Solidity Basics for JavaScript Devs
by Fullstack Frontend - http://fllstck.dev

K (he/him) for Fullstack Frontend
Posted on Sep 29, 2021 • Updated on Nov 1, 2021

Solidity Basics for JavaScript Devs Part 1

#solidity #javascript #web3

Solidity for JavaScript Devs (3 Part Series)

With all the NFT hype around, it happened that I got tasked to write an article about NFTs and serverless. So, last three weeks, I dived into books, courses, and videos about tokens and smart contracts.

It's an exciting topic, and I think despite the downsides of the token economy, it can be the next step for the internet, being a solution to many problems we currently face.

But in this blog post, I won't go deeper into my opinions about all this and teach you something. I will explain smart contracts written in Solidity with JavaScript equivalents to

clear things up a bit and explain some main differences between these languages. I won't go deep here; I want to explain the basics.

Static vs. Dynamic Typing

The main difference between JavaScript and Solidity is typing. Solidity is statically typed at build time, and JavaScript is dynamically typed.

The reasoning being that the Ethereum Virtual Machine (EVM) is very nitpicky about the costs of calculations and storage. Everything has to be accounted for so you can be charged accordingly.

JavaScript's goal was a bit more ease of use.

JavaScript

```
let x = 10;
```

Solidity

```
int256 x = 10;
```

So, Solidity is a bit like Java or C in that regard.

You also have to type your function arguments and return values.

JavaScript

```
function f(a, b) {
  return a + b;
}
```

Solidity

```
function f(int256 a, int256 b) returns (int256) {
  return a + b;
}
```

If you have more complex types like arrays or structs, the typing system requires you to define the memory location the data will be live.

JavaScript

```
function f(a, b) {
  let c = [];

  for(let i = 0; i < a.length; i++) {
    c[i] += a[i] + b;
  }

  return c;
}
```

Solidity

```
function f(int256[] calldata a, int256 b) returns (int256[] memory) {
  int256[] memory c;

  for(uint i = 0; i < a.length; i++) {
    c[i] = a[i] + b;
  }

  return c;
}
```

Here I defined the first argument `a` as an array of `int256` and said it should be stored in the `calldata` location. `calldata` isn't persistent and can't be modified, and I only read `a` and never write it in the function.

The other variables are either explicitly stored in the `memory` location or have basic types that don't require defining the location.

Integers vs. Numbers

Another fundamental difference between the two languages is their default number type. JavaScript uses `number`, which is always a floating-point number. Solidity uses various sizes of `int`.

The idea behind this is that Solidity, deep down at its core, is about payments, and if you have a currency that is worth thousands of dollars per one whole unit, it could get costly to have rounding errors, which are the norm with JavaScript's `number` type.

It's a bit like working with the dollar and using 1234 cents as storage type instead of 12,34 dollars.

Also, Solidity programmers like the `int256` type as their default type, which can't be mapped 1:1 to JavaScript's `number`. Luckily JavaScipt got a new number type some time ago called `BigInt`, which can store all Solidity numbers with no problem.

JavaScript

```
let x = 9999999999999999;
// will become 10,000,000,000,000,000
// because the number type can't store that big numbers reliably

let y = 9999999999999999n;
// will become 9,999,999,999,999,999
// because the n at the end tells JS that this is a BigInt and not a number
```

Solidity

```
int256 x = 9999999999999999;
```

Contract vs Class

Solidity's contracts are similar to JavaScript classes, but they are different. These contracts are why Solidity applications are called smart contracts.

Solidity is a bit like Java in the regard that a contract is the entry point of a Solidity application. Contracts look like classes in JavaScript, but the difference lies in the instance creation.

When you create an object from a class in JavaScript, that is a relatively straightforward task. You use the `new` keyword with the class name and be done with it.

This can be done with contracts too. Using the `new` keyword on a contract name also leads to a new instance deployed to the blockchain.

JavaScript

```
class MyClass {
```

```
class MyClass {
  #value = 10;
  setValue(x) {
    this.#value = x;
  }
}
```

Solidity

```
contract MyContract {
  int256 private value = 10;
  function setValue(int256 x) external {
    value = x;
  }
}
```

As you can see, `this` is implied in contract methods. So, the attributes of the contract are always in scope in all methods.

The contracts instance, the object, so to say, and its data live on the blockchain and not just inside your Solidity applications memory.

When you deploy a contract to the Ethereum blockchain, you're essentially instancing the contract, and then you can call it from other contracts or a blockchain Client like Ethers.js.

The contract gets an address which you can use later to interact with it. If you deploy the contract multiple times, you have multiple addresses to interact with the different instances.

JavaScript

```
let x = new MyClass();
x.setValue(3);
```

Solidity

```
MyContract x = new MyContract(); // creates a new instance
x.setValue(3);

MyContract x = MyContract(contractAddress); // uses an existing instace
x.setValue();
```

In JavaScript, the objects you create are done if you close the application; in Solidity, the contract instances are persistent on the blockchain.

Interfaces

You need the contract's code to use an already deployed contract, which isn't always available. That's why Solidity also has interfaces, which you can define and use as the type when loading an existing contract.

Solidity

```
interface MyInterface  {
  function setValue(int256 x) external;
}

...

MyInterface x = MyInterface(contractAddress); // uses an existing instace
x.setValue();
```

There are many standardized interfaces for contracts. For example, fungible and non-fungible tokens are standardized, which means we can look in the standard, copy the function signatures we need, and create an interface to call them inside our contracts. Projects like OpenZeppelin also supply us with libraries that already include these well-known interfaces; we don't have to create them ourselves.

NPM for Package Management

Solidity uses the NPM package manager we already know from JavaScript; this way, we can reuse many of the skills we already have.

With the following command, we get a library with all the interfaces that are out in the wild:

```
$ npm i @openzeppelin/contracts
```

Global Variables and `payable`

Some hidden global variables are available in every function. Just like the `window` object in JavaScript, there is a `msg` object in Solidity that contains the data of the caller of the function.

Here is an example in JavaScript that loads data from the global `window` object into a private attribute of a class.

JavaScript

```
class MyClass {
  #title = null;

  constructor() {
    this.#title = window.document.title;
  }
}
```

Same in Solidity, but this time, the contract owner will be set from the global `msg` variable.

Solidity

```
contract MyContract {
  address paybale public owner;

  constructor() payable {
    owner = payable(msg.sender);
  }
}
```

The `msg` variable contains information about the sender of a message. In this case, the address that was used to deploy the contract.

The `constructor` is called automatically when a new instance of a contract is created, just with new objects from classes in JavaScript. Someone had to create the instance, so their blockchain address ended up in the `msg.sender` variable.

In the example, all these functions and variables are defined as `payable`, which means a caller can send Ether to them.

This is pretty awesome because it allows us to use payments for our Solidity application standardized for the whole Ethereum eco-system right in at language level. There isn't an equivalent in JavaScript; we would have to program it on our own.

Summary

Solidity is a straightforward language, and its baked-in payment mechanisms are probably the killer feature that will propel it in the long run.

JavaScript developers should be very familiar with most of the syntax, and the few differences that exist can be learned relatively quickly. The fact that the eco-system also uses NPM makes things even more excellent for JavaScript devs.

This guide isn't exhaustive and talks about a few basics that I saw. I'm by no means a Solidity pro since I only played around with it for three weeks or so.

If you are interested in more content in that direction, let me know!

Also, let me know if I got something wrong :D

Solidity for JavaScript Devs (3 Part Series)

Discussion (9)

Brendan H. Murphy • Oct 3 '21

Fullstack Team,

Thanks for sharing!

Quite honestly, I am new to this community, but have been helping senior engineers get into Web3/Ethereum development as well!

Check it out: optilistic.notion.site/Optilistic-...

Our founding team, is a group of experienced coding school entrepreneurs and smart contract developers, eager to help engineers contribute in Web3 and collaborate in our community through DAO participation.

I would love to connect with you and anyone on this post! I am eager to share interests and see where we can provide value to one another.

Thanks for reading - I hope to connect soon,
Brendan H. Murphy
optilistic.com

CallMeDKay • Oct 8 '21

Great one! The article has followed a pretty understandable style of comparison for JS dev like me :D

K (he/him) • Oct 8 '21

Glad you liked it!

Thorsten Hirsch • Sep 30 '21

Great article! I just want to add: modifiers are also a killer feature of Solidity.

K (he/him) • Sep 30 '21

I like them, but wrapping a function with another function is also possible in JS. :D

Akash • Oct 3 '21

You made it pretty simple to grasp. Would love to read more on topics like Dapps and NFTs from development POV.

K (he/him) • Oct 4 '21

Thanks.

Yes, I'm learning it myself right now and I had the impression there weren't many good sources for that topic. And the stuff that's there gets lost in a sea of investment articles.

Usman Khalil • Sep 30 '21

You're a savior

K (he/him) • Sep 30 '21

I'm glad, I could help!

Code of Conduct • Report abuse

Fullstack Frontend

Frontend is the new fullstack. 🏙

Frontend is the new fullstack. 🏙
Talking about tech that empowers frontend developers! ☁

fllstck.dev

More from Fullstack Frontend

Devconnect: Web2 View
#devconnect #web3 #conference

Devconnect: Chronological View
#devconnect #web3 #conference

Serverless Remix Sessions with Cloudflare Pages
#cloudflare #remix #javascript