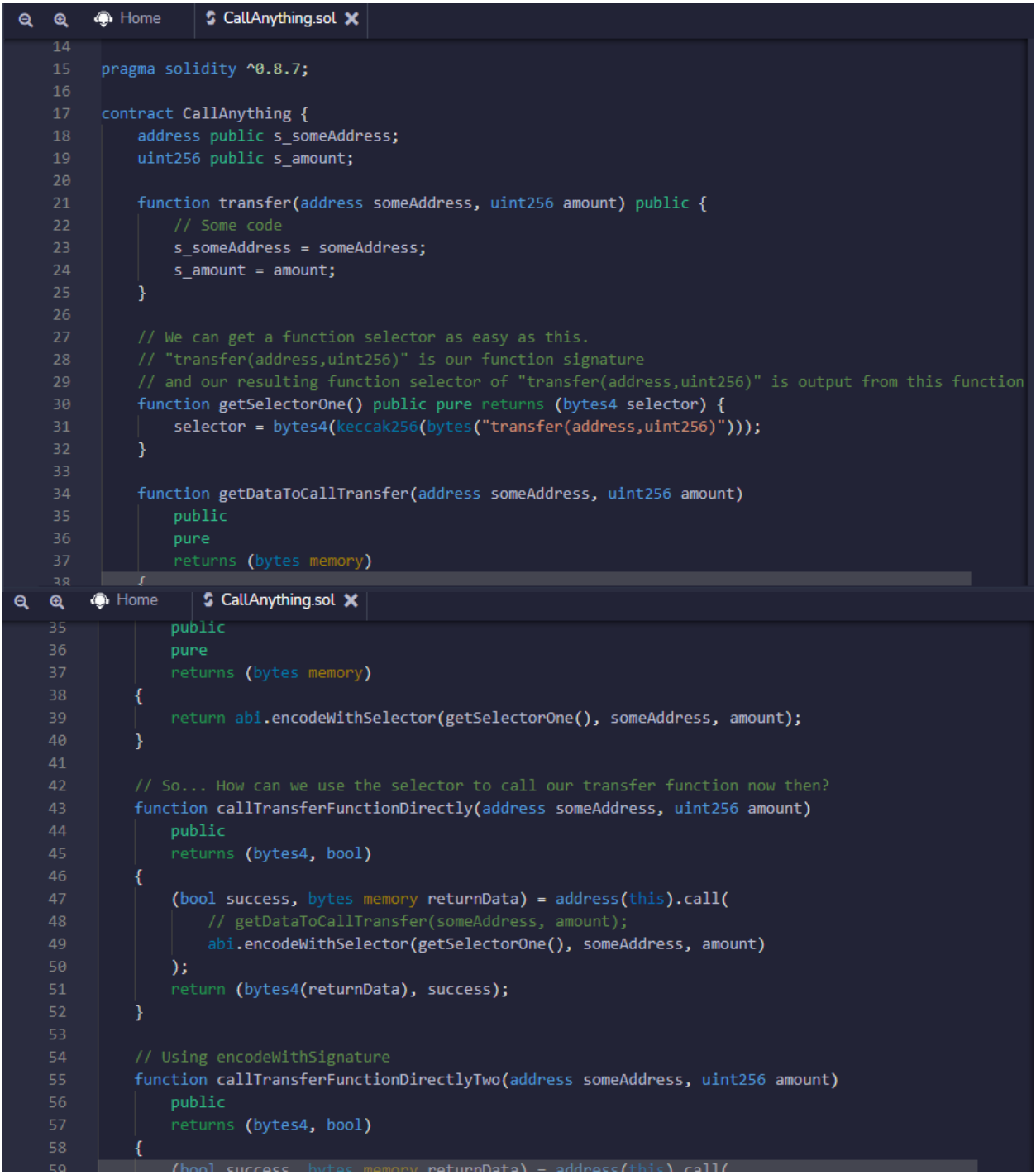
Home
CallAnything.sol
pragma solidity ^0.8.7;
contract CallAnything {
address public s_someAddress;
uint256 public s_amount;
function transfer(address someAddress, uint256 amount) public {
// Some code
s_someAddress = someAddress;
s_amount = amount;
}
// We can get a function selector as easy as this.
// "transfer(address,uint256)" is our function signature
// and our resulting function selector of "transfer(address,uint256)" is output from this function
function getSelectorOne() public pure returns (bytes4 selector) {
selector = bytes4(keccak256(bytes("transfer(address,uint256)")));
}
function getDataToCallTransfer(address someAddress, uint256 amount)
public
pure
returns (bytes memory)
Home
CallAnything.sol
public
pure
returns (bytes memory)
{
return abi.encodeWithSelector(getSelectorOne(), someAddress, amount);
}
// So... How can we use the selector to call our transfer function now then?
function callTransferFunctionDirectly(address someAddress, uint256 amount)
public
returns (bytes4, bool)
{
(bool success, bytes memory returnData) = address(this).call(
// getDataToCallTransfer(someAddress, amount);
abi.encodeWithSelector(getSelectorOne(), someAddress, amount)
);
return (bytes4(returnData), success);
}
// Using encodeWithSignature
function callTransferFunctionDirectlyTwo(address someAddress, uint256 amount)
public
returns (bytes4, bool)
{

Home | CallAnything.sol | BasicNft.sol

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract BasicNft is ERC721 {
    string public constant TOKEN_URI =
        "ipfs://bafybeig37ioir76s7mg5oobetncojcm3c3hxasyd4rvid4jqhy4gkaheg4/?filename=0-PUG.json";
    uint256 private s_tokenCounter;

    constructor() ERC721("Dogie", "DOG") {
        s_tokenCounter = 0;
    }

    function mintNft() public returns (uint256) {
        _safeMint(msg.sender, s_tokenCounter);
        s_tokenCounter = s_tokenCounter + 1;
        return s_tokenCounter;
    }

    function tokenURI(uint256 tokenId) public view override returns (string memory) {
        // require(_exists(tokenId), "ERC721Metadata: URI query for nonexistent token");
        return TOKEN_URI;
    }
```

Home | CallAnything.sol | BasicNft.sol | DynamicSvgNft.sol

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";
import "base64-sol/base64.sol";
import "hardhat/console.sol";

contract DynamicSvgNft is ERC721, Ownable {
    uint256 private s_tokenCounter;
    string private s_lowImageURI;
    string private s_highImageURI;

    mapping(uint256 => int256) private s_tokenIdToHighValues;
    AggregatorV3Interface internal immutable i_priceFeed;
    event CreatedNFT(uint256 indexed tokenId, int256 highValue);

    constructor(
        address priceFeedAddress,
        string memory lowSvg,
        string memory highSvg
    ) ERC721("Dynamic SVG NFT", "DSN") {
        s_tokenCounter = 0;
```

Home | CallAnything.sol | BasicNft.sol | DynamicSvgNft.sol | Encoding.sol

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

// For the cheatsheet, check out the docs: https://docs.soliditylang.org/en/v0.8.13/cheatsheet.html?hi

contract Encoding {
    function combineStrings() public pure returns (string memory) {
        return string(abi.encodePacked("Hi Mom! ", "Miss you."));
    }

    // When we send a transaction, it is "compiled" down to bytecode and sent in a "data" object of th
    // That data object now governs how future transactions will interact with it.
    // For example: https://rinkeby.etherscan.io/tx/0x924f592458b0e37ee17024f9c826b97697455cd97f6946b8

    // Now, in order to read and understand these bytes, you need a special reader.
    // This is supposed to be a new contract? How can you tell?
    // Let's compile this contract in hardhat or remix, and you'll see the the "bytecode" output - tha
    // creating a contract.

    // This bytecode represents exactly the low level computer instructions to make our contract happe
    // These low level instructions are spread out into soemthing call opcodes.

    // An opcode is going to be 2 characters that represents some special instruction, and also option
```

Home | CallAnything.sol | BasicNft.sol | DynamicSvgNft.sol | Encoding.sol

```sol
    // encodePacked
    // This is great if you want to save space, not good for calling functions.
    // You can sort of think of it as a compressor for the massive bytes object above.
    function encodeStringPacked() public pure returns (bytes memory) {
        bytes memory someString = abi.encodePacked("some string");
        return someString;
    }

    // This is just type casting to string
    // It's slightly different from below, and they have different gas costs
    function encodeStringBytes() public pure returns (bytes memory) {
        bytes memory someString = bytes("some string");
        return someString;
    }

    function decodeString() public pure returns (string memory) {
        string memory someString = abi.decode(encodeString(), (string));
        return someString;
    }

    function multiEncode() public pure returns (bytes memory) {
        bytes memory someString = abi.encode("some string", "it's bigger!");
        return someString;
    }
```