## **Basic Solidity Data Types**

In this tutorial we'll learn basic Solidity data types!

We'll use state variables to store them in the contract's persistent data storage:

Explain

**contract** Contract {

**bool** **public** value = **true**;

**int** **public** a = 10;

**string** **public** msg = "Hello World";}

We're going to have to look at each of these data types in closer detail.

Since Smart Contracts deal primarily with digital value, it's important to understand everything that is happening on a low level. Any mistake could have dire consequences! It's also important to do things ****efficiently**** since all storage and computation on the blockchain will cost money.

With this in mind, let's dive into Basic Solidity Data Types!

## **Accounts**

In Ethereum, accounts are often distinguished into two types: ****Externally Owned Accounts**** and ****Contract Accounts****. The differences between these accounts is largely conceptual as the EVM essentially treats them the same!

Every account on the EVM has a public address and a balance. Contract accounts will also store their bytecode as well as their internal storage data.

When making a call from an EOA to a Contract Account it's important to know things like who is making the call, how much ether they are sending and the function they are intending to invoke with which arguments.

The Solidity language handles the wiring up of the transaction data to the function we have defined on the contract. The language also gives us access to the transaction parameters through globals like msg.sender and msg.value.

By providing these utilities for working with accounts we can easily define roles, permissions and track token balances in contracts. Let's learn all about working with accounts in Solidity!

## **Learning Revert**

We've been building up to this lesson for quite a few coding lessons now! It's time to learn the basics of reverting transactions in Solidity!

In this lesson we'll discuss how to detect an error condition and ****immediately halt**** the smart contract code, stopping any further execution of the transaction that the code is running in, and refunding any remaining gas to the user. The EVM gives us a few ways to ****stop a transaction and roll-back any state changes and emitted events**** with the REVERT EVM opcode.

In Solidity version ^0.8.0 there are 3 ways to express errors in Solidity. They are:

* assert
* require
* revert

Ok, enough talk. Time to dig in!

## **Calldata**

When we want to communicate with a smart contract, we send a transaction from an Externally Owned Account. Inside of that transaction is a data property which is commonly referred to as the "calldata". This call data format is the same for calling solidity functions whether it is in a transaction from an EOA or if its in a message call from one contract to another.

The format looks a little like this. Let's say you wanted to call a method approve on a contract, that takes a uint:

**function** approve(**uint** val) **external**;

 We can target this function by taking its signature and hashing it with keccak256, then taking the first 4 bytes. So for approve here, it would be the keccak256("approve(uint256)"). The first 4 bytes of the resulting hash is 0xb759f954. This is the first part of our calldata!

Then we need to decide how much we want to approve. What is our val? Let's say it was 15, that would 0xf in hexadecimal. We will need to pad this value out to 256 bits, or 64 hexadecimal characters. The resulting value will be:

000000000000000000000000000000000000000000000000000000000000000f

If we combine this with the function signature, our call data would look like this:

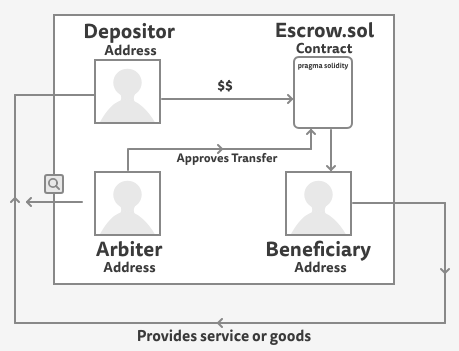
0xb759f954000000000000000000000000000000000000000000000000000000000000000f

Regardless of whether its in a transaction from an EOA or a message call from one contract to another, this would be our calldata sending 15 to an approve function.

Ready to start working with calldata? Let's jump in!

## **What's an Escrow?**

An escrow is an agreement often used when transferring funds in exchange for a good or service. Funds can be held in escrow and a third party can be chosen to "arbitrate" or approve the transfer when the service or good is provided.



There are many use cases for Escrows across real estate, charities and marketplaces. It's the bread and butter of Ethereum Smart Contracts as it's quite easy to write, and yet, so powerful.

## **Solidity Arrays**

Working with arrays in Solidity can be ****tricky****!

We'll need to understand the ****data location****, or where the array is stored.

We'll also need to understand the ****memory allocation****, whether it is fixed size or it can be dynamically resized.

Don't you worry! We'll bring these concepts to ****light**** in this coding tutorial.

## **Structs**

Structs provide Solidity Developers with a way to build ****custom data types****. These data types can have multiple fields of data types we already have previously discussed.

For example:

Explain

**struct** Account {

**uint** balance;

**bool** isActive;}

 This struct is composed of a uint and a bool. If this struct were defined outside of a contract, it can be shared across multiple contracts like so:

Explain

**struct** Account {

**uint** balance

**bool** isActive;}

**contract** A {

Account owner;

Account recipient;}

**contract** B {

**mapping** (**address** => Account) accounts;}

## **Mappings**

Mappings are an important data type in Solidity. With a mapping we can take values of one data type and map them to values of another data type.

Let's consider an example. Say you wanted to track balances of a bunch of addresses:

|  |  |
| --- | --- |
| **address** | **balance** |
| 0xc783df8a850f42e7F7e57013759C285caa701eB6 | 500 |
| 0xeAD9C93b79Ae7C1591b1FB5323BD777E86e150d4 | 100 |
| 0xE5904695748fe4A84b40b3fc79De2277660BD1D3 | 20 |

Based on what we have learned so far you might be thinking of creating an array of structs:

Explain

**struct** Account {

**address** addr;

**uint** balance;}

Account[] accounts;

 This works, certainly. When we need to find an address balance we can iterate through the accounts to find the address we're looking for and retrieve the balance.

****However****, there's a better way! We can use a mapping:

**mapping**(**address** => **uint**) balances;

 With a mapping each address is mapped to a unique place in memory where it stores the uint balance. Looking up the balance is as simple as providing the address to the mapping: balances[addr].

 Under the hood, the storage location is found by hashing the balances variable location with the address passed in. It's a simple lookup! Much more efficient than iteration. Generally it's better to try to use a mapping ****first**** and only use an array when you realize you need to iterate.

## **Voting Contract**

In this tutorial we're going to build a voting contract! We'll use the lessons learned here to understand how the Governor standard emerged

## **Proposals**

We're going to focus on creating a voting contract that will allow members to create new proposals. This contract can contain many proposals which will be voted on by a group of members. Each proposal will keep track of its votes to decide when its time to execute.

At execution time, these proposals will send calldata to a smart contract. The calldata could be anything! We could have a Voting system that allows 100 members to decide when to upgrade a protocol. The calldata might target a function with the signature upgrade(address) and send over the new protocol implementation. That would be a very cool use of your Voting contract!

## **Contract Inheritance**

As with many object-oriented languages, Solidity includes ****inheritance**** as a feature.

Inheritance means that you can create an object with some values/methods and use it as a ****base**** for other objects.

In Solidity, the objects we're referring to are contracts and interfaces. We can write a contract with state variables and functions. Then we can create contracts that ****inherit**** those variables and functions. These ****derived**** contracts can then choose to add behavior as necessary.

This use case may seem quite similar to libraries! Just like libraries, code re-use is a big motivation for inheritance. We'll take a look at why inheritance can be a more powerful feature than libraries. Of course, that power comes with tradeoffs! With time and practice you'll be able to understand these tradeoffs and figure out which tool is right for the particular job at hand.