Contracts and Basic Functions

Hello World

Copy page

Write your first contract with Solidity.

As is tradition, we'll begin coding with a variant of "Hello World" written as a smart contract. There isn't really a console to write to*, so instead, we'll write a contract that says hello to the sender, using the name they provide.

*You will be able to use `console.log` with *Hardhat*, with some restrictions.

Objectives

By the end of this lesson you should be able to:

- Construct a simple "Hello World" contract
- List the major differences between data types in Solidity as compared to other languages

Select the appropriate visibility for a function

Hello World

Writing “Hello World” in a smart contract requires a little more consideration than in other languages. Your code is deployed remotely, but it isn’t running on a server where you can access logs, or on your local machine where you have access to a console. One way to do it is to write a function that returns “Hello World”.

Creating the Contract

To create a contract:

1. Create a new workspace in Remix.
2. Name it `Hello World` and delete the `.deps` folder.
3. Leave `.prettierrc.json` and click the settings gear in the bottom left.
4. Uncheck the top option to *Generate contract metadata...*
5. Open the *Solidity Compiler* plugin and enable *Auto compile*.
6. Create a new folder called `contracts`, and within that folder, create a file called `hello-world.sol`.

Solidity files usually start with a comment containing an ***SPDX-License-Identifier***. It's not a requirement, but there are a couple of advantages to doing this. First, everything you deploy on the blockchain is public. This doesn't mean you are giving away everything you deploy for free, nor does it mean you have the right to use the code from any deployed contract. The license determines allowed usage and is generally protected by international copyright laws, the same as any other code.

If you don't want to give a license, you can put `UNLICENSED`. Common open source licenses, such as `MIT` and `GPL-3.0` are popular as well. Add your license identifier:

```
// SPDX-License-Identifier: MIT
```

Below the license identifier, you need to specify which **version** of Solidity the compiler should use to compile your code. If by the time you read this, the version has advanced, you should try to use the most current version. Doing so may cause you to run into unexpected errors, but it's great practice for working in real-world conditions!

```
pragma solidity 0.8.17;
```

Finally, add a `contract` called `HelloWorld`. You should end up with:

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.17;

contract HelloWorld {

}
```

SayHello Function

Add a function to your contract called `SayHello` :

```
function SayHello() {

}
```

You'll get a compiler syntax error for *No visibility specified. Did you intend to add "public"?*.

Is `public` the most appropriate **visibility specifier**?

It would work, but you won't be calling this function from within the contract, so `external` is more appropriate.

You also need to specify a return type, and we've decided this function should return a string. You'll learn more about this later, but in Solidity, many of the more complex types require you to specify if they are `storage` or `memory`. You can then have your function return a string of `"Hello World!"`.

Don't forget your semicolon. They're mandatory in Solidity!

You should have:

```
function SayHello() external returns (string memory) {
    return "Hello World!";
}
```

Before you deploy, check the `Compiler` plugin. You've got one last warning:

> Warning: Function state mutability can be restricted to pure

Modifiers are used to modify the behavior of a function. The `pure` modifier prevents the function from modifying, or even accessing state. While not mandatory, using these modifiers can help you and other programmers know the intention and impact of the functions you write. Your final function should be similar to:

```
function SayHello() external pure returns (string memory) {
    return "Hello World!";
}
```

Deployment and Testing

Confirm that there is a green checkmark on the icon for the compiler plugin, and then switch to the *Deploy & Run Transactions* plugin.

Click the *Deploy* button and your contract should appear in *Deployed Contracts*. Open it up and then click the *SayHello* button. Did it work?

You should see your message below the button. Another option to see the return for your `HelloWorld` function is to expand the entry in the console. You should see a *decoded output* of:

```
{
    "0": "string: Hello World!"
}
```

Greeter

Now, let's modify your say hello function to greet a user by name, instead of just saying "Hello World!"

First Pass Attempt

You'd probably expect this to be pretty easy. Start by changing the name of the function (or adding a new one) to `Greeter` and giving it a parameter for a `string memory _name`. The underscore is a common convention to mark functions and variables as internal to their scope. Doing so helps you tell the difference between a storage variable, and a memory variable that only exists within the call.

Finally, try creating a return string similar to how you might in another language with `"Hello " + _name`. You should have:

```
// Bad code example: Does not work
function Greeter(string memory _name) external pure returns (string memory)
    return "Hello " + _name + "!";
}
```

Unfortunately, this does not work in Solidity. The error message you receive is a little confusing:

> TypeError: Operator + not compatible with types literal_string "Hello" and string memory.

You might think that there is some sort of type casting or conversion error that could be solved by explicitly casting the string literal to string memory, or vice versa. This is a great instinct. Solidity is a very explicit language.

However, you receive a similar error with `"Hello " + "world"`.

String concatenation is possible in Solidity, but it's a bit more complicated than most languages, for good reason. Working with string costs a large amount of gas, so it's usually better to handle this sort of processing on the front end.

Plan B

We still want to return something with the name provided by the user, so let's try something a little different. Solidity is a *variadic* language, which means it allows functions to return more than one value.

Modify your return declaration: `returns (string memory, string memory)`

Now, your function can return a **tuple** containing two strings!

```
return ("Hello", _name) ;
```

Deploy and test your contract. You should get a *decoded output* with:

```
{
    "string _name": "Your Name"
}
```

Full Example Code

```
contract HelloWorld {

    function SayHello() external pure returns (string memory) {
        return "Hello World!";
    }

    // Bad code example: Does not work
    // function Greeter(string memory _name) external pure returns (string
    //     return "Hello " + _name;
    // }

    function Greeter(string memory _name) external pure returns (string mem
        return ("Hello", _name);
    }
}
```

Conclusion

Congratulations! You' act! You selected a license and a version ded a function that

returns a value.

You also learned more about some of the ways in which Solidity is more challenging to work with than other languages, and the additional elements you sometimes need to declare functions and types.

Introduction to Contracts

Basic Types