**Advanced EVM - Opcodes, calling, etc**

**Advanced EVM - Opcodes, Calling, and Encoding**

You can find all the code we'll be working with in this section in this [**Sublesson Repo**](https://github.com/PatrickAlphaC/hardhat-nft-fcc/tree/main/contracts/sublesson).

Until now, we've been using abi.encode and abi.encodePacked effectively as a means to concatenate strings (which has actually been deprecated in favour of string.concat). In this lesson we'll head back into [**Remix**](https://remix.ethereum.org) to further explore what's actually happening when we invoke these functions.

❗ **NOTE** This section is definitely going to be a little advanced. We're going to go deep into the Ethereum Virtual Machine, op codes and the binary that makes everything work behind the scene.

If you don't understand things right away that's 100% ok. I encourage you to challenge yourself to absorb as much of this as possible before moving on. There's no shame in going through the content a couple times.

**Encoding**

When ready, in [**Remix**](https://remix.ethereum.org), create a new file named Encoding.sol. We can set this contract up with some boilerplate before writing the functions we'll use to explore encoding and decoding.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract Encoding {}

Within this contract, let's now right a simple function to concatenate two strings using abi.encodePacked.

function combineStrings() public pure returns(string memory){

return string(abi.encodePacked("Hi Mom! ", "Miss you!"));

}

Now, if we deploy this and call our combineStrings function, our output is Hi Mom! Miss you!.

What our function is ultimately doing is encoding Hi Mom! and Miss you! into its bytes form and then casting these bytes into a string.

If we just run abi.encodePacked without converting to a string we get:

bytes: 0x4869204d6f6d21204d69737320796f7521

abi.encodePacked exists in Solidity as a [**globally available method**](https://docs.soliditylang.org/en/latest/units-and-global-variables.html#abi-encoding-and-decoding-functions). Solidity actually has a whole bunch of these. You can use the [**Cheatsheet**](https://docs.soliditylang.org/en/latest/cheatsheet.html) made available in the Solidity documentation as a reference.

You'll see lots of things throughout this list that you're already familiar with. Things like msg.sender, msg.value, block.chainid and more. I encourage you to look through the list!

❗ **NOTE** Since Solidity v0.8.12, you no longer need to use abi.encodePacked to concatenate strings. The preferred method is via string.concat(stringA, stringB).

Before we dive deeper into what's happening when we call encodePacked, let's first investigate some of the finer details of sending a transaction.

**Compilation**

When we compile using forge build, a JSON file is added to our out directory. This file contains a lot of data, but our purposes focus primarily on the abi and the bytecode.

Looking at this for the first time can be a little overwhelming, but don't worry we'll break things down a bit.

You can actually get this data right out of Remix. With our Encoding.sol deployed, navigate to the Solidity Compiler tab and click Compilation Details. This will provide a readout which includes the ABI and Bytecode for this contract!

The Bytecode object represents the binary that is actually being put on the blockchain, when we send a transaction.

When our transaction is a new deployment, the to parameter is left empty, but our data parameter is very important. The bytecode we send to the blockchain represents our contract's initialization code and the contract itself.

We can see this in Etherscan for any contract we've deployed. Here's an [**example deployment**](https://sepolia.etherscan.io/tx/0x72d701257bd21f261294e47bb749ff28409aec5e7de64528c3d8d8cb220f4bb4) of BasicNFT.sol. The input data you can see at the bottom (show more details) represents the binary data, the bytecode that we've been talking about. This data includes both the initialization of a contract as well as data representing the contract itself.

***How does the blockchain understand what to do with this bytecode?***

**Op Codes**

Bytecode Example



The above may look like random numbers and letters to us, but to the Ethereum Virtual Machine (EVM), this is effectively the alphabet it uses to perform computation. Every 2 bytes in the data above actually represents an op code. The website [**evm.codes**](https://www.evm.codes/) is an amazing resource for referencing these things.

You could almost use this reference like a dictionary. It tells us any time we see 00 in our bytecode, this represents the STOP operation, for example. In the bytecode example above, the first op code is 60. This pertains to the PUSH1 operation!

This is what is meant by being EVM Compatible, Polygon, Avalanche, Arbitrum etc all compile to the same style of binary, readable by the Ethereum Virtual Machine.

Now, why are we talking about all this? How does it relate to abi encoding?

Until now we've only seen abi.encodePacked used to concatenate strings, but it's capable of much more.

**abi.encode**

Strictly speaking, we can use abi encoding to encode anything we want into the bytecode format understood by the EVM.

Lets write a function to explore this. In our Encoding.sol file, in Remix add:

function encodeNumber() public pure returns(bytes memory){

bytes memory number = abi.encode(1);

return number;

}

❗ **IMPORTANT** ABI stands for application binary interface. What we've largely seen is the human readable version of an ABI.

Go ahead and compile/deploy Encoding.sol with this new function and call it. We should have the encoded version of the number 1 output.

This hex format, this encoding, is how a computer understands the number 1.

Now, as mentioned, this can be used to encode basically anything, we can write a function to encode a string and see what it's output would be just the same.

function encodeString() public pure returns(string memory){

byte memory someString = abi.encode("some string");

return someString;

}

Something you may notice of each of our outputs is how many bytes of the output are comprised of zeros. This padding takes up a lot of space, whether or not it is important to the value being returned.

bytes: 0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000000b736f6d6520737472696e67000000000000000000000000000000000000000000

This is where [**abi.encodePacked**](https://docs.soliditylang.org/en/latest/abi-spec.html#abi-packed-mode) comes in and is available as a non-standard packed mode.

**abi.encodePacked**

abi.encodePacked does much of the same encoding as abi.encode, but comes with some disclaimers.

* types shorter than 32 bytes are concatenated directly, without padding or sign extension
* dynamic types are encoded in-place and without the length.
* array elements are padded, but still encoded in-place

You can kind of think of encodePacked as a compressor which removed unnecessary padding of our binary objects.

We can demonstrate this in Remix by adding this function, redeploying our Encoding.sol contract and calling it.

function encodeStringPacked() public pure returns(bytes memory){

bytes memory someString = abi.encodePacked("some string");

return someString;

}

We can clearly see how much smaller the encodePacked output is, if we were trying to by gas efficient, the advantages of one over the other are obvious.

Encoding in this way is very similar to something else we've done before, typecasting. Add this function to Encoding.sol, and redeploy to see how these compare in practice:

function encodeStringBytes() public pure returns(bytes memory) {

bytes memory someString = bytes("some string");

return someString;

}

So, it looks like abi.encodePacked and bytes casting are doing the same thing here, and for us - functionally they are - but behind the scenes things are a little more complicated. We won't go into the spefics here, but I encourage you to check out the deep dive in [**this forum post**](https://forum.openzeppelin.com/t/difference-between-abi-encodepacked-string-and-bytes-string/11837).

**Decoding**

Concatenating strings is fun and all, but in addition to *encoding* things, we can also *decode*.

From the docs we can see the decode function takes the encoded data and a tuple of types to decode the data into.

function decodeString() public pure returns(string memory) {

string memory someString = abi.decode(encodeString(), (string));

return someString;

}

Once again, we can add this function to our Encoding.sol contract and redeploy in remix to see how it works.

**Muli-Encoding/MultiDecoding**

To take all this one step further, this encoding functionality affords us the ability to encode as much as we want. We can demonstrate this with the following functions:

function multiEncode() public pure returns(bytes memory){

bytes memory someString = abi.encode("some string", "it's bigger!");

return someString;

}

function multiDecode() public pure returns(string memory, string memory){

(string memory someString, string memory someOtherString) = abi.decode(multiEncode(),(string,string));

return (someString, someOtherString)

}

When we multiEncode, you can see that our output is an *even bigger* bytes object, with tonnes of padding. What do you think we can do about it?

You probably guessed, we can **also** multiEncodePacked. Try it out with:

function multiEncodePacked() public pure returns (bytes memory){

bytes memory someString = abi.encodePacked("some string", "it's bigger!");

return someString;

}

This is actually where our fun stops a little bit. Because we're packing the encoding of multiple strings, the decoding function is unable to properly split these up. It's not possible to multiDecode a multiEncodePacked object 😦. If you try something like:

function multiDecodePacked() public pure returns (string memory, string memory){

string memory someString = abi.decode(multiEncodePacked(), (string));

return someString;

}

... this will actually error. We do have an alternative method though.

function multiStringCastPacked() public pure returns (string memory){

string memory someString = string(multiEncodePacked());

return someString;

}

This one actually *will* work.

**Wrap Up**

Don't feel bad if this doesn't click right away, we're broaching some low-level concepts and functions here.

We're making great progress though and should have at least a somewhat better understanding of how the various methods of encoding and decoding are used in the EVM.

In the next lesson we'll apply these learnings and demonstrate how entire function calls can be encoded.