Interview

1. What is Ethereum smart contract?

It is a clock of code that runs on Ethereum blockchain. Consisting of functions and states which resides on a specific address on the Ethereum blockchain.

1. How is ether better smart contract better than other programming language?  
   Once the smart contracts are being deployed, they cannot be stopped even by the owner, hacked or modified. They are immutable.
2. Can a smart contract interact other smart contract?  
   Yes, by using the import statement you can import various smart contract that already been deployed. Carry out chain of execution through the blockchain.
3. Can a solidity smart contract call an API on the web?  
   No, the smart contract can run its own code and interact with another smart contract. If you want your smart contract
4. Can solidity smart contract store a lot of data?  
   No, as each transaction have some cost involved like a gas. The gas consumption
5. Smart contract in other language?  
   Yes, Vyper, LLL, etc
6. Solidity is Dynamic or static?  
   Static. Which means the variable type needs to be defined in solidity opposite to Python or JavaScript where you do not define and language figures out what data type it should be.
7. Solidity compiled or interpreted?  
   Compiled. Because the code needs to be first compiled and then can be deployed.
8. File extension? .sol
9. Can a single solidity file have several smart contract?  
   yes. Use contract keyword every time
10. Layout of solidity smart contract?  
    pragma solidity ^0.8.0;  
    contract ConrtactNam{    
    state Variable;  
    function functionName(){ }}
11. What is the difference between state and local variables?
12. What are the 4-memory location?  
    Storage, memory, stack
13. What can source file contain?  
    Source files can contain an arbitrary number of [contract definitions](https://docs.soliditylang.org/en/v0.5.10/structure-of-a-contract.html#contract-structure), [import](https://docs.soliditylang.org/en/v0.5.10/layout-of-source-files.html#import) directives and [pragma directives](https://docs.soliditylang.org/en/v0.5.10/layout-of-source-files.html#pragma).
14. What is pragma?  
    The pragma keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in all of your project. If you [import](https://docs.soliditylang.org/en/v0.5.10/layout-of-source-files.html#import) another file, the pragma from that file does not automatically apply to the importing file. version —> 0.x.0 or x.0.0 if the pragma solidity ^0.5.2; —>A source file with the line above does not compile with a compiler earlier than version 0.5.2, and it also does not work on a compiler starting from version 0.6.0
15. What is the default visibility of state variable ?  
    Private
16. What is difference between address and address payable ?  
    The address is a data type for variable and address payable is also defined data variable with a payable option ie, solidity can only send ether to payable addresses.
17. Is it necessary to make and address payable to transfer ERC20 tokens?  
    No, you only need payable if you need to send ether. Tokens are different.
18. What is main change between Solidity 0.5.x VS 0.4.x?  
    constructor declaration. After 0.5.x we started using keyword contractor.Address payable and address Var is removed  emit keyword added memory location explicit for dynamic-length types function visibility also needs be explicit
19. Give 3 ways to save gas? Put less data on chain use events instead of storage Optimal order for variable declaration
20. How will you optimally order uint128, bytes32 and uint128 to save gas? Now you will have to understand —>  256 bit = 32 bytes = 1 word  uint8 = 1 byte  And storage is continuous: rest read  Gas optimisation is a challenge that is unique to developing Ethereum smart contracts. To be successful, we need to learn how Solidity handles our variables and functions under the hood.  **Optimising variables**

**Variable packing**

Solidity contracts have contiguous 32 byte (256 bit) slots used for storage. When we arrange variables so multiple fit in a single slot, it is called variable packing.

Variable packing is like a game of Tetris. If a variable we are trying to pack exceeds the 32 byte limit of the current slot, it gets stored in a new one. We must figure out which variables fit together the best to minimise wasted space.

Because each storage slot costs gas, variable packing helps us optimise our gas usage by reducing the number of slots our contract requires.

Let’s look at an example:

uint128 a;

uint256 b;

uint128 c;

These variables are not packed. If b was packed with a, it would exceed the 32 byte limit so it is instead placed in a new storage slot. The same thing happens with c and b.

uint128 a;

uint128 c;

uint256 b;

These variables are packed. Because packing c with a does not exceed the 32 byte limit, they are stored in the same slot.

Keep variable packing in mind when choosing data types — a smaller version of a data type is only useful if it helps pack the variable in a storage slot. If a uint128 does not pack, we might as well use a uint256.  Variable packing can only be done on storage . And cannot be done in memory and call data say local variables and variable passed as a function argument.  **Reference data types** Structs and arrays always begin in a new storage slot — however their contents can be packed normally. A uint8 array will take up less space than an equal length uint256 array.

It is more gas efficient to initialise a tightly packed struct with separate assignments instead of a single assignment. Separate assignments makes it easier for the optimiser to update all the variables at once.

Initialise structs like this:

Point storage p = Point()

p.x = 0;

p.y = 0;

Instead of:

Point storage p = Point(0, 0);  **Inheritance** When we extend a contract, the variables in the child can be packed with the variables in the parent.

The order of variables is determined by [C3 linearization](https://en.wikipedia.org/wiki/C3_linearization). For most applications, all you need to know is that child variables come after parent variables.

**Data types**

We have to manage trade-offs when selecting data types to optimize gas. Different situations can make the same data type cheap or expensive.

**Memory vs. Storage** Performing operations on memory — or call data, which is similar to memory — is always cheaper than storage.

A common way to reduce the number of storage operations is manipulating a local memory variable before assigning it to a storage variable.

We see this often in loops:

uint256 return = 5; // assume 2 decimal places

uint256 totalReturn;function updateTotalReturn(uint256 timesteps) external {

uint256 r = totalReturn || 1;   
for (uint256 i = 0; i < timesteps; i++) {

r = r \* return;

} totalReturn = r;

}

In calculateReturn, we use the local memory variable r to store intermediate values and assign the final value to our storage variable totalReturn.

**Fixed vs. Dynamic** Fixed size variables are always cheaper than dynamic ones.

If we know how long an array should be, we specify a fixed size:

uint256[12] monthlyTransfers;

This same rule applies to strings. A string or bytes variable is dynamically sized; we should use a byte32 if our string is short enough to fit.

If we absolutely need a dynamic array, it is best to structure our functions to be additive instead of subractive. Extending an array costs constant gas whereas truncating an array costs linear gas.

**Mapping vs. Array** Most of the time it will be better to use a mapping instead of an array because of its cheaper operations.

However, an array can be the correct choice when using smaller data types. Array elements are packed like other storage variables and the reduced storage space can outweigh the cost of an array’s more expensive operations. This is most useful when working with large arrays.

**Other techniques**

There are a few other techniques when working with variables that can help us optimize gas cost.

**Initialization** Every variable assignment in Solidity costs gas. When initializing variables, we often waste gas by assigning default values that will never be used.

uint256 value; is cheaper than uint256 value = 0;.

**Require strings** If we are adding message strings to require statements, we can make them cheaper by limiting the string length to 32 bytes.

**Unpacked variables** The EVM operates on 32 bytes at a time, variables smaller than that get converted. If we are not saving gas by packing the variable, it is cheaper for us to use 32 byte data types such as uint256.

**Deletion** Ethereum gives us a gas refund when we delete variables. Its purpose is an incentive to save space on the blockchain, we use it to reduce the gas cost of our transactions.

Deleting a variable refunds 15,000 gas up to a maximum of half the gas cost of the transaction. Deleting with the delete keyword is equivalent to assigning the initial value for the data type, such as 0 for integers.

**Storing data in events** Data that does not need to be accessed on-chain can be stored in events to save gas.

While this technique can work, it is not recommended — events are not meant for data storage. If the data we need is stored in an event emitted a long time ago, retrieving it can be too time consuming because of the number of blocks we need to search.

<https://medium.com/coinmonks/inheritance-in-solidity-debunked-3d8dd32d3a99> <https://medium.com/coinmonks/gas-optimization-in-solidity-part-i-variables-9d5775e43dde> **Inheritance in Solidity demystified with analysis of depth-first and breadth-first resolution of contracts**

* Concatenate two strings in solidity ?   pragma solidity ^0.4.14;

import "https://github.com/Arachnid/solidity-stringutils/blob/master/src/strings.sol";

contract C {

using strings for \*;

string public s;

string public res;

function foo(string s1, string s2) {

//1. First way use this

s = s1.toSlice().concat(s2.toSlice());

// second way is to use

res = string(abi.encodePacked(s1,s2));

} }  **Non-standard Packed Mode**

Through abi.encodePacked(), Solidity supports a non-standard packed mode where:

* types shorter than 32 bytes are neither zero padded nor sign extended and
* dynamic types are encoded in-place and without the length.

This packed mode is mainly used for indexed event parameters.

As an example, the encoding of int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!") results in:

0xffff42000348656c6c6f2c20776f726c6421

^^^^ int16(-1)

^^ bytes1(0x42)

^^^^ uint16(0x03)

^^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field

**More specifically:**

* Each value type takes as many bytes as its range has.
* The encoding of a struct or fixed-size array is the concatenation of the encoding of its members/elements without any separator or padding.
* Mapping members of structs are ignored as usual.
* Dynamically-sized types like string, bytes or uint[] are encoded without their length field.

In general, the encoding is ambiguous as soon as there are two dynamically-sized elements, because of the missing length field.

If padding is needed, explicit type conversions can be used: abi.encodePacked(uint16(0x12)) == hex"0012".

Since packed encoding is not used when calling functions, there is no special support for prepending a function selector. Since the encoding is ambiguous, there is no decoding function.

1. How to get length ?  
    string name;   
   bytes memory bytestr = bytes(name);   
   bytestr.length;
2. Create smart contract from a smart contract?

Graphical user interface, text

Description automatically generated

1. Call smart contract function from another smart contract

Graphical user interface, text, application, email

Description automatically generated

1. how to get the address of a smart contract that was deployed from a smart contract?  
     
   address ChildContract = address(new Child());
2. What will be the value of msg.sender if a contract calls another one?  
   Graphical user interface, text, application

   Description automatically generated  
     
   msg.sender = address creating call. tx.origin = original address creating call. (ie, always the user, since contracts can't automatically issue their own transactions, yet).

Thus if only a user is interacting with a contract, then tx.origin == msg.sender. If the contract (A) calls contract (B), then msg.sender in contract B == Contract A.

1. How to transfer ERC20 token  
   Graphical user interface, text, application

   Description automatically generated
2. How to declare and emit an event?

contract eventContract{

event Log(address sender, string message);

event ActiveLog();

function fireEvent() public{

emit Log(msg.sender, "Hello Vinee");

emit Log(msg.sender, "Hello Aman");

emit Log(msg.sender, "Yash Welcome");

}

# .send(...) vs. .transfer(...)

# What is indexed keyword in in event?

# How many event fields can be marked indexed? 3

# Is It possible for smart contract to read the events emitted before? No

# Is it possible to delete or modify past event ? No , they are immutable.

# In solidity, how to do like a JavaScript console.log for debugging ? No you cannot use console log. But yes you can use events to emit the messages.

# Implement access control without modifier?

# Access control with a modifier ?

|  |
| --- |
| pragma solidity ^0.8.0;contract ChecKAccess{address public owner;constructor() public{owner = msg.sender;}function transferOwner(address \_owner) public{require(msg.sender == owner, "Not an Owner");owner = \_owner;}}// contract with modifiercontract checkOwnerAndTransfer{address public currentOwner;constructor() {currentOwner = msg.sender;}modifier onlyOwner(){require(msg.sender == currentOwner, "Not an owner");\_;}function transfer(address \_currentOwner) public onlyOwner{currentOwner = \_currentOwner;}} |

# How to cancel transaction ?

**You cannot cancel a transaction once the transaction is been carried out.** It can't be canceled, but it can be "out-gassed". You can grab the nonce (can be found on the pending tx on etherscan) and send another transaction with a higher gas price with the **same nonce**. If you use a different nonce, then they can both be mined, but if you use the same nonce, only one can be mined. If miners see both, they should be smart enough to choose to mine the tx with the higher gas price.  
  
you can also : A pending transaction can be cancelled by using a wallet that supports it.

<https://ethereum.stackexchange.com/questions/31298/is-it-possible-to-cancel-a-transaction>

1. What is ABIEncoderV2 pragma statement?

Enables experimental solidity features which are not yet enabled for example it enables to written a struct from a function called externally which is not possible yet in standard solidity

1. Is it safe to use ABIEncoder V2?  
   No should only be used in development not in production.
2. Is it possible to send transaction without requiring users to pay gas?  
   Yes, it is called gassless transaction

First you will ask the users to sign a message on front end then the message and the signature will be sent to a centralized back end that would be in your application and that would be off chain and this blockchain will create a transaction and embed the payloads and embed the payloads of the message plus the signature into it. That means the gas fees will be covered by the wallet of the app instated of the user wallet on chain a smart contract will veify the validity of the signature and perform an operation on behalf of the user /   
<https://medium.com/coinmonks/how-to-implement-gas-less-transactions-on-ethereum-9f9273d2f059>

1. Which solidity function would you use to verify a signature ?

EC recover

1. Library in solidity ?

Libraries are like contract but there is no storage

1. Hence you cannot store any state variables or ether cant be send.
2. Helps you to keep DRY (don’t repeat yourself)
   1. Add functionality types say   
      uint x   
      x.myLibraryFunction()
3. Can save gas so this because once you create a library say myLibrary and use it in contract A 🡪 first you will deploy the library and then the contract which has used the library after linking with the library.  
   Now you want to use and link contract B to myLibrary 🡪 now you do not need to deploy the library just the contract B. hence we are deploying the library just once. Which will save the gas price.

Reusable piece of code , two types of libraries . Deployed and embedded. Deployed libraries have there own addresses and can be used by several other smart contracts. Embedded do not have own addresses but deployed as a part of the code of the smart contract that use them. Library don’t have there own storage They only provide function for other smart contract.

Text

Description automatically generated

1. When is a library embedded VS deployed   
   internal once are embedded while public are deployed internal functions can only be called by smart contract while public can be called anyone  
   Graphical user interface, text, application, email

   Description automatically generated

# Re-entrancy attack? <https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21>

# Text, application, email Description automatically generated

# <https://jeancvllr.medium.com/solidity-tutorial-all-about-bytes-9d88fdb22676> <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol>

# Update the balance before calling other contract

# Put in place re-entrancy guard

# Limit gas available in call contract

# How to produce hash multiple values in solidity ? keccak256(abi.ecodePacked(a,b,c)) 🡪 here the arguments are the data that you want to has

|  |
| --- |
| pragma solidity ^0.4.14;contract HashValue{string a = "You are";uint256 b = 100;function func() external view returns (bytes32) {return keccak256(abi.encodePacked(a)) ;}}//hash code --> {// "0": "bytes32: 0x79fc753be4f734a14ed44b04cb23a4d25ee59c9e743f47c2effbaec3ecacf43b"// } |

# <https://ethereum.stackexchange.com/questions/72199/testing-sha256abi-encodepacked-argument>

# Generate random integer in solidity?

|  |
| --- |
| pragma solidity ^0.8.0;contract RandomGenerator {function generate() public view returns(uint){return uint(keccak256(abi.encodePacked(block.timestamp, block.difficulty)));}} |

# What are two kinds of assembly in solidity ? functional , instructional : functional is codes instruction is set of rules

# How to declare assembly code? assembly { }

# Create a function to determine if another address is a contract or regular address ?

# Signature verification: (read later)

# Keccak256 (cryptographic Hash function)

# Function takes in arbitrary size input and outputs data of fixed size

# Properties:

# Deterministic

# Hash(x) = h, ie no matter what time Hash(x) will always be equal to h.

# Quick to compute the hash.

# Irreversible

# Given h we cannot find x such that Hash(x) = h

# Small change in input will change the output significantly.

# Collision resistant 🡪 hard to find x and y , such that hash(x) = hash(y)

# Bytes32 ans = Keccak256(abi.encodePacked(“hello”, uint(1), uint(2), address(123)) this would take inputs and turn them into bytes. This would turn ans to 32 bytes hash.

# contract HashValue{

# string a = "You are";

# uint256 b = 100;

# function func() external view returns (bytes32) {

# return keccak256(abi.encodePacked(a)) ;

# }

# }

# //hash code --> {

# // "0": "bytes32: 0x79fc753be4f734a14ed44b04cb23a4d25ee59c9e743f47c2effbaec3ecacf43b"

# // }

# If I add b

# contract HashValue{

# string a = "You are";

# uint256 b = 100;

# function func() external view returns (bytes32) {

# return keccak256(abi.encodePacked(a,b)) ;

# }

# }

# // this will create a hash code 0x3f1784fc7ddc9f738edbf4aa343984bbd47dbb1861375cec240f4ac129326fb1

# // and if I remove b again it will generate the hash as before.

# 0x79fc753be4f734a14ed44b04cb23a4d25ee59c9e743f47c2effbaec3ecacf43b

# The keccak256 is collision resistant, but we can create collision

# Now the way is to pass two string for example

# String a = AA,

# string b = CC return keccak256(abi.encodePacked(a,b))

# the keccak256 will concatenate two string 🡪 AACC

# and if you pass string a = A and string b = ACC then again it will concatenate creating 🡪 AACC this will give you same hash as the values have become same now.

# When you encode more than one dynamic data type you should always call encode rather than encodePacked

# 