upGrad

Interview Questions:
Application Development on Ethereum

Q1	What does the frontend use to connect to the Ethereum backend (smart contracts)?
Ans	Web3.js is used as an interface to connect the frontend to the backend of any blockchain application.

Q2	What is ABI used for?
Ans	ABI is a description of the public interface of a contract which is used by DApps for invoking the contract.

Q3	Let’s say you have a Test.sol file. Deploy this contract and save the contract address in a variable without using truffle.

Ans

```
myContract.deploy({
  data: '0x12345...',
  arguments: [123, 'My String']
})
.send({
  from:
'0x1234567890123456789012345678901234567891',
  gas: 1500000,
  gasPrice: '30000000000000'
}, function(error, transactionHash){ ... })
.on('error', function(error){ ... })
.on('transactionHash', function(transactionHash){ ... })
.on('receipt', function(receipt){
  console.log(receipt.contractAddress) // contains the new
contract address
})
.on('confirmation', function(confirmationNumber, receipt){ ...
})
.then(function(newContractInstance){
```

	console.log(newContractInstance.options.address) // *instance with the new contract address*
	});
	
	Here, data is the bytecode of the contract which is generated by compiling the Test.sol file with solc.
	`var solc = require('solc');`
	
	`var input = {`
	`  language: 'Solidity',`
	`  sources: {`
	`    'Test.sol': {`
	`      content: 'contract Test { function f() public { } }'`
	`    }`
	`  },`
	`  settings: {`
	`    outputSelection: {`
	`      '*': {`
	`        '*': ['*']`
	`      }`
	`    }`
	`  }`
	`};`
	
	`var output =`
	`JSON.parse(solc.compile(JSON.stringify(input)));`
	
	`` // `output` here contains the JSON output as specified in the documentation ``
	`for (var contractName in output.contracts['test.sol']) {`
	`  console.log(`
	`    contractName +`
	`      ': ' +`
	
	`output.contracts['Test.sol'][contractName].evm.bytecode.object`
	`  );`
	`}`

Q4	Write the command to create an Ethereum wallet address in JavaScript without using truffle.
Ans	web3.eth.accounts.create();

Q5	Is sending one ether like this '.send({ value: 1 })' okay?
Ans	No, you send 1 wei. Transactions always work with wei.

Q6	What will be the output of the following code?
	`const arr = [10, 12, 15, 21];`
	`for (var i = 0; i < arr.length; i++) {`
	`  setTimeout(function() {`
	`    console.log('Index: ' + i + ', element: ' + arr[i]);`
	`  }, 3000);`
	`}`

Reference	https://medium.com/coderbyte/a-tricky-javascript-interview-question-asked-by-google-and-amazon-48d212890703
Ans	Index: 4, element: undefined(printed 4 times).

Q7	What is hoisting?

Ans	Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Remember that JavaScript only hoists declarations, not initialisation. Let's take a simple example of variable hoisting:
	
	`console.log(message); //output : undefined`
	`var message = 'The variable has been hoisted';`

Q8	What is the difference between call, apply and bind?

Ans	The difference between call, apply and bind can be explained with the following examples.
	
	Call: The call() method invokes a function with a given value and arguments provided one by one.
	
	`var employee1 = {firstName: 'John', lastName: 'Rodson'};`
	`var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};`
	
	`function invite(greeting1, greeting2) {`
	`  console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);`
	`}`
	
	`invite.call(employee1, 'Hello', 'How are you?'); // Hello John Rodson, How are you?`
	`invite.call(employee2, 'Hello', 'How are you?'); // Hello Jimmy Baily, How are you?`
	
	Apply: The apply() method invokes the function and allows you to pass in arguments as an array.
	
	`var employee1 = {firstName: 'John', lastName: 'Rodson'};`
	`var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};`
	
	`function invite(greeting1, greeting2) {`
	`  console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);`
	`}`
	
	`invite.apply(employee1, ['Hello', 'How are you?']); // Hello John Rodson, How are you?`

```
invite.apply(employee2, ['Hello', 'How are you?']); //
Hello Jimmy Baily, How are you?
```

Bind: The bind() method returns a new function, allowing you to pass in an array and any number of arguments.

```
var employee1 = {firstName: 'John', lastName:
'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName:
'Baily'};

function invite(greeting1, greeting2) {
  console.log(greeting1 + ' ' + this.firstName + ' ' +
this.lastName+ ', '+ greeting2);
}

var inviteEmployee1 = invite.bind(employee1);
var inviteEmployee2 = invite.bind(employee2);
inviteEmployee1('Hello', 'How are you?'); // Hello
John Rodson, How are you?
inviteEmployee2('Hello', 'How are you?'); // Hello
Jimmy Baily, How are you?
```

Call and apply are pretty interchangeable. Both execute the current function immediately. You need to decide whether it is easier to send in an array or a comma separated by a list of arguments. You can remember them as call for comma (separated list) and apply for array. On the other hand, bind creates a new function that will have this set to the first parameter passed to bind().

Q9	What is the strict mode in JavaScript? How do you declare it ?
Ans	Strict mode is a new feature in ECMAScript 5 that allows you to place a program, or a function in a 'strict' operating context. This way it prevents certain actions from being taken and throws more exceptions. The literal

	expression ‘use strict’; instructs the browser to use the JavaScript code in the strict mode.
	`"use strict";`
	`x = 3.14; // This will cause an error because x is not declared`

Q10	What will the following code output to the console and why?
	`console.log(1 +  "2" + "2");`
	`console.log(1 +  +"2" + "2");`
	`console.log(1 +  -"1" + "2");`
	`console.log(+"1" +  "1" + "2");`
	`console.log( "A" - "B" + "2");`
	`console.log( "A" - "B" + 2);`

Ans	The code will output the following to the console: "122" "32" "02" "112" "NaN2" NaN Here is the reason: The fundamental issue here is that JavaScript (ECMAScript) is a loosely typed language, and it performs automatic type conversion on values to accommodate the operation being performed. Let’s see how this plays out with each of the mentioned examples. Example 1: 1 + "2" + "2" Output: "122" Explanation: The first operation to be performed is 1 + "2". Since one of the operands ("2") is a string, JavaScript assumes it needs to perform a string concatenation and therefore converts the type of 1 to "1", 1 + "2" yields "12". Then, "12" + "2" yields "122".

Example 2: 1 + +"2" + "2" Output: "32"
Explanation: Based on the order of operations, the first operation to be performed is +"2" (the extra + before the first "2" is treated as a unary operator). Thus, JavaScript converts the type of "2" to a numeric value and then applies the unary + sign to it (i.e., treats it as a positive number). As a result, the next operation is now 1 + 2 which of course yields 3. However, we have an operation between a number and a string (i.e., 3 and "2"). So, once again, JavaScript converts the type of the numeric value to a string and performs a string concatenation, yielding "32".

Example 3: 1 + -"1" + "2" Outputs: "02"
Explanation: The explanation here is identical to the prior example, except the unary operator is - rather than +. So "1" becomes 1, which then becomes -1 when the - is applied, which is then added to 1 yielding 0. This is then converted to a string and concatenated with the final "2" operand, yielding "02".

Example 4: +"1" + "1" + "2" Outputs: "112"
Explanation: Although the first "1" operand is typecast to a numeric value based on the unary + operator that precedes it, it is then immediately converted back to a string when it is concatenated with the second "1" operand, which is then concatenated with the final "2" operand, yielding the string "112".

Example 5: "A" - "B" + "2" Outputs: "NaN2"
Explanation: Since the - operator cannot be applied to strings, and since neither "A" nor "B" can be converted to numeric values, "A" - "B" yields NaN which is then concatenated with the string "2" to yield “NaN2”.

Example 6: "A" - "B" + 2 Outputs: NaN
Explanation: As explained in the previous example, "A" - "B" yields NaN. However, any operator applied to NaN with any other numeric operand will still yield NaN.

Q11	What will be the output of this code?
	
	`var x = 21;`
	`var girl = function () {`
	`  console.log(x);`
	`  var x = 20;`
	`};`
	`girl ();`

Ans	Neither 21 nor 20; the result is undefined. This is because JavaScript initialisation is not hoisted. (Why does it not show the global value of 21? The reason is that when the function is executed, it checks whether there is a local x variable present but does not yet declare it, So, it will not look for a global one.)

Q12	What is the output of the following code? Explain your answer.
	
	`var a={},`
	`  b={key:'b'},`
	`  c={key:'c'};`
	`a[b]=123;`
	`a[c]=456;`
	`console.log(a[b]);`

Ans	The output of this code will be 456 (not 123). The reason for this is when setting an object property, JavaScript will implicitly stringify the parameter value. In this case, since b and c are both objects, they will both be converted to "[object Object]". As a result, a[b] and a[c] are both equivalent to a["[object Object]"] and can be used interchangeably. Therefore, setting or referencing a[c] is precisely the

	same as setting or referencing a[b].

Q13	What do the following lines output, and why?
	`console.log(1 < 2 < 3);`
	`console.log(3 > 2 > 1);`

Ans	The first statement returns true which is as expected. The second returns false because of how the engine works with regards to operator associativity for < and >. It compares left to right. So, 3 > 2 > 1 JavaScript translates to true > 1. True has a value 1, so it then compares 1 > 1, which is false.

Q14	Write a sum method which will work properly when invoked using either of the following syntax.
	`console.log(sum(2,3));   // Outputs 5`
	`console.log(sum(2)(3));  // Outputs 5`

Ans	`function sum(x, y) {`
	` if (y !== undefined) {`
	`  return x + y;`
	` } else {`
	`  return function(y) { return x + y; };`
	` }`
	`}`

Q15	What is memoization?

Ans

Memoization is a programming technique which attempts to increase a function’s performance by caching its previously computed results. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned without executing the entire function. Otherwise, the function is executed and then the result is added to the cache. Let's take an example of adding function with memoization.

```
const memoizAddition = () => {
  let cache = {};
 return (value) => {
  if (value in cache) {
   console.log('Fetching from cache');
   return cache[value]; // Here, cache.value cannot be used as property
name as starts with the number which is not a valid JavaScript  identifier.
Hence, can only be accessed using the square bracket notation.
  }
  else {
   console.log('Calculating result');
   let result = value + 20;
   cache[value] = result;
   return result;
  }
 }
}
// returned function from memoizAddition
const addition = memoizAddition();
console.log(addition(20)); //output: 40 calculated
console.log(addition(20)); //output: 40 cached
```

Q16	What is the use of the promise.all(...) function?

Reference	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all
Ans	The **Promise.all()** method takes an iterable of promises as an input and returns a single Promise that resolves to an array of the results of the input promises. This returned promise will resolve when all of the input's promises have resolved, or if the input iterable contains no promises. It rejects immediately upon any of the input promises rejecting or non-promises throwing an error and will reject with this first rejection message/error.

Q17	let num = 0;
	async function increment() {
	        num += await 2;
	        console.log(num);
	      }
	      increment();
	      num += 1;
	      console.log(num)
	
	What is the resulting output?
	1. 2, 3
	2. 1, 3
	3. 1, 2
	4. 2, 1

Ans	3. 1,2

Q18	What is Swagger Documentation and why is it used?
Reference	https://swagger.io/docs/specification/about/
Ans	Swagger is a set of open-source tools built around the OpenAPI specification that can help you design, build, document and consume REST APIs. The major Swagger tools include:

	• Swagger Editor: A browser-based editor where you can write OpenAPI specs • Swagger UI: Renders OpenAPI specs as interactive API documentation • Swagger Codegen: Generates server stubs and client libraries from an OpenAPI spec

Q19	What are closures?

Ans	A closure is the combination of a function and the lexical environment within which that function was declared, i.e., it is an inner function that has access to the outer or enclosing function’s variables. The closure has the following three scope chains:
	i. Own scope where variables are defined between its curly brackets
	ii. Outer function’s variables
	iii. Global variables
	Let's take an example of a closure concept.
	a) function Welcome(name){
	b) var greetingInfo = function(message){
	c) console.log(message+' '+name);
	d) }
	e) return greetingInfo;
	f) }
	g) var myFunction = Welcome('John');
	h) myFunction('Welcome '); //Output: Welcome John
	myFunction('Hello Mr.'); //output: Hello Mr.John
	
	As per the code, the inner function (greetingInfo) has access to the variables in the outer function scope (Welcome) even after the outer function has returned.

Q20	What is web3.js
Reference	https://web3js.readthedocs.io/en/v1.2.6/#:~:text=web3.-,js%20%2D%20Ethereum%20JavaScript%20API,a%20HTTP%20or%20IPC%20connection.&text=js%2C%20as%20well%20as%20providing%20a%20API%20reference%20documentation%20with%20examples.
Ans	Web3.js is a set of libraries in the JavaScript language which helps in interacting with an Ethereum node using a HTTP or an IPC connection. The web3 JavaScript library interacts with the Ethereum blockchain network. It can perform multiple functions like interacting with smart contracts, sending transactions and retrieving user accounts.