

What is a smart contract? How are they deployed?

So what are contracts, contracts are terms of agreement made for two or more parties. Now talking about smart contracts, they are digital self executing terms (when pre defined terms are met) of agreement written into code. These kinds of contracts are stored, created and built via blockchain. The blockchain platforms mostly include Ethereum. Deploying a smart contract requires interaction with a blockchain network such as Ethereum. These contracts are deployed in the following steps:

1. Programmers write the smart code with predefined terms in programming language such as Solidity.
2. The written code is then compiled in IDE's such as remix
3. By using a wallet we then connect to the blockchain. After this a deployment script is executed and a transaction is sent which includes the compiled Solidity code.
4. Once a transaction is mined the smart contract is finally deployed

What is gas? Why is gas optimization such a big focus when building smart contracts?

Just like other systems used to measure the values of certain physical quantities in Physics, gas is a unit of measurement in the blockchain world which is used to measure the computational power required to execute the code/operation. Gas optimization is such a big focus due to the following reasons:

- As gas fees are paid in Ether, using high gas consumption will result in higher costs.
- If there is high gas consumption by multiple networks then the network might be congested which will lead to slower transactions.
- If used efficiently gas will help to allow for more transactions to be processed.

What is a hash? Why do people use hashing to hide information?

In the computer world, hash function is a method to encrypt the text to a fixed string of certain bytes. A hash is the output generated by the hash function. For example I enter a “Hello world” into a hash function `hashFunction(input)` then this hash function will return an output such as “2jadlsvnull9nfadfna;-x”. People use hashing to hide information due to various reasons. Some of them are listed below:

- Hashes can verify data integrity by comparing the hash of the original data with the hash of the received data. If they match, the data is genuine.
- Hashing can securely store sensitive information. As hash encrypts the data, data such as password pins etc can easily be stored.
- Hashes provide a quick way to compare large amounts of data. As systems compare the hash values rather than the data itself, the comparison is faster and requires less memory.

How would you prove to a colorblind person that two different coloured objects are actually of different colours?

I would prove to the colour blind person in the following steps

Step 1: Firstly I would ask the colorblind person to randomly shuffle the two Objects (Objects A and Object B) randomly. To insure more difficulty I would look in the opposite direction.

Step 2: When I have to distinguish the objects, I can easily identify the two Objects by their colour.

Step 3: Repeat the above two steps multiple times to ensure that I am consistent to the colour blind person.

By following the above steps I can prove to a colorblind person that two different coloured objects are actually of different colours

Question 1 of Solidity exercise

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is visible. The 'Solidity compiler' section shows the file 'helloWorld - helloWorld.sol'. The 'Deploy' button is highlighted. Below it, the 'Transactions recorded' section shows a list of transactions. The 'Pinned Contracts' section shows 'No pinned contracts found for selected workspace & network'. The 'Deployed/Unpinned Contracts' section shows a list of contracts, including 'HELLOWORLD AT 0xD2A...FD'. The 'storeNumber' function is selected, and the 'number' parameter is set to '32'. The 'transact' button is highlighted. The main editor shows the Solidity code for the 'helloWorld' contract. The code includes a pragma statement for Solidity version 0.7.0, a contract definition for 'helloWorld', a public function 'storeNumber' to store a number, and a public view function 'retrieveNumber' to retrieve the stored number. The bottom status bar shows the current transaction details, including the gas used and the transaction hash.

```
1 // SPDX-License-Identifier: GPL-3.0
2 //solidity version is specified here
3 pragma solidity ^0.7.0 ^0.9.0;
4
5
6 //creating a contract named helloWorld
7 contract helloWorld {
8
9     //creating a string named greetings
10    string public greetings = "Hello this is assignment question 1";
11    //creating a private number which is used to store the number when passed from the parameter in a function
12    uint private number;
13
14
15    //function to store a number. the function is public as it stores the number
16    function storeNumber(uint _number) public {
17        number = _number;
18    }
19
20    //function to retrieve a number. The view keyword here indicates that the function will not modify state and the "returns (uint)" says that it will return
21    function retrieveNumber() public view returns (uint) {
22        return number;
23    }
24 }
```

Question 4 of Solidity exercise

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is visible. The 'Solidity compiler' section shows the file 'contract-a401ebc115.sol'. The 'Deploy' button is highlighted. Below it, the 'Transactions recorded' section shows a list of transactions. The 'Pinned Contracts' section shows 'No pinned contracts found for selected workspace & network'. The 'Deployed/Unpinned Contracts' section shows a list of contracts, including 'BALLOT AT 0xF8E...3FBE8 (ME)' and 'BALLOT AT 0xD7A...F771B (ME)'. The 'delegate' function is selected, and the 'address' parameter is set to '0x5838da6a701c568545dcfc803fc8875f56bedd4'. The 'transact' button is highlighted. The main editor shows the Solidity code for the 'Ballot' contract. The code includes a pragma statement for Solidity version 0.8.11, a contract definition for 'Ballot', a public function 'vote' to cast a vote, a public view function 'winningProposal' to get the winning proposal, and a public view function 'winnerName' to get the winner's name. The bottom status bar shows the current transaction details, including the gas used and the transaction hash.

```
103 // Give your vote (including votes delegated to you)
104 // to proposal 'proposals[proposal].name'
105 function vote(uint proposal) external voteEnded {
106     Voter storage sender = voters[msg.sender];
107     require(sender.weight != 0, "Has no right to vote");
108     require(!sender.voted, "Already voted.");
109     sender.voted = true;
110     sender.vote = proposal;
111
112     // If 'proposal' is out of the range of the array,
113     // this will throw automatically and revert all
114     // changes.
115     proposals[proposal].voteCount += sender.weight;
116
117     /// @dev Computes the winning proposal taking all
118     /// previous votes into account.
119     function winningProposal() public view
120         returns (uint winningProposal_) {
121         uint winningVoteCount = 0;
122         for (uint p = 0; p < proposals.length; p++) {
123             if (proposals[p].voteCount > winningVoteCount) {
124                 winningVoteCount = proposals[p].voteCount;
125                 winningProposal_ = p;
126             }
127         }
128     }
129
130     // Calls winningProposal() function to get the index
131     // of the winner contained in the proposals array and then
132
133     You may want to cautiously increase the gas limit if the transaction went out of gas.
134     call to Ballot.winnerName
135
136     [call] from: 0x5838da6a701c568545dcfc803fc8875f56bedd4 to: Ballot.winnerName() data: 0xe2b...a53f0
```

Github Link: [Sachyamd](#)