**Reverting Transaction**

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

1. **Constracter Revert**

**Reverting Transactions**

In the EVM the main opcode to revert a transaction is REVERT. There are three ways to invoke the REVERT opcode from Solidity are assert, require and revert. We'll focus on the last two for now (see this stage's details section for more on assert). We can revert a transaction in Solidity by using the require function and the revert statement. A require statement has two forms:

**require(someBooleanCondition);**

**require(someBooleanCondition,**

**"Optional error message");**

These will revert if someBooleanCondition is false. We can use these to check for all kinds of conditions. The revert keyword also has two forms:

**// old syntax to revert with a string**

**revert("Some error message");**

**// new syntax to revert with a custom error**

**revert SomeCustomError(arg1, arg2, ...);**

Notice that, either way, revert does not take a boolean condition. Revert will always revert, so you will usually see it wrapped in a conditional statement:

**if(someCondition) {**

**revert SomeCustomError(arg1,**

**arg2, ...);**

**}**

Both revert and require use the REVERT opcode, they just provide different syntaxes to do so. The recommended approach is to use custom errors in most cases as they provide a gas-efficient way to identify and debug issues. A custom error, like a function, is identified by the first 4 bytes of the keccak256 hash of its canonical representation, which includes its name and parameter types. In contrast, using string literals for error messages can consume many more bytes, as we discussed in our lesson on string literals.

1. **Only Owner**

**Restricting by Address**

We can provide certain roles to an address. For instance, let's say we had an address with the privilege of creating new game items:

**contract Game {**

**address itemCreator =**

**0xc783df8a850f42e7F7e57013759C285caa701eB6;**

**error NotItemCreator(address);**

**function createItem() external {**

**if(msg.sender != itemCreator) {**

**revert**

**NotItemCreator(msg.sender);**

**}**

**// TODO: create the item!**

**}**

**}**

This function createItem may be public, but there's only one address that can call it without the transaction reverting!

1. **Owner Modifier**

Function Modifiers

We can write modifiers on functions to run logic before and/or after the function body. Let's see an example:

**import "forge-std/console.sol";**

**contract Example {**

**function logMessage() public view**

**logModifier {**

**console.log("during");**

**}**

**modifier logModifier {**

**console.log("before");**

**\_;**

**console.log("after");**

**}**

**}**

Let's say we called logMessage, what would you expect the order of the logged messages to be? It would be: before, during or after Why? Notice that the logMessage function signature is decorated with the logModifier modifier. This modifier can add behavior to the function before and after the function body runs. The \_ in the modifier body is where the function body of the modified function will run.

**Calling Contracts**

**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.

1. **Call Function  
   Interfaces**

The easiest way to enable one contract to interact with another is by defining the target contract. Interfaces serve this purpose; for example:

**interface IToken {**

**function getBalance(address user)**

**external;**

**}**

We can use this interface to properly communicate with a token contract that implements the getBalance method:

**// tokenAddress: a contract adddress we**

**want to communicate with**

**// userAddress: the address we want to**

**lookup the balance for**

**uint balance =**

**IToken(tokenAddress).getBalance(**

**userAddress);**

Behind the scenes Solidity is creating a message call that encodes the calldata for this getBalance call.

1. **Signature**

**Function Signature**

The first step to forming calldata manually is taking the keccak256 hash of the function signature you are targeting. So, for example, if we we are trying to call rumble:

**function rumble() external;**

We need to take the keccak256 hash of rumble() and grab the first 4 bytes. As it turns out, those 4 bytes are 0x7e47cd7e. This would be our entire calldata to make the function call to rumble on a contract!

1. **With Signature**

**Encode With Signature**

As a bit of a shortcut to the previous stage, we can use the method abi.encodeWithSignature! This method will do everything we did in the last stage, in one function.

**// replace this:**

**bytes4 memory payload = abi.encodePacked(bytes4(keccak256("rum**

**ble()")));**

**// with this:**

**bytes memory payload =**

**abi.encodeWithSignature("rumble()");**

And if you want to add arguments, you can add them to signature and as comma separated arguments to the encodeWithSignature method. If rumble took two uint arguments, we could pass them like this:

**bytes memory payload =**

**abi.encodeWithSignature("rumble(uint256,uint25**

1. **", 10, 5);**

**(bool success, ) = hero.call(payload);**

1. **Arbitrary Alert**Taking Calldata If we take calldata as an argument to a function we can pass that arbitrary calldata along to another contract. This can be super useful, especially in contracts that require many people to pass their approval before a transaction is executed. We'll talk about governance proposals and multiple-signature wallets later on in the course and you'll see that storing calldata for later use is critical for maximum flexibility in these cases.
2. **Fallback**

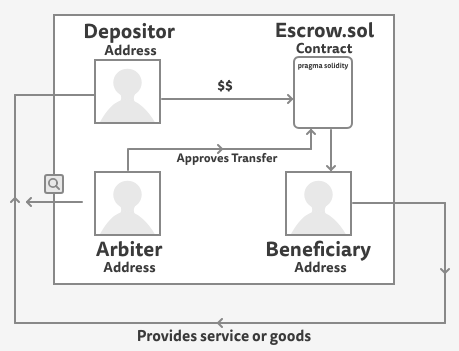
When data is sent to a contract and it doesn't match any of the contract's function 'identifiers' (the first 4 bytes of the hash of the function's signature), the contract's fallback function will be triggered. This means that if you send a random 4-byte value to a contract, it will most likely not match any function and will invoke the fallback function if one exists.

The same holds true if you send less or more than 4 bytes! As long as those initial 4 bytes don't match a function identifier, the fallback function gets triggered

**Escrow Smart Contract**

## **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.

## **Let's Get Building**

We hope that during this block you'll be thinking about ways that you can use an escrow to launch new decentralized applications. You'll be able to use all the code you wrote in a working web application!

1. **Setup**

**State Variables**

We'll have three parties involved in the Escrow:

1. **Depositor -** The payer of the Escrow, makes the initial deposit that will eventually go to the beneficiary.
2. **Beneficiary** - The receiver of the funds. They will provide some service or good to the depositor before the funds are transferred by the arbiter.
3. **Arbiter** - The approver of the transaction. They alone can move the funds when the goods/services have been provided. For this first stage, let's create these addresses as public storage variables!
4. **Constructor**

**Constructor Storage**

Each time that a depositor, arbiter and beneficiary come to an agreement upon Escrow terms, they can deploy a contract. The depositor will be the deployer of the contract. They will ask the arbiter and beneficiary for addresses that those two parties have access to. Then the depositor will provide those addresses as the arguments to the Escrow contract for storage.

1. **Funding**

It's time to fund the contract! The depositor will send some ether to the contract, which will be used to pay the beneficiary after the transfer is approved by the arbiter.

1. **Approval**

After the contract has been deployed with the appropriate amount of funds, the beneficiary will provide the good or service. They are now secure in knowing that the money is on its way!

Once the good or service is provided, the arbiter needs a way to approve the transfer of the deposit over to the beneficiary's account.

1. **Security**

**Lock it Down**

There's only one address that should be able to call the approve method: the arbiter.

This is their role in the escrow transaction, to decide when the funds can be transferred.  
**6. Events**

It's important for servers and front-ends to be able to listen and index on important blockchain events so that they can show the most up-to-date information to users! An event can be declared and emitted like this:

contract X {

event MyEvent(address);

function importantThing() external {

emit MyEvent(msg.sender);

}

}

Let's create an event so it is easy for an application to subscribe to when the Escrow is approved!