**Voting Smart Contract**

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.

1. **Proposal**

**Proposal Storage**

In this stage we're going to focus on the storage of new proposals.

A proposal should take in some calldata, and a target. When the proposal passes the voting stage, the contract will send that calldata to the target. To begin with, the yesCount and noCount should be zero until we record some votes!

1. **Voting**

Voting Now that we have proposals with vote counts, it's time to create voting functionality!  
**3. Multiple Votes**

**Multiple Votes**

We need to handle the case where an address votes twice.

One potential way to handle this is to prevent voters from voting twice. However, what if they want to change their vote?

Let's allow voters to change their vote. Adding this functionality will require that we account for the vote change in the vote counts. This means that if the address previously voted yes and switched to no, we'll want to decrement the yesCount and increment the noCount. And vice-versa!

**4 .Events**

**Events**

We'll want to make it easy for the user interface to subscribe to new proposal and voting events! Let's add some new events so we can listen closely.

**5 .Members**

**Voting Members**

It's important for us to maintain a list of voting members.

After all, it's relatively easy for anyone to make hundreds of Ethereum addresses very quickly and vote with each of these addresses. The only thing stopping them is gas and effort!

When a single person operates many accounts it is known as a Sybil attack. Any system that is setup to handle this is known to be sybil resistant.

1. **Execute**

**Execute Vote**

In smart contract governance systems there is usually some minimum voting participation that must be reached before a proposal can be executed. Most governance systems today use coin voting, where the number of voting ERC20 tokens you hold decide your vote weight.

**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.

1. **Inheritance**

**Inheritance**

Traditionally, inheritance is when one class copies (or inherits) functionality from another class.

For Solidity, we'll be talking about inheritance for Contracts instead of Classes. However, the concept is the same!

It's quite easy to inherit contracts in Solidity. Let's see an example:

**contract Base {**

**uint public value = 10;**

**function changeValue(uint \_value)**

**external {**

**value = \_value;**

**}**

**}**

**contract Derived is Base {**

**// inherits everything from base**

**contract!**

**}**

In this example the Derived inherits both the value storage variable as well as the changeValue function!

To setup the inheritance, all we need is the is keyword in the contract declaration, which specifies which contract to inherit from. The body of the changeValue function is actually copied into the Derived functions bytecode, when inherited. Then we can deploy Derived as a standalone contract with all of its inherited functionality baked in.

1. **Constructor Args**

Constructor Inheritance In the previous example, the Base contract had a state variable with an initial value and a function to modify it. What if the Base contract had an initial value that was set in a constructor?

**contract Base {**

**uint public value;**

**constructor(uint \_value) {**

**value = \_value;**

**}**

**}**

Can we pass that initial value into the Base constructor when we're inheriting it?

Sure can!

Same syntax as invoking a function:

**contract Derived is Base(10) {**

**// inherits everything from base**

**contract!**

**}**

See how we passed 10 into the Base contract? This is provided as a constructor argument!

It is possible to send multiple arguments to the constructor as well. This is also function syntax, comma-separated arguments within the parenthesis.

1. **Vertual Override**

**Virtual & Override**

It's time to introduce two new function keywords: virtual and override.

Sometimes we'll want to leave a function on a base contract open to re-implementation by its derived class. That's where these two new keywords come in. The virtual keyword allows us to specify a function on a contract that can be overridden using the override keyword. **contract Base {**

**uint public value = 5;**

**// this method can be overridden**

**function increaseValue() virtual**

**external {**

**value += 10;**

**}**

**}**

**contract Derived is Base {**

**// this method overrides the**

**virtual method**

**function increaseValue() override**

**external {**

**value \*= 2;**

**}**

**}**

In this case, both Derived and Base have different function bodies for increaseValue.

The Derived contract will use its own implementation of increaseValue, which overrides the virtual function implemented in Base.

The overriding function must have the same visibility as the virtual function. If not the compiler will throw a TypeError: "Overriding function visibility differs". Keep an eye out for that one!

We can also specify abstract contracts where virtual functions do not require an implementation. However, these functions must be implemented at some point by a derived contract.

1. **Super**

Super Call

In the last stage we wrote an override function for an unimplemented function on the base contract.

In other cases, the base contract will have functionality in its virtual functions that we want to share with our derived contracts. That's when it's time to use super.

**contract Base {**

**uint public value = 10;**

**function modify() virtual external**

**{**

**value \*= 2;**

**}**

**}**

**contract Derived is Base {**

**function modify() virtual override**

**external {**

**value += 20;**

**super.modify(); // results in**

**value = 60**

**// Base.modify() would also**

**work!**

**}**

**}**

You can see in our Derived contract we are modifying the value and then calling super.modify to invoke the function on the base contract as well. This will first perform value += 20 from the override function, then perform value \*= 2 from the super function, resulting in value=60. With arguments this works like any other function: we would pass through arguments to super.modify().

1. **Ownable**

**Base Utility Contracts**

It is often quite useful for a base contract to provide utility functions and modifiers.

Let's see an example:

**contract Depositable {**

**modifier requiresDeposit {**

**require(msg.value >= 1 ether);**

**\_;**

**}**

**}**

**contract Escrow is Depositable {**

**address owner;**

**constructor() requiresDeposit {**

**owner = msg.sender;**

**}**

**}**

Here the Escrow contract requires a deposit of at least 1 ether in order to deploy. Otherwise the transaction will revert. This requirement comes from the base contract Depositable and is used through the inherited requiresDeposit modifier.

1. **Multiple Inheritance**

**Multiple Inheritance**

It's possible to inherit from multiple contracts.

The derived contract will inherit state variables and functions from each base contract:

**contract Base1 {**

**uint a = 5;**

**}**

**contract Base2 {**

**uint b = 10;**

**}**

**contract Derived is Base1, Base2 {**

**// has access to both b and a!**

**}**

You can see we specify the contracts to inherit from in a comma-separated list. The Derived contract is inheriting from both Base1 and Base2. When it comes to multiple inheritance, order matters! Let's take a closer look in details.