BCDV 1010
Smart Contract Development Essentials

2023 January

week 01 - class 05

Error Handling

Solidity uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller.

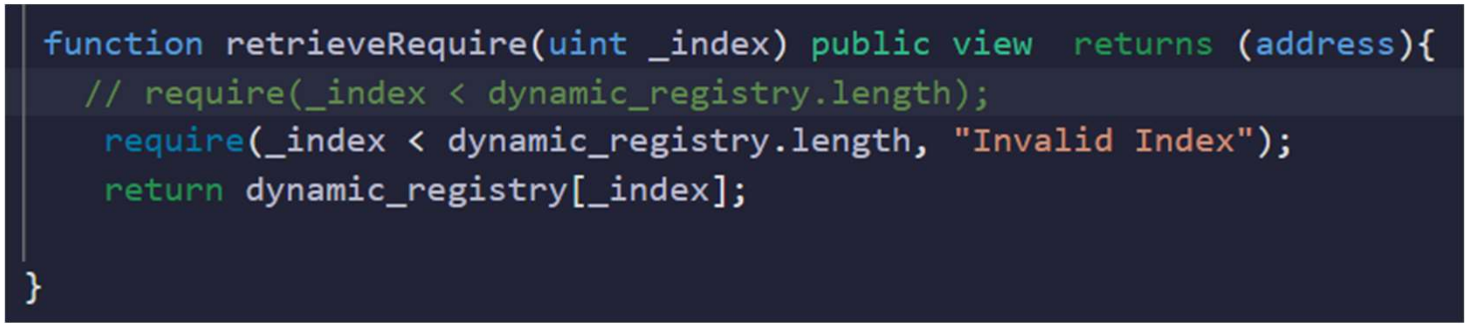
Once the error is caught, the transaction fails.

The errors in the solidity can be handled in the following ways:

1. require
2. revert
3. Assert
4. try/catch

1. Require

require can be used to check for conditions and throw an exception if the condition is not met. It either creates an error without any data or an error of type Error(string). It should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts.

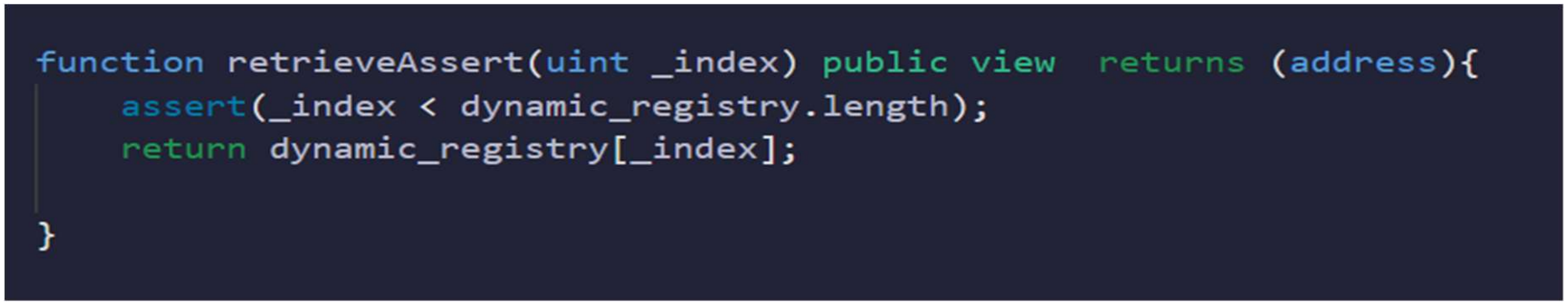


```
function retrieveRequire(uint _index) public view  returns (address){
  // require(_index < dynamic_registry.length);
   require(_index < dynamic_registry.length, "Invalid Index");
   return dynamic_registry[_index];

}
```

2. Assert

The `assert` function creates an error of type `Panic(uint256)`. The same error is created by the compiler in certain situations as listed below.

Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract that you should fix.



```
function retrieveAssert(uint _index) public view  returns (address){
    assert(_index < dynamic_registry.length);
    return dynamic_registry[_index];

}
```

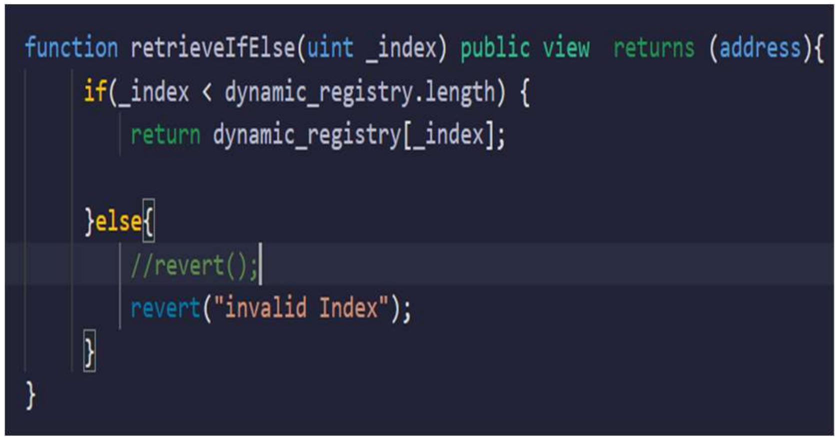
3. revert

A direct revert can be triggered using the revert statement and the revert function. The revert statement takes a custom error as a direct argument without parentheses:

revert CustomError(arg1, arg2);

For backwards-compatibility reasons, there is also the revert() function, which uses parentheses and accepts a string:

revert(); revert(“description”);



```
function retrieveIfElse(uint _index) public view  returns (address){
    if(_index < dynamic_registry.length) {
        return dynamic_registry[_index];

    }else{
        //revert();
        revert("invalid Index");
    }
}
```

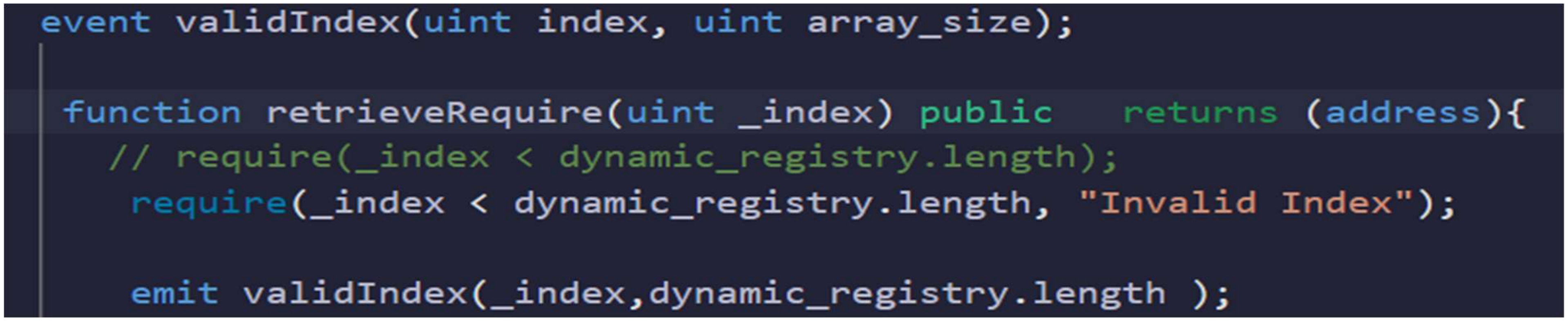
The error data will be passed back to the caller and can be caught there. Using revert() causes a revert without any error data while revert("description") will create an Error(string) error.

Note: The revert is always used along with the if-else conditions.

Events

Events in solidity are used to create the logs in the Ethereum ledger. We can pass any type of data as arguments to it. The event can be triggered by using the keyword ‘emit’.

Events are costly, so it’s making sure to use it only where needed in order to keep the transaction costs low.

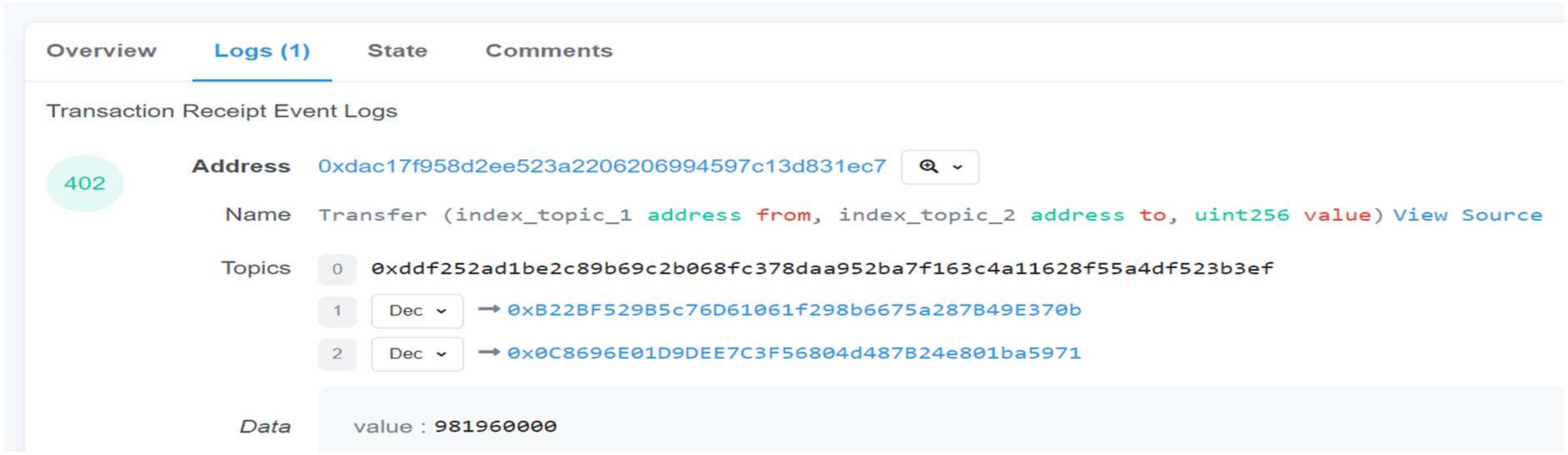


```
event validIndex(uint index, uint array_size);

 function retrieveRequire(uint _index) public   returns (address){
   // require(_index < dynamic_registry.length);
    require(_index < dynamic_registry.length, "Invalid Index");

    emit validIndex(_index,dynamic_registry.length );
```

E.g:

You can also see the contract events logged on Etherscan.

https://etherscan.io/tx/0x9ba98d07fcda82cfbc30336bcb2f67899e2bea7577e0e75a0c45a18e58c8c554#eventlog



Transaction Receipt Event Logs

402

Address 0xdac17f958d2ee523a2206206994597c13d831ec7

Name `Transfer (index_topic_1 address from, index_topic_2 address to, uint256 value)` View Source

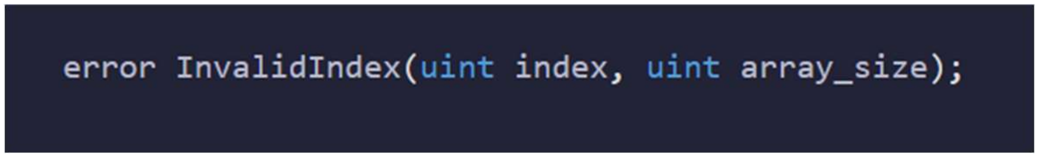
Topics
```
0        0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
1  Dec → 0xB22BF529B5c76D61061f298b6675a287B49E370b
2  Dec → 0x0C8696E01D9DEE7C3F56804d487B24e801ba5971
```

Data value : 981960000

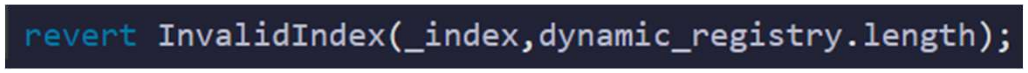
Errors

In case of a failure inside a contract, the contract can use a special opcode to abort execution and revert all state changes. In addition to these effects, descriptive data can be returned to the caller. This descriptive data is the encoding of an error and its arguments in the same way as data for a function call.

e.g:



```
error InvalidIndex(uint index, uint array_size);
```

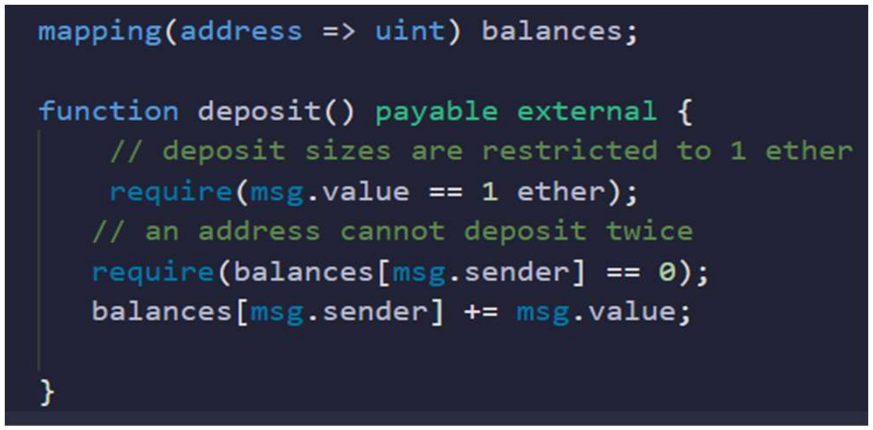


```
revert InvalidIndex(_index,dynamic_registry.length);
```

Payable

The keyword *payable* allows someone to send ether to a contract and run code to account for this deposit. This code could potentially log an event, modify storage to record the deposit, or it could even revert the transaction if it chooses to do so.

When a developer explicitly marks a [smart contract with the payable type](), they are saying “I expect ether to be sent to this function”.

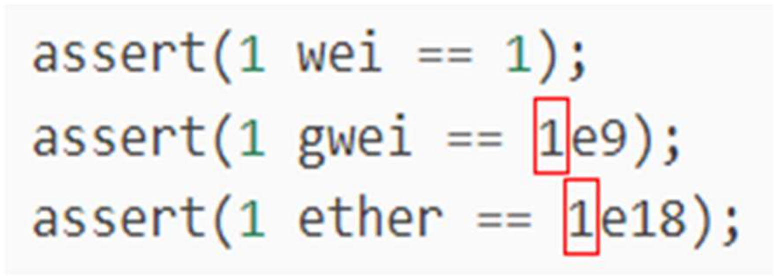


```
mapping(address => uint) balances;

function deposit() payable external {
    // deposit sizes are restricted to 1 ether
    require(msg.value == 1 ether);
   // an address cannot deposit twice
   require(balances[msg.sender] == 0);
   balances[msg.sender] += msg.value;

}
```

Units and Globally Available Variables

Ether Units

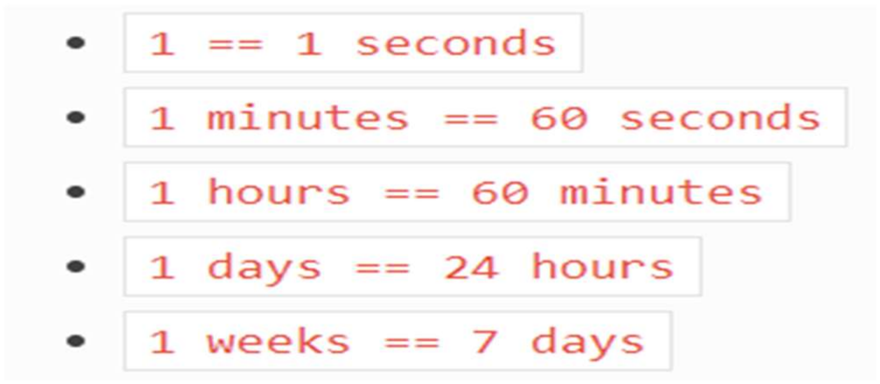
A literal number can take a suffix of `wei`, `gwei` or `ether` to specify a subdenomination of Ether, where Ether numbers without a postfix are assumed to be Wei.



```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);
```

Time Units

Suffixes like `seconds`, `minutes`, `hours`, `days` and `weeks` after literal numbers can be used to specify **units** of time where seconds are the base unit and units are considered naively in the following way:



- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`

Special Variables and Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

Block and Transaction Properties

- blockhash(uint blockNumber) returns (bytes32): hash of the given block when blocknumber is one of the 256 most recent blocks; otherwise returns zero
- block.basefee (uint): current block's base fee (EIP-3198 and EIP-1559)
- block.chainid (uint): current chain id
- block.coinbase (address payable): current block miner's address
- block.difficulty (uint): current block difficulty
- block.gaslimit (uint): current block gaslimit
- block.number (uint): current block number
- block.timestamp (uint): current block timestamp as seconds since unix epoch
- gasleft() returns (uint256): remaining gas
- msg.data (bytes calldata): complete calldata
- msg.sender (address): sender of the message (current call)
- msg.sig (bytes4): first four bytes of the calldata (i.e. function identifier)
- msg.value (uint): number of wei sent with the message
- tx.gasprice (uint): gas price of the transaction
- tx.origin (address): sender of the transaction (full call chain)