Truffle

Sometimes, debugging a smart contract is beyond the capacity offered by Remix, especially if this smart contract is interacting with the blockchain, like the following example:

```
    //SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity >=0.7.0 <0.9.0;
  /**
   * @title Lottery
   * @dev Mohamed E. Najd <menajd@uconn.edu>
   */
contract Lottery{
    struct entry_t{
        address payable participant_address;
        string input;
    }

    mapping (address => bool) registration;
    uint registration_duration;
    uint ticket_purchase_duration;
    uint start_block;
    bool called = false;
    address payable owner;

    uint private  prize = 0;

    uint private  _balance = 0;
    entry_t[] entries;

    event Won(address a, string msg);
    event NoWin(string msg);

    constructor (uint _registrationDuration,
      uint _ticketPurchaseDuration) payable{
        registration_duration = _registrationDuration;
        ticket_purchase_duration = _ticketPurchaseDuration;
        start_block = block.number;
        prize = msg.value;
        owner = payable(msg.sender);
    }

    /**
     * register to the lottery
     *  modifiers: public
     *  args: none
     *  returns: none
     */
    function register() public{
        uint finish  = start_block + registration_duration;
        require(block.number <= finish,
          "registration is no longer open");
        registration[msg.sender] = true;
    }

    /**
     * buy ticket to the lottery
     *  modifiers: public, payable
     *  args: attempt: the string submitted by a participant
     *  returns: none
     */

    function buy_ticket(string calldata attempt) public payable{
        uint finish = start_block
                   + registration_duration
                   + ticket_purchase_duration;
        uint start = start_block + registration_duration;

        require(block.number >= start && block.number <= finish,
          "You can't buy a ticket at the moment");
        require (msg.value >= 1,
          "A Payment of 1Eth is needed for the ticket,");
        require(registration[msg.sender] == true,
          "You are not registered");

        _balance += msg.value;
        entry_t memory entry = entry_t(payable(msg.sender), attempt);
        entries.push(entry);
    }

    /**
     * redeem lottery winnings
     *  modifiers: public, payable
     *  args: none
     *  returns: none
     */
    function redeem_winnings() payable public{
        uint cutoff = start_block
                    + registration_duration
                    + ticket_purchase_duration;
```

```
        require(block.number > cutoff,
          "You cannot redeem your winnings at the moment");
        require (called == false,
          "This Function can be called only once");
        called = true;

        bool won = false;
        uint block_to_hash = start_block
                      + registration_duration
                      + ticket_purchase_duration;
        bytes32 hash = blockhash(block_to_hash);
        for (uint i = 0; i < entries.length; i++){
            bytes32 msg_hash = sha256(bytes(entries[i].input));
            if (hash[31] == msg_hash[31]){
                entries[i].participant_address.transfer(prize);
                won = true;
                emit Won(entries[i].participant_address, "won the lottery");
                break;
            }
        }
        if (!won){
          emit NoWin("Nobody won the lottery");
        }
    }

    /**
     * collect the balance after the lottery is over (admin only)
     *  modifiers: public, payable
     *  args: none
     *  returns: none
     */
    function collect_balance() payable public{
        require(msg.sender == owner,
          "Only the owner can collect the balance");
        require(called == true,
          "the lottery isn't finished yet");
        selfdestruct(owner);
    }
}
```

For these types of smart contracts, we use a set of two complementary tools (Truffle and Ganache) in order to debug and visualize the behavior of the smart contract.

2.1 Truffle

Truffle is a testing framework for smart contracts that run on the Ethereum Virtual Machine. It allows the use of breakpoints, variable analysis, and stepping through the code (like a normal debugger would).

2.1.1 How to Install

To install Truffle, you need the following requirements: Node.js (v14-v18), npm and you run the following command

```
npm install truffle
```

2.1.2 My first truffle project

To initialize a project create a new folder and in your command line application (bash, zsh, pwsh ..) go to that directory and run:

```
truffle init
```

Once this operation is completed, you'll now have a project structure with the following items:

- contracts/: Directory for Solidity contracts
- migrations/: Directory for scriptable deployment files
- test/: Directory for test files for testing your application and contracts
- truffle-config.js: Truffle configuration file

To compile a project, run:

```
truffle compile
```

2.1.3 Testing smart contracts

Using Truffle, one can write tests using either JavaScript, or Solidity. In this section, we will learn how to write unit tests using Javascript.

First, you need to obtain an abstraction of your contract from the artifacts object, you can do so as follows:

```
const LotteryAbstraction = artifacts.require("Lottery");
```

Then from that abstraction, we define a contract object as follows

```
const lotteryContract= LotteryAbstraction.new(registrationDuration,
    ticketPurchaseDuration);
```

Then, we define our test harness as follows

```
contract("Lottery", (accounts) => {
    it("register test success", async () => {
        var account_0 = accounts[0]
        await lotteryContract.register({from:account_0});
    });
    //....
})
```

Verifying the outputs and the correct behaviors of tests is usually done through assertions

```
assert.equal; assert.throws...
```

To debug your smart contract use the following command:

```
truffle test --debug
```

The debugging interface is defined through the following commands:

- o - step over
- i - step into
- ; - step instruction
- p - print instruction
- l - print additional source context
- e - print recent events
- g - turn on generated sources
- G - turn off generated sources
- h - print this help
- q - quit
- r - reset
- b - set a breakpoint
- B - remove a breakpoint
- c - continue until breakpoint
- : - evaluate and print expression
- + - add watch expression
- - - remove watch expression
- ? - list existing watch expressions and breakpoints
- v - display variables
- T - unload transaction
- t - load transaction
- y - Reset and advance to final error
- Y - Reset and advance to previous error

2.2 Visualize your work using Ganache

Ganache is a tool that allows you to deploy a local Ethereum test net where you can develop, test, and visualize the actions of your smart contract / dApp.

2.2.1 How to get it

You just need to go onto `https://trufflesuite.com/ganache/` and download the executable that runs in your operating system.

2.2.2 Configuring Truffle to run with Ganache

To start with Ganache, you need to create an Ethereum workspace and link to a truffle project as follows:

- Click on the gear icon on the upper right of your window, it should get you into your settings pane.

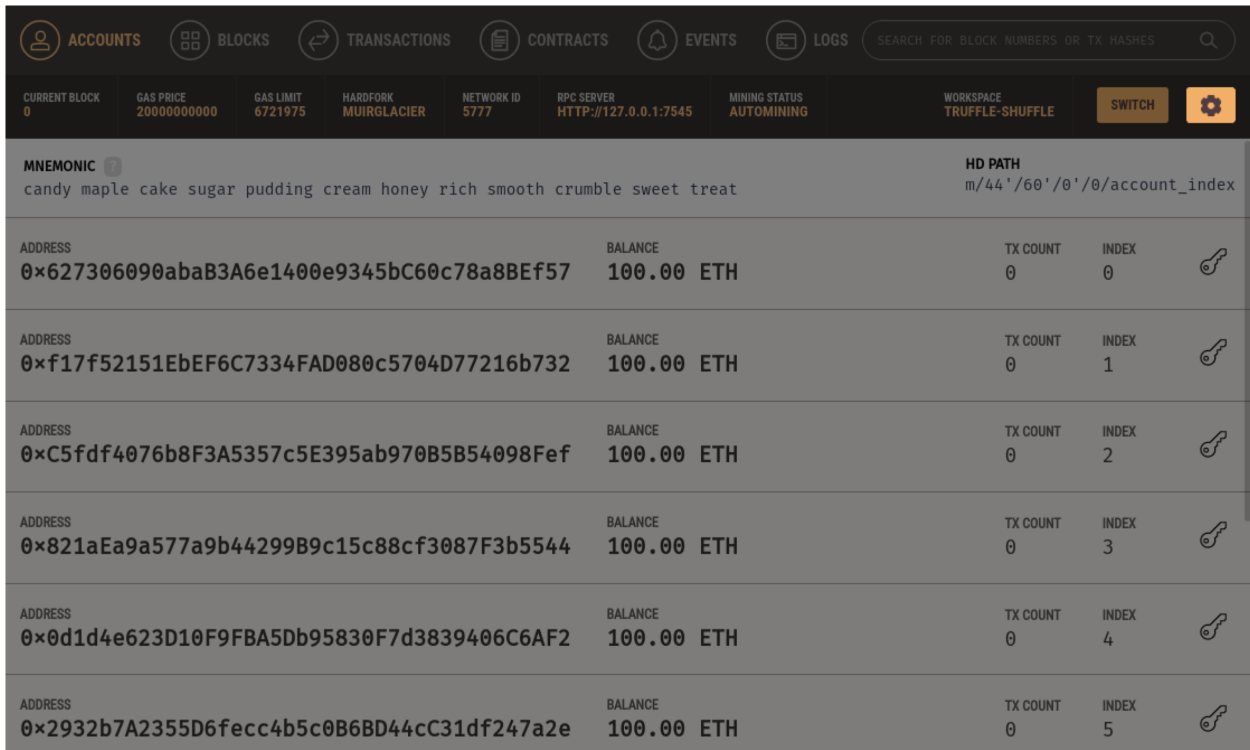

Figure 7: Settings Icon

- go to your workspace menu and click on add project

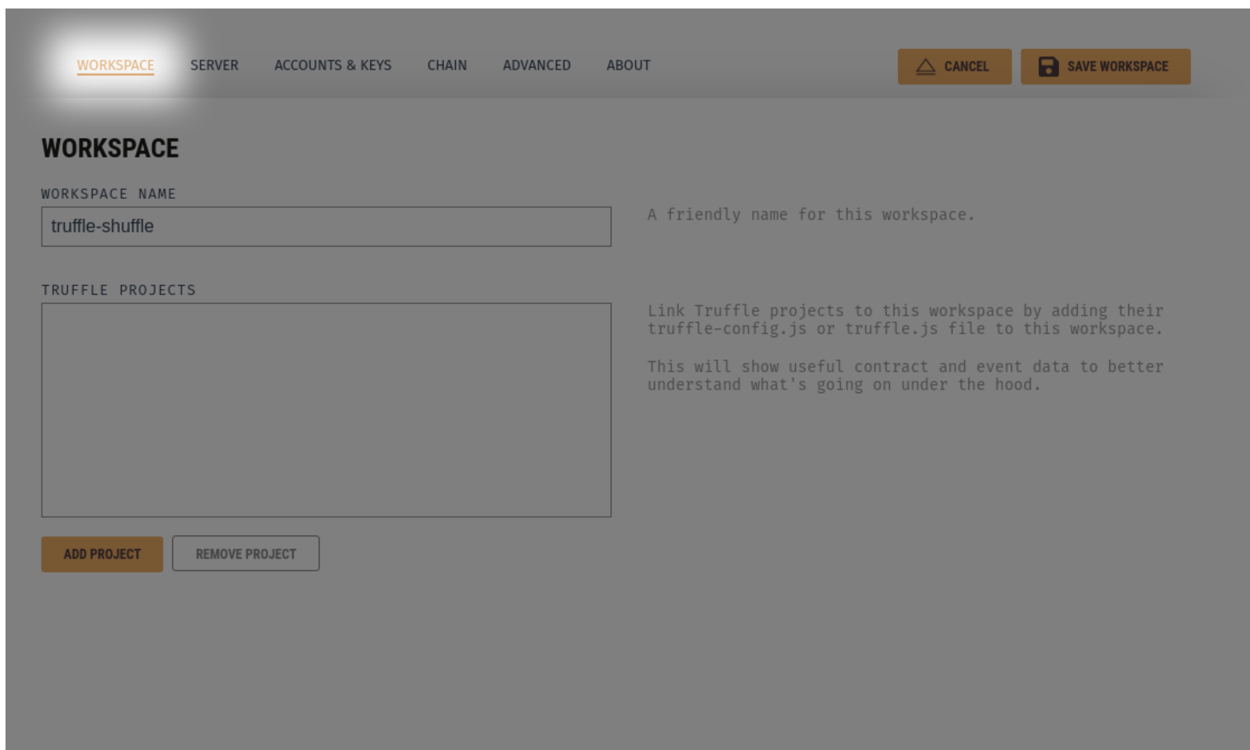

Figure 8: Workspace Settings Pane

- find the 'truffle-config.js' for the project you want to debug.

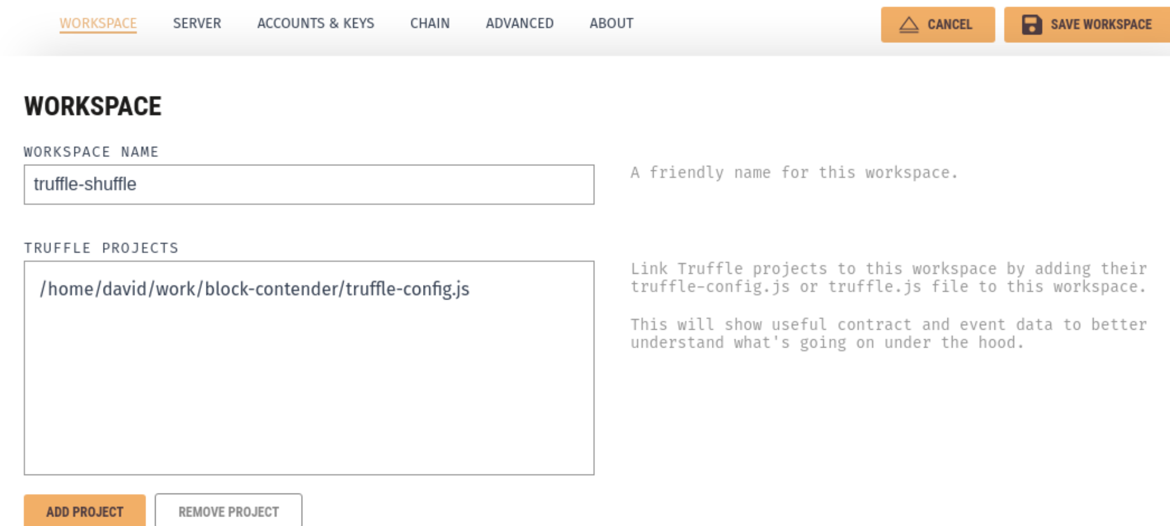

Figure 9: Project Added to Workspace

- Once you're finished, click on 'Save Workspace'.

2.3 deploy a smart contract to the Ganache Ethereum blockchain

to deploy your contract to the Ganache Ethereum blockchain, create a new migration under the `migrations/` folder as follows:

```
   // Help Truffle find ‘TruffleTutorial.sol‘ in the ‘/contracts‘ directory
const Lottery = artifacts.require("Lottery");

module.exports = function(deployer) {
  // Command Truffle to deploy the Smart Contract
  deployer.deploy(Lottery, 3, 3, {value: 1500000000000000000});
};
```

then run the following command

```
   truffle migrate
```

Now, you can interact with your smart contract through the truffle console and visualize the results on Ganache.

to open the truffle console use the following command:

```
   truffle console
```

3 Tokens On Top Of Etherum

Another functionality that Ethereum provides is the ability to run tokens (or cryptocurrencies) on top of it. In this tutorial, we explore how to run an ERC-20. To achieve this goal, we write a Solidity smart contract that corresponds to the ERC-20 specification. In other words, our contract must implement the following functions:

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns (bool
    success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender) public view returns (uint256
    remaining)
```

and the following events:

```
event Transfer(address indexed _from, address indexed _to, uint256 _value)
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

And those functions are defined as follows

- `name()`: the name given for your token.
- `symbol()`: the three or four letter symbol used for the token.
- `decimals()`: defines the small subdivision of the token
- `totalSupply()`: returns the total supply of the tokens in the chain.
- `balanceOf()`: returns the balance in tokens for a specific account.
- `transfer()`: sends tokens from the address of the initiator to another.
- `transferFrom()`: Once you approve an account to send a transaction on your behalf, they can transfer the tokens using this function. (see `approve()` below).
- `approve()`: Allow a third-party account to transfer tokens from your account
- `allowance()`: return the value a third-party account is allowed to spend on your behalf.

3.1 My First ERC-20 Token

To create an ERC-20 Token, we need two mappings, one that keeps track of the account balances, and one that keeps track of third-party allowances, as follows:

```
mapping(address => uint256) balances;
mapping(address => mapping (address => uint256)) allowances;
```

In our constructor, we define our total supply of tokens and assign them to the contract owner's balance

```
uint256 _totalSupply;
constructor(uint256 total) public {
    _totalSupply = total;
    balances[msg.sender] = _totalSupply;
}
```

Implementing `balanceOf()`, `totalSupply()`,

`decimals()`,

`symbol()`,

`name()` is trivial, and is left as an exercise to the reader. For now we implement the method to transfer tokens from a sender to another account.

```
function transfer(address _to, uint256 _value) public returns (bool success){
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender]      _value;
    balances[_to] = balances[_to] + _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
}
```

To allow third-party transfers, we implement the methods designed for this application, mainly `approve()`, `transferFrom()`.`allowance()` is trivial.

```
function approve(address _spender, uint256 _value) public returns (bool success){
    require (balance[msg.sender] >= _value)
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}

function transferFrom(address _from, address _to, uint256 _value) public returns (bool
    success){
    require(_value <= balances[_from]);
```

```
10          require(_value <= allowed[_from][msg.sender]);
11          balances[_from] = balances[_from]     _value;
12          allowed[_from][msg.sender] = allowed[_from][msg.sender]     _value;
13          balances[_to] = balances[_to] + _value;
14          emit Transfer(_from, _to, _value);
15          return true;
16      }
```