**# Crypto-Zombies**

1. Fixed length array uint [5] arr; length cannot be increase.
2. Array variable can be limited like address[] memory pools = new address[](2); .
3. Variable 2 types i.e., value and reference. https://www.geeksforgeeks.org/solidity-types/
4. Encoding is done using keccak256 which expect single arg in bytes. https://cryptozombies.io/en/lesson/1/chapter/11
5. Hash of specific length can be attained using modulo i.e.; hash % (10 \*\* length).
6. Modulo only works for uint/int data type.
7. Key can be incremented for mapping like mapping[key]++; .
8. An account can be used by using it with interface i.e.; ```ContractInterface contractVar = KittyInterface(contractAddress);```. The function then can be called with method``` contractVar.func(args)```.
9. Renounce ownership is setting owner to address(0) which cannot be called anymore. https://cryptozombies.io/en/lesson/3/chapter/2
10. Better to emit event just before changes takes place.
11. If file is imported all contracts will be inherited, if a specific contract is called then it will be inherited then.
12. Inheritance rule is followed from origin if a base contract inherits a contract it can be inherited to every other contract that inherits base contract.
13. Inheritance depends on import..import is linking file name, inheritance is linking with contract name.
14. Use smallest integers inside struct for minimum storage space.
15. Storage pointer can be used as arg inside a internal or private function.
16. Identical data types packing cost less gas as they are clustered together. https://cryptozombies.io/en/lesson/3/chapter/4
17. Uint default return value is 256 bits.
18. Tuple are structs which requires input value to be passed with same arguments numbers as in struct. https://remix-ide.readthedocs.io/en/latest/udapp.html#passing-in-a-tuple-or-a-struct-to-a-function
19. Calldata is for external functions while memory is for internal. View functions are gas free for external function.
20. Storage operations are most expensive mainly for writing.
21. Memory arrays are created with fixed length as they cannot be resize by using push method.
22. Memory array inside external function with view attribute built with for loop is overall cheap in terms of gas.
23. Bytes(str).length method for knowing string length.
24. Functions in interface should be of same name with return value to be placed in order according to the main contract(same applied to implementing contracts.
25. keccak data type is bytes32...abi. datatype is bytes with storage as memory or calldata.
26. The ‘unchecked’ keyword is used for particular operation without any checks. When ‘unchecked’ is used, Solidity skips these default checks and assumes that the operation will not result in any issues.
27. Unchecked can cause arithmetic over and underflow and should be used cautiously.
28. Require and if condition works in an opposite way (mainly for the relational operators).
29. View functions don't cost any gas when they're called externally. View function when called by a non-view function internally within a contract cost gas.
30. Overflows occur when the value to be represented exceeds the range of values representable by a given type. The integer overflow occurs when a number is greater than the maximum value the data type can hold. Uint8 = max[255] + 1 overflow leads to 0, Uint8 = min[0 ] - 1 overflow leads to 255 as it is a –ve overflow.
31. Underflows occur when the precision a type offers isn't enough to fully represent the result of an operation. The integer underflow occurs when a number is smaller than the minimum value the data type can hold. Underflow example is when denom is smaller than numerator 3/2.
32. new uint[](3); means initiating an array with length 3. Values are assigned as values[0] = 1;
33. An array can be build inside a view function with memory pointer which make it retrieving data without any spending any gas.
34. Code ordering is important in making logic to be strong.
35. The keyword block.timestamp is constant as storage variable in contract but updated when called within a function.
36. To pass message in error function string keyword must be declared inside error thisError(string).
37. Modifiers are of 4 types i.e., visibility, state, custom and payable. Visibility is pvt, public, internal and external; state are view and pure; and custom are defined inside contracts. Payable modifier provides function with ability to receive ether.
38. Ether can only be transfer to a payable address.
39. Send and transfer are carried out using ( ) while call is done through call{value: msg.value}(" "); for transferring ether.
40. call{value: msg.value}(" "); msg.value is the amount transfer while (" "); is checking return value. https://consensys.io/diligence/blog/2019/09/stop-using-soliditys-transfer-now/
41. .call( ) is prone to reentrancy and should be used with caution.
42. In solidity 0.8 and above transfer to caller is like this payable(msg.sender).transfer(address( this) .balance).
43. Random values can be created in solidity using this keccak256(abi.encode(block.timestamp, msg.sender, randNonce))), this is due to all params will be different.
44. % \_modulus will limit number of number generated like for 100 it will give number from 1-99.
45. Always put condition first in require that have higher computation process i.e., if (ids[msg.sender] == id) as this save gas.
46. Redundant condition can be used as modifier for better access control.
47. Memory pointer does not write to state variables. Memory pointer has block scope. Always use storage pointer to write data on state variables.
48. Implementing a contract is linked to creating its interface which includes all functions and events. Then it is inherited in the contract. Functions in interface have external visibility with name to remain same.
49. Function and modifier cannot have similar name and will lead to error.
50. Mapping[num] = address; is the method to change address of a mapping.
51. Library is a contract that is attached to functions with specific data type i.e., using library for data type. Library must be linked with the contract either making it internal to file or import.
52. An argument can be passed to an function through variable with accessing i.e.; uint test = 2; test = test.mul(3);.
53. require refund the rest of gas when a function fails, whereas assert not.
54. Safemath method can be instead of decrement or increment like instead maker++ use maker.add(1).
55. Natspec are comments with different tags like author, dev, title, param, audit etc.

**# Patrick-Foundry - solidity**

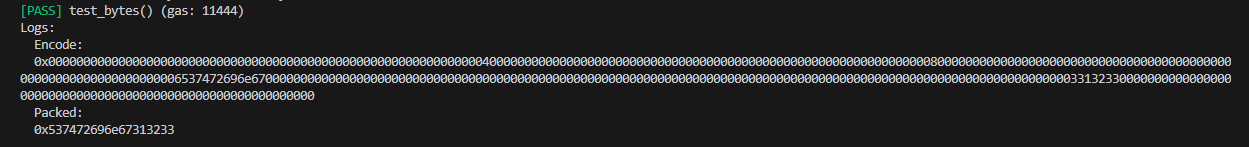
1. Code are executed in top to bottom and left to right order.
2. Bytes and bytes32 have different bytes values in solidity.
3. Strings are converted to bytes.
4. View function does not allow state change. They are gas less and will cost gas only when they are called by another function internally.
5. Struct are custom data type that can have different data types in its. Structs are declared with pointers according to need.
6. Values can be passed simply according to data location or json key pair value.
7. Datatype also need to be declared while passing values.
8. Array declared with data types are dynamic i.e., struct[] arr. Static array are defining number of values during declaration struct[5] arr.
9. 10 \*\* 18 and 1e18 is same thing.
10. Struct[index].datatype is method to read value inside of struct at a given index.
11. keccak256(abi.encode(string ) is method to convert string into bytes.
12. 6 areas evm store information i.e.; storage, memory, call data, code, stack and logs[events].
13. Calldata and memory are temporary storage executed during function.
14. Calldata cannot be modified. Storage and memory data types can be modified.
15. Data storage are defined for struct, arrays and string.
16. Same return type should be declared in returns in which data is stored i.e.; simpleContract myContract should return simpleContract;.
17. Chainlink price feed return 8 decimal which can be converted to 18 using the method...return uint256(price \* 1e10).
18. Chainlink latestRoundData returns 5 parameter of which second is taken i.e. price (, int256 price,,,).
19. Msg.value can be converted to token price in usd by formula: ```msg.value \* latestrounddata / 1e18```.
20. Token price in usd can be converted to msg.value using formula ```(amountInUSD \* 1e18) / TokenPriceInUSD;
21. Amount difference can be calculated using above point formula.
22. Multiple input can be added to array variable using loops.
23. arrVar = new array[](0); is the method to delete every index of array.
24. Value increment in mapping should be use with += i.e.; map[address] += uint;
25. Modifier should be used if same condition have to be applied to different function.
26. Custom error an also be used instead of require condition.
27. Multiplication should always be carried out before division with also ensuring denominator is lower than numerator.
28. Transferring amount are carried out using transfer, send and call method.
29. Transfer and send have 2300 gas limit. Transfer throws error if txn need more than 2300 gas, Call fwd all the gas
30. Send and call method are low level calls which return bool value used to check for revert.
31. .call method is used to call function of contract `.call("")`.
32. Receive and fallback functions are used to receive ether on contract. Fallback is called when no function with specific name exist and some data is sent with ether.
33. Libraries cannot be inherited. Libraries are used inside contract for specific data type i.e., ‘using library for uint’.
34. When using a function of library inside a contract, its first parameter must be the data type for which it is used.
35. dynamic datatypes (bytes, string or array) are not use with calldata instead we use the memory keyword.
36. calldata is a read-only temporary memory location that holds the function arguments sent to the function by the caller (EOA or another smart contract). You cannot instantiate calldata in the contract.
37. Amount in wei \* tokenPrice / 1e18 gets the token value in decimal[USD].
38. Amount in wei \* 1e18 / tokenPrice gets the token value in ether.
39. Struct with mappings cannot have pointers in memory.
40. Updating time variables through function should be carried out in last to avoid collision with condition to check for time.
41. In require condition the value which is target(influence the state change) should come first i.e.; require(block.timestamp > timeLimit).
42. Script contract must have run(), and test must have setup() as the first function.
43. Test contract should inherit script as well as the main contract.
44. Struct does not allow mapping and array in it.... https://github.com/ethereum/solidity/issues/12302
45. Constructor cannot have visibility but can be mark payable.
46. Indexed events are topics...normal events are data that are merged with abi.
47. If and require condition are backward compatible.
48. Enums are custom data types which act mainly in context of Boolean condition. They can be used similarly to struct by defining them and using them with pointer variables.
49. Default value of enum will be the 1st element.
50. Enums can be converted into uint by type casting them like uint256(enum.elementIndex);
51. Error can be revert with data by declaring data in them. “revert error\_name(address, uint, bool)”.
52. Empty arguments can be passed for function if like ``` function check\_UpKeep(bytes memory /\* check\*/)``` and will be called in like ```check\_UpKeep(“”)```.
53. Values in struct can be passed simply as struct as well as using json key-pair value syntax.
54. Any type returns in a function must be returned in its type or casted to it type before returning, e.g. returns (datatype memory) {return datatype()}.
55. Stack too deep error can occur if many local variables are used in a function. The solution is to divide them into two or more parts.
56. Vm.prank makes call to only one state changing function in a single txn but vm.startPrank can makes multiple calls.
57. Event in foundry can be tested using ```expectEmit``` which takes upto 5 params after which event is emitted followed by actual txn that is emitting the event in the contract function.
58. Events must be separately defined in the test contract with same name and parameters(datatypes).
59. ERC-677 are upgradeable version and erc-777 as they are backward compatible erc-20.
60. If a contract is inherited (and have a constructor), it constructor can be passed as a modifier with the parent contract constructor.
61. An account can receive ERC20 token even if it does not have ```receive``` or ```fallback``` function.
62. ERC20 contract ```approve``` function returns true even if an address does not have ERC20 tokens and call the approve function.
63. ```transfer``` and ```send``` method will revert if receive and fallback function have any logic in it. This is due to 2300 gas limitation on these functions.
64. ``call`` method is low-level function which is called only by functions and not contractTypes. It returns bool and data.
65. Eth fund can be withdraw from contract through low level call even if there is no withdraw function in the contract.
66. Integers cannot be directly converted to strings it requires OZ’s string lib and ```Strings.toString(tokenID)``` syntax.
67. If multiple strings are required to concatenate they must be encode and then casted to string before concatenation.
68. Abi.encode/encodePacked is used to convert a data type into bytes. Abi.decode is used to convert bytes into its original datatype.
69. Integers conversion to bytes required first casting to its type ```uint16 or uint256``` then ```abi.encodePacked``` method is used.



1. Abi.decode is used to bytes into its original datatype, by explicitly defining datatype as 2nd argument.



1. Library should be declared for data types they are used for ```using Strings for uint256```, Strings library of OZ is used for uint data type.
2. Base64 data can be converted only when datatype is of ```bytes```.
3. Integer can be changed to enum value by casting ```enum(integer)```.
4. All elements in enum elements can be retrieved using ```uint(type(enum).max)```.
5. ```Abi.encode``` generate whole length bytes result while ```abi.encodePacked``` generated data without padding.



1. Abi.encode is actual encoding while abi.encodePacked is casting.
2. Hash of encodePacked can be converted simply to it original form by casting to its type while encode require decoding with type define.



1. Data encrypt from abi.encode can be decode using abi.decode(bytes, datatype). Multiple data can be encoded and decoded in same manner.



1. Function selector is 1st four bytes of a function having its name and types of param.
2. Function selector bytes code can be generated by using function signature.



1. Bytes is not required if function signature is converted inside a function while required if called from outside. ```bytes4(keccak256(bytes(func))); // method to generate function selector from function signature```
2. Function selector can also be generated by using foundry cast command e.g: cast sig "functionName(address,uint)".



1. Function selector can be generated using cmd ```this.functionName.selector``` this only works for function having external or public visibility.



1. Function signature is function name along with its parameters types e.g: "transferFrom(address, address, uint)".
2. Low level call can be made using function signature with method ``abi.encodeWithSignature(functionsignature, param)``.



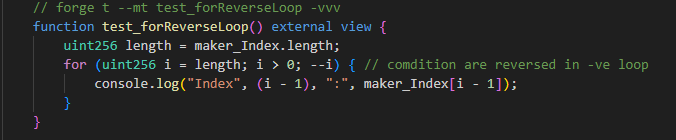
1. Function signature can be generated from function selector using foundry cast command e.g: cast 4byte functionselectorbyte.



1. Function selector can also be used to make low level calls with method ``abi.encodeWithSelector(functionselector, param)``.



1. Difference between calldata and memory is based on type of call made. Memory is call made from inside the contract and calldata is made from outside the contract.
2. ```Bytes.length``` is method to get data size of bytes.
3. ```Bytes``` is dynamic data which is used to save or return data without any limit.
4. Reverse loop have condition in reversed form with length starting from length of loop and run till condition is greater than 0 and decrement for i. The index taken is (i-1) inside loop code.



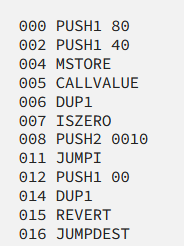
**# Foundry-test**

**###** [**https://github.com/Cyfrin/foundry-full-course-f23#lesson-7-foundry-fund-me**](https://github.com/Cyfrin/foundry-full-course-f23#lesson-7-foundry-fund-me)

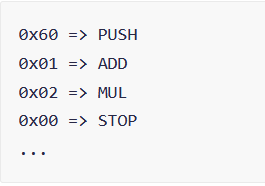
1. Foundry install on windows using git...guide YT smart contract programmer...’foundryup’ inside git install foundry on windows.
2. forge init is used to start a new project...forge build or compile for compiling contract.
3. In foundry contract are deployed locally on anvil. Anvil is local blockchain like ganache.
4. Forge create take contract name not contract file. Anvil needs to be run before contract deployment.
5. ``test`` is the main prefix to run test on any function and cannot run without it.
6. forge create contractName –private-key is command-line `CL` to deploy contract in foundry.
7. Forge create contractName –interactive is used to insert pvt key without showing.
8. forge create contractName --rpc-url $test\_rpc\_url --private-key $test\_pvt\_key --constructor-args constructor arguments is the method to deploy contract with constructor .
9. ```forge script deployRaffle --rpc-url $test\_rpc\_url --broadcast --private-key $test\_pvt\_key --etherscan-api-key $eth\_api –verify``` is cli for deploying and verifying contract through deployment.
10. Contract also can be deployed using script by writing script in script folder.
11. The main function in script is `run` which have the code for deployment. The contract file is imported and then the contract is deployed and saved as contract type variable which is returned.
12. To deploy contract onchain `forge script script/contract --rpc-url –broadcast –private-key` is the CL using script.
13. Create method to deploy contract onchain is linked to forge Contractname --rpc-url –broadcast –private-key` `` option.
14. Adding variables in .env file shouldn’t have gap b/w var name and values like `varName=0x512`
15. `source .env` is means to add .env into source file.
16. To call value from .env file they should be with `$` sign without any gap in between.
17. Broadcast is only used with script and not with create method. It requires pvtkey for wallet
18. –rpc-url is means to deploy on chain..in script deployment does not need pvt key with rpc but with create it need –private-key as well.
19. Forge clean remove the `out` folder.
20. `Cast to-base` is method to convert values.
21. Cast is also used to interact with a deployed contract using CL `cast send contractAddress ‘’functionName(uint param)’’ args123
22. Cast call can be made to check on result for a write function as a static call.
23. `cast call contractAddress "func(argumentuint256)" --rpc-url ` is the syntax to use for making call on-chain.
24. ‘vm’ cheat code ignore the next line if also the ‘vm’ is present.
25. Vm.prank(caller) method is used to initiate txn with specific addres instead of msg.sender or address (this).
26. Vm.deal(caller, 1 ether) is used to fund the contract with required amount.
27. hoax(caller, 1 ether) is combination of both prank and deal.
28. If an amount is to be sent for a payable contract it is called like a low level call. i.e.; contract.function{value: amount}();
29. –f is short form of –fork-url and doesn’t work with local host...only with testnet or mainnet.
30. Calling elements of array in test are done by caching them in array and calling each element separately (address[] memory fuser = (funded.getFunders());assertEq(fuser[0], caller));) or through loop.
31. Testing needs to be developed around 3 A’s i.e. Arrange, Act and Assert.
32. Arrange is setting all parameters and initial state of all variables...Act is calling the contract or method in contract...assert is validating state changes taken place in Act.
33. Branching tree Technique or BTT is method for organising unit testing based on state changing taken place at each step.
34. BTT starts with default behaviour of contract and then modifier are created and added before each state changing.
35. Vm.warp() is the cheat code to create a time difference between two txns. By default it is 1 i.e., vm.warp(block.timestamp) is 1.
36. Receive and fallback is must for contract receiving native token...payable function does not provide contract with ability to receive ether.
37. Contract abi can be generated through cmd “forge inspect ContractName abi”. Internal/pvt functions abi are not generated.
38. Contract abi can be convert back to interface using cmd “cast interface src/contract.sol” .
39. Foundry default time is 1 if using ```block.timestamp```. can be changes using ```vm.warp(129)``` returns current time ==129.
40. Value in scientific notation can be returned in console by using ```%e``` syntax, e.g.; ```console.log(‘This is one ether %e:’, 1 ether )```.
41. Vm.startbroadcast is used to deploy a contract within a function.
42. Vm.startbroadcast makes foundry default sender as owner which can be changed accordingly i.e., ``` Vm.startbroadcast(user)```.
43. Passing values in variables from a function or struct from a inherited contract must have parenthese”()” else it allows only 1 parameter to be passed.
44. assertEq() only checks for specific data types which requires casting to the needed type first.
45. Vm.expectRevert(‘custom error string’) custom error message should be passed as same define in the code.
46. Vm.expectRevert(contractType.customError.selector) is used when custom error is used for revert in contract having no parameters.
47. Vm.expectRevert(abi.encodewithSelector(contractType.customError.selector, param1, param2, ...) is used when custom error have parameters in it .
48. expectRevert with custom error and arguments only allows 3 params.
49. Custom error shows message for the reason a txn fails, while empty ```revert()``` shows ```EvmError: Revert```.
50. Events are required to be separately defined in test contract.
51. ```vm.expectEmit(true, false, false, false, address(emitter))``` takes max of 5 args...1st 4 are bool, last one is emitting contract address. 1st argument must be true to pass the test or pass empty parentheses for default check.
52. If no args are passed in ```vm.expectEmit()``` first topic is true by default.
53. If only address is passed in last parameter ```vm.expectEmit(address(emitter))``` it checks the emitter of the event which is the contract.
54. Events parameter if is an address it should be indexed, as only topics are checked.
55. Events are stored using ``vm.recordLogs()`` syntax in foundry. After this all txn are made which emits event and are stored.
56. ```Vm.Log[] memory event\_Entries = vm.getRecordedLogs()``` is expression which is used for accessing all events.
57. Events emitted are stored in 1st index of topic as 0th index stored the whole event expression.
58. Events are stored in bytes32 type and all events must be casted to type bytes32 before validating any event emitted.
59. Converting address into bytes32 require first them to be casted to 20 bytes which is done by uint160 then into uint256 to be casted into bytes32 ```bytes32(uint256(uint160(funder)))```.
60. ```forge coverage --report debug``` is syntax to get coverage report for each line in contract.
61. Continue skips the current iteration while ``break`` stop the loop at current iteration.
62. There are different method of making address in foundry.
63. The true return of ERC20 contract approve can cause ```expectRevert``` call to fail as it expect return value to be false.
64. Address are passed with datatype pointer in deployment script and not with casting. ContractType pointer needs to be passed with pointer and not with casting into Address type.
65. Address of pointer to a contract type for a contract can be retrieve using ``.address`` syntax, e.g; ``contract.ContractTypePointer.address``.
66. Caller can be checked of a function by returning ```msg.sender``` in contract function and caching in testing function.
67. ```gasleft()``` is method for getting gas price for txn. The formula is ```uint256 gasUsed = (Start\_Gas - End\_Gas) \* tx.gasprice;``` where ```gasleft()``` is placed at start and end of txn.
68. Gas price in eth can be calculated depend on price of Gwei, i.e., formula ```(190000(gas consumed) \* 10 Gwei(depends on network activity) \*1e9) / 1e18 = 0.0019 ETH(gas price)```.
69. If low level call return value is not check reverting txn will pass.
70. Foundry devops tool only get contracts that are deployed on chain and are in ```broadcast``` folder.
71. Contract should be deployed with ```vm.startBroadcast()``` to have them in ```broadcast``` folder.
72. Persmission should be given with ```fs\_permissions = [{ access = "read", path = "./broadcast" }]``` for correct functioning of devops.
73. ```vm.readFile(relative path)``` is foundry cheatcode use to access data of other files.
74. vm.readFile("relative path") returns data in string form.

**# EVM Bytecode- Solidity**

1. Smart contract bytescode can be generated in remix compile tab on ‘Compilations Details` tab under bytecode object.
2. Solidity/EVM version and optimization settings have direct impact on contract bytecode.
3. Activating debugger in remix gives details in instructions section.

****

1. Every instruction have an opcode to execute. EVM identifies specifics opcodes with particular action.

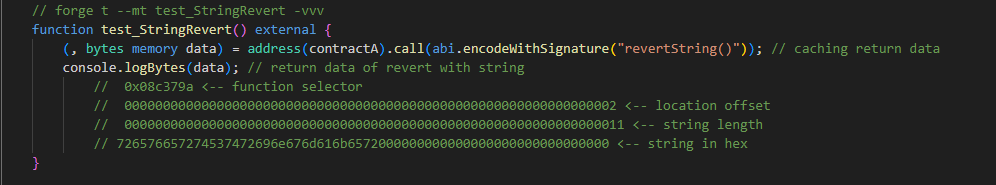


1. EVM opcode reference <https://www.evm.codes/>
2. Codes executes from top down order with changing location around with ```JUMP``` opcode.
3. ```JUMP``` opcode takes the most top value of stack and move to target location.
4. The target location of ```JUMP``` opcode must contain ```JUMPDEST``` opcode.
5. ```JUMPDEST``` opcode mark the target location, valid jump target.
6. ```JUMPI``` opcode only work if 2nd position in stack does not have 0 value, else it fails.
7. ```STOP``` opcode completely halts contract executions with no return value while ```RETURN``` also halt but with return data from EVM memory.
8. ```JUMP``` opcode make a jump with value that is pushed on stack before it, from its location to the target location ```JUMPDEST```.
9. Contract creation in evm have ```JUMPI``` opcode with following a ```Revert``` and ```JUMPDEST```.
10. Following instruction will lead to ```Revert``` and if not, it will jump to target ```JUMPDEST``` location.
11. The value taken to ```JUMPDEST``` location is value of ```PUSH2``` opcode before the ```JUMPI``` opcode.
12. The ``Return``` and ```STOP``` opcode points towards end of creation code from the very first instruction.
13. Creation code is not part of contract as it is executed only once.
14. Creation code sets contract initial stage and returning copy of runtime code.
15. Contract constructor is part of creation code and not runtime code but return the runtime code copy as it is the actual code of the contract.
16. ``PUSH1`` opcode push 1 byte data on top of stack.
17. ```Mstore``` opcode grabs last 2 bytes from stack and store one of them (usually first one into 2nd one) in memory.
18. `` PUSH`` instructions are composed of 2 or more bytes if a ``PUSH`` opcode is given it will skip next number on instruction depend on its number in opcode.
19. The free memory pointer is opcode that is present till ``MStore`` opcode.
20. After ```MSTORE```, if ``PUSH1`` opcode is present the constructor is marked payable.
21. Payable constructor after ``PUSH1`` opcode directly proceed to return which is end of creation code.
22. After ```MSTORE```, if ``CALLDATA`` opcode is present the constructor is non payable.
23. Non-payable constructor have check starting from ``CALLDATA`` opcode and ends at ``REVERT`` opcode.
24. ``CALLDATA`` opcode in present only when constructor is non-payable which ensures if value sent != 0 it proceed to revert else it jumps.
25. All instruction in between ``CALLDATA`` and ``REVERT`` opcode are usually ``DUP1, ISZERO, PUSH2 and JUMPI`` opcode.
26. ``DUP1`` duplicates 1st element on stack, `` ISZERO`` pushes 1 byte to stack if is 0 at top most and `` PUSH2`` can push 2 bytes to stack.
27. ``JUMPI`` Opcode only works if no value is involved and will jump to next instruction i.e. ``JUMPDEST`` after the revert opcode.
28. After jump is complete at `` JUMPDEST``, ``POP`` (which is removing of a value from stack) Opcode indicates constructor presence.
29. If a value is passed in constructor it will be read and pushed to stack indicates through a ``PUSH1`` and ``MLOAD`` opcode.
30. ``DUP1`` is a duplicating of value at above instruction after ``PUSH1``.
31. After ``DUP1`` if a value is passed to constructor, it will have ``push1`` and ``push2`` followed by ``DUP`` opcode respectively.
32. If no value is passed after ``DUP1`` it will have ``push2`` and ``push1`` respectively.
33. Afterwards there will be ``CODECOPY`` opcode present in both case if value is passed or not to constructor.
34. ``CODECOPY`` opcode takes 3 arguments; memory position where result is copied to copy code from(in some case the immediate opcode before), instruction number of the code to copy from and bytes size of code [length] from the instruction.
35. When deploying a contract whose constructor contains parameters, the arguments are appended to the end of the code as raw hex data.
36. After ``CODECOPY`` the next instruction till ``MStore`` the value is updated at current position from previous value to current offset value if there is a constructor.

**# Miscellaneous Topics**

**## Try/Catch issue**

1. Ty/catch responds only to external function calls and contract creation error.
2. Response of Try/catch is based on return value and data of low-level calls, when it fails.
3. External call fails on 3 conditions i.e.; called contract (or contract function) reverts, called contract does an illegal operation (like dividing by zero or accessing an out-of-bounds array index) and called contract uses up all the gas.
4. If an empty revert() is present in the calling function it will return no data ``0x``.
5. If revert() have string message revert(‘revertMessage’) the return data will be abi encoding of the function `` Error(string)``.
6. Revert error can be empty or takes only string arguments in it.
7. The abi encoding of function error includes function selector of `` Error(string)``, offset(location) of string, length of string in bytes and content of the string encoded in hexadecimal.



1. If a function have custom error ``'emptyError()'``without arguments the return data will be first four bytes of the error function selector i.e. ``0xa97a0bd2``.If a function have custom error `` errorArgs(address)'``with arguments the return data will be first four bytes of the error function selector i.e. ``0xad682f1b`` and next 32 bytes will be address of the caller.
2. If function reverts with require statement without error message it will return ``0x`` data similar to empty revert.
3. When function reverts with require statement the return data will be similar to revert with string message, i.e.; abi encoding of error function.

**# IPFS**

1. Any file can be uploaded/import using IPFS desktop application.
2. ```ipfs://``` is used on supported browser for read-only purpose.

**## Script to initialize a project of foundry**

```sh

# 1st try this

forge i foundry-rs/forge-std --no-commit

# If above cmd fail install lib/forge-std

forge init --force --no-commit

# Install openzeppelin-contracts gitmodules◊// same for any other dependencies

# all installation will be on latest version if specific version is require use @v4.9.4

forge i OpenZeppelin/openzeppelin-contracts --no-commit

forge i OpenZeppelin/openzeppelin-contracts-upgradeable --no-commit

# Run test

forge t

```

**# Auditing**

https://github.com/ComposableSecurity/SCSVS/tree/master

https://gist.github.com/Abbasjafri-syed/773bef4cd2d199dc083221127c43684e

https://lab.guardianaudits.com/encyclopedia-of-solidity-attack-vectors/block.timestamp-manipulation

https://github.com/0xNazgul/Blockchain-Security-Library

Test tokens on Sepolia <https://blog.sui.io/sui-bridge-live-on-testnet-with-incentives/>