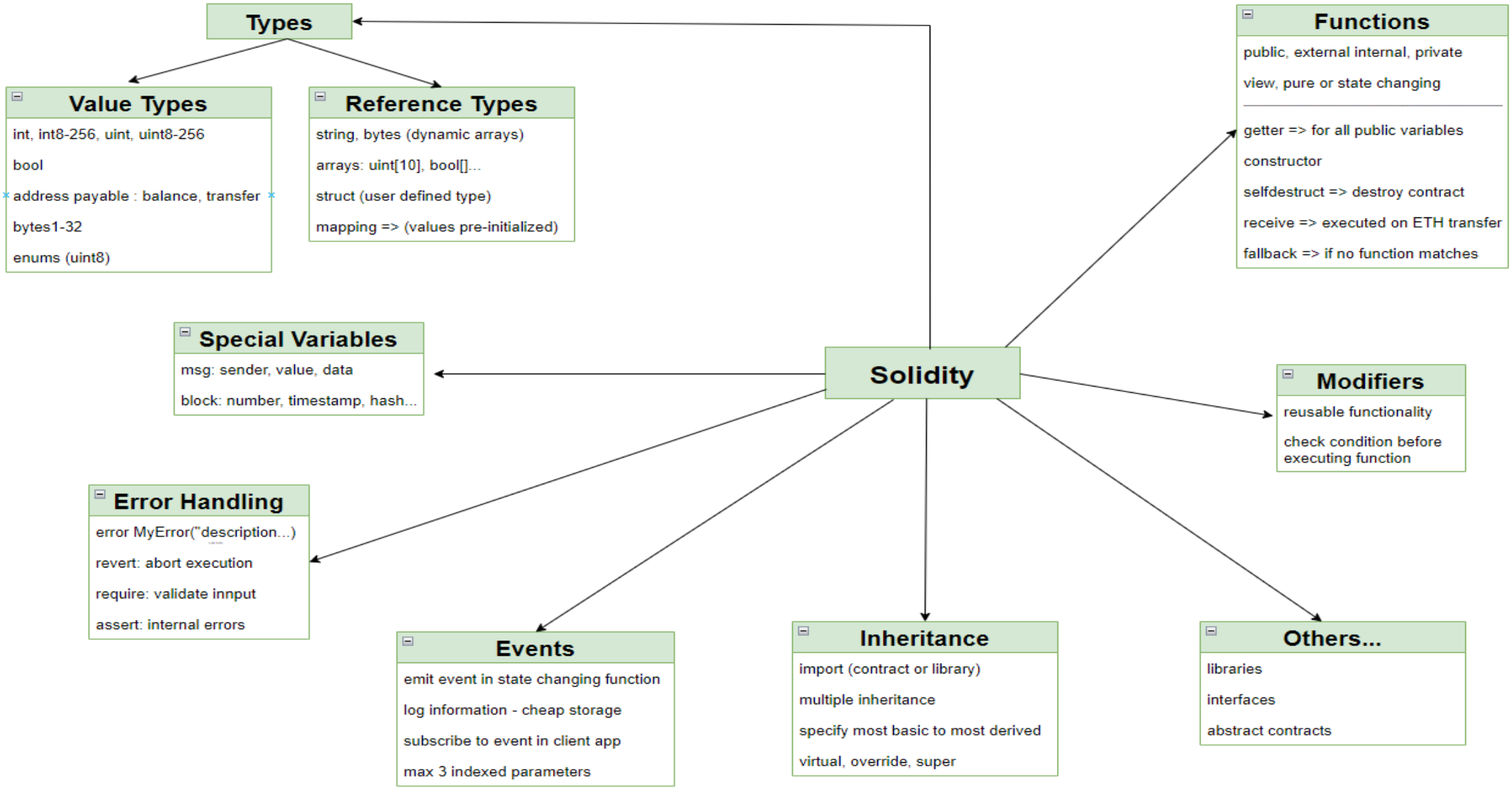

ETHEREUM SMART
CONTRACT DEVELOPMENT
Solidity

Solidity docs: https://docs.soliditylang.org/en/v0.8.17



Sample Smart Contract

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import "../SomeOtherContract.sol";

contract Voting is SomeOtherContract {
    address public owner;

    struct Proposal {
        string name;
        uint voteCount;
    }

    struct Voter {
        bool allowedToVote;
        bool voted;
    }

    mapping(address => Voter) public voters;

    Proposal[] public proposals;

    event UserHasVoted(address indexed voter, uint proposal);
```

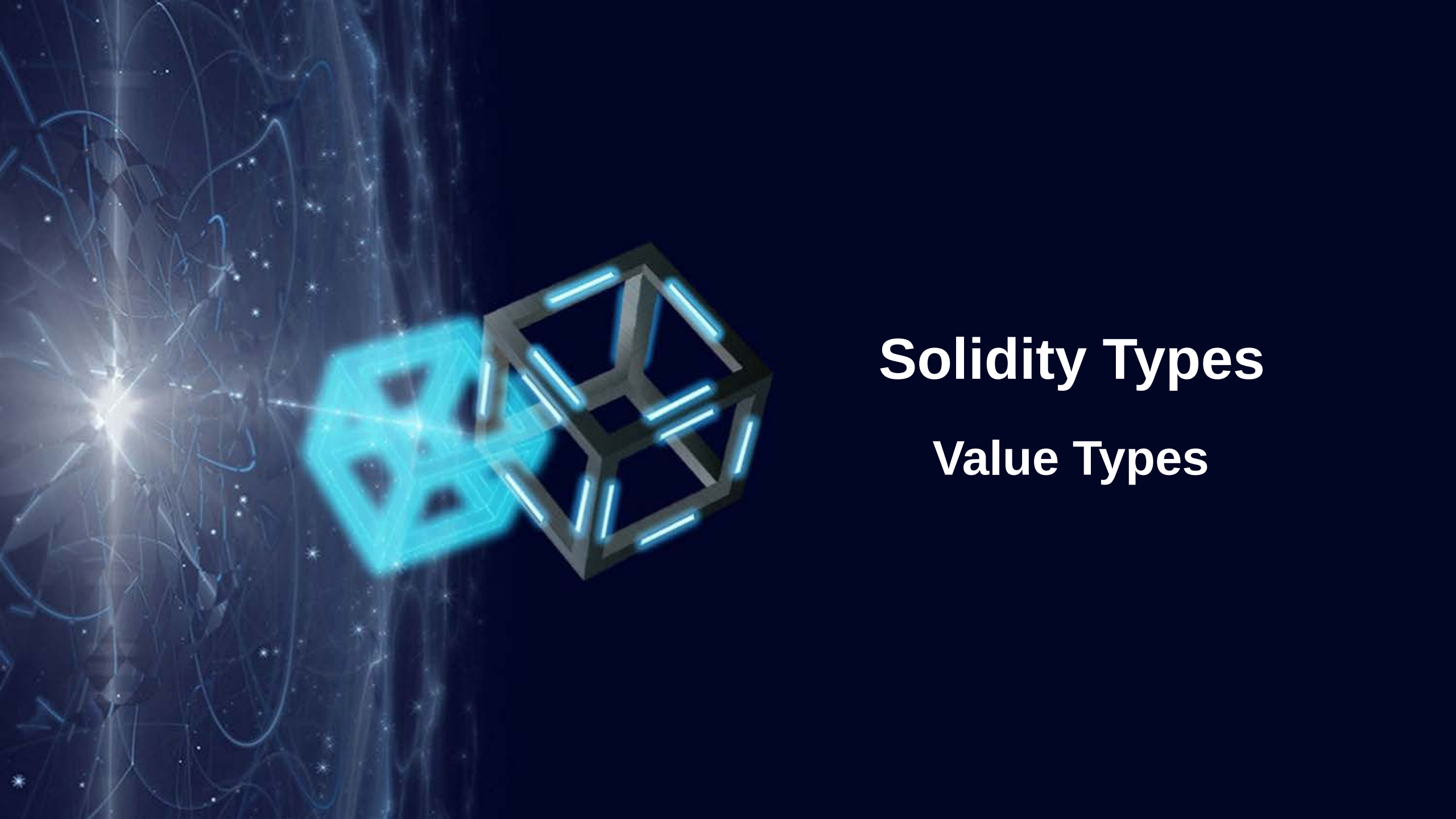
```solidity
    ///The user has already voted
    error UserHasAlreadyVoted(address voter);

    modifier onlyOwner() {
        require(msg.sender == owner, "You are not authorized");
        _;
    }

    constructor(string[] memory proposalNames) {
        owner = msg.sender;
        voters[owner].allowedToVote = true;
    }

    function giveRightToVote(address voter) external onlyOwner {
        if (voters[voter].voted) revert UserHasAlreadyVoted(voter);
        voters[voter].allowedToVote = true;
    }
}
```

Features and Basic Layout of a Solidity Smart Contract 1/2

- SPDX license identifier - the solidity compiler encourages the use of a SPDX License Identifier
- Pragmas - pragma solidity defines the compiler version to be used
- Imports
- Comments - standard comments and NatSpec comments
- State variables - permanently stored in contract storage
- Reference and value data types: int, uint, bool, address, arrays, string, bytes
- Mappings - initialized key/value pairs, like a hash table
- Struct - custom defined types that group several variables

Features and Basic Layout of a Solidity Smart Contract 2/2

- Enum - custom types with a finite set of values
- Events - to log specific information for client apps
- Modifiers - amend functions in a declarative way, for common checks applied to various functions
- Errors - define descriptive names and data for failure situations: used in revert statements
- Constructor - executed once during creation of contract
- Functions
- Control structures (if / else) and loops (for / while)
- Inheritance - multiple inheritance is supported



Solidity Types

Value Types

Solidity Types

- Solidity is a statically typed language - the type of each variable needs to be specified
- All variables are initialized by default: int = 0 ; bool = false ; string = ""...
- There is no null or undefined
- There are 2 categories of types: Value and reference types
- When we use a reference type, we also need to specify the location of the variable
  - memory
  - storage
  - calldata

Visibility / Value Types

Visibility of state variables (stored permanently in contract storage):

- ***public:*** The compiler automatically generates a getter function with the name of the variable. Can be accessed internally via the name of the variable and externally via the getter function
- ***internal:*** Can only be accessed from within the contract and from derived contracts. This is the default visibility
- ***private:*** Can only be accessed from within the contract

Value Types:

- int, uint, bool, address, fixed size byte arrays (bytes1, bytes2... bytes32)
- Value types provide an independent copy of the value
- Declaration: uint public myInt = 5; ... bool myBool = true;

Integers, Unsigned Integers and Boolean

- Default value for int/uint is 0 – default value for bool is false
- uint8 to uint256 in 8 bit increments
- uint8 from 0 to 255 (2 ** 8 = 256) ... uint256 from 0 to (2 ** 256) – 1
- int8 from -128 to 127
- uint / int is an alias for uint256 / int256
- Exponentiation is only available for unsigned types via the ** parameter

Overflow / Underflow – Unchecked Mode

- Solidity (since version 0.8.0) throws an error and reverts all state changes when we leave the defined value range (on overflow and underflow)

- To obtain the previous behavior (wrapping mode) an "unchecked {...}" block needs to be used.

```
uint8 public myInt;

function decrement() public {
    myInt = 0;
    myInt--; //throws an error and reverts

    unchecked {
        myInt--;
        console.logUint(myInt);
    }
}
```

Address Type

- Declaration: address public myAddress;
- Default value is 0x0000...
- Every interaction on Ethereum is address based
- 20 byte value (40 characters)
- 2 Types: address and address payable
- Conversion from address to address payable must be explicit via payable(myAddress)
- A contract type can be converted to address or address payable: payable(address(this)) => if it can receive Ether (has a payable fallback or receive function)

Address Type

- address member: balance
- address payable members: transfer (revert on failure) and send (returns false on failure)
- send and transfer provide only 2300 gas => instead, use: receiverAddress.call{value: 1 ether}(“”);

```
function transferEther() public {
    address payable address2 = payable(0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2);
    address addressContract = address(this);

    console.log("Balance of contract: ", addressContract.balance);

    if (addressContract.balance >= 1 ether) {
        //address2.transfer(1 ether);
        (bool success, ) = address2.call{value: 1 ether}("");
        require(success, "Transfer failed.");
    }
}
```

Fixed Size Byte Arrays and Enums

Fixed size byte arrays:

- bytes1... bytes32: holds up to 32 bytes
- access individual bytes by index
- Members: length
- Example: *bytes4 functionSig = bytes4(payload);*

Enums:

- Can be used to create a user-defined type
- Stored as uint8 - if we have more than 256 values, than uint16
- Can be converted to and from integer types
- Example: *enum Directions { Left, Right };*

Example: Value Types

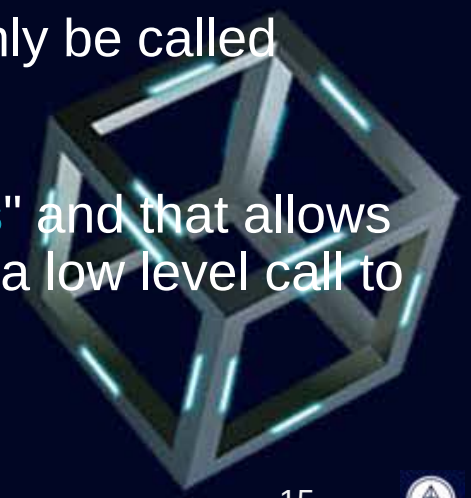
Create a smart contract that contains the following functionalities:

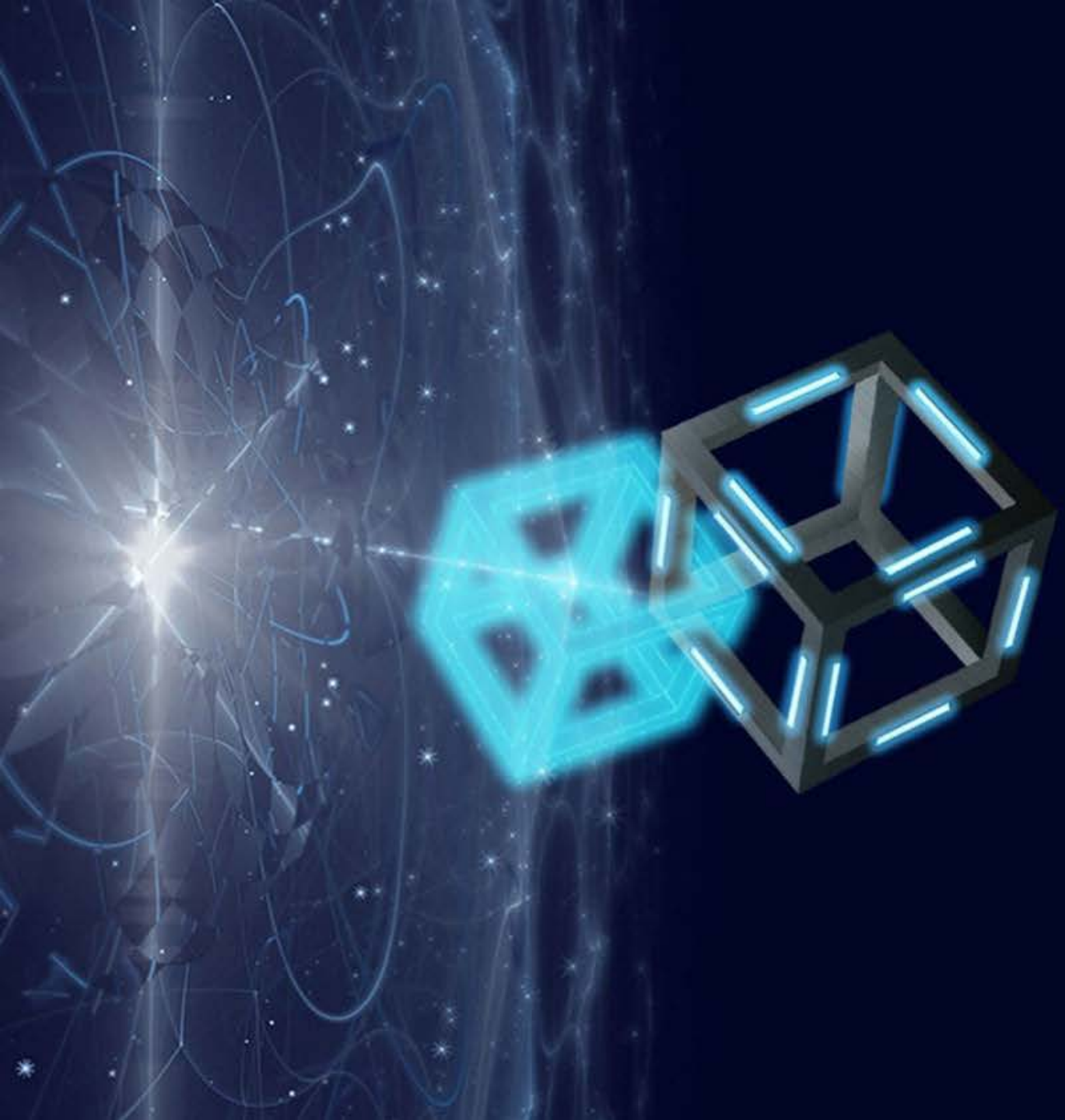
- Create various state variables for different value types
- Create a constructor that can receive ETH and that initializes some state variables
- Create a function that changes one of the state variables
- Create a function that returns the balance of the contract
- Create a function that allows to send ETH using transfer or send
- Create a function that allows to send ETH using call

Exercise

Create a smart contract that contains the following functionalities:

- Create a public state variable of type **uint256** with the name "**myValue**"
- Create a private state variable of type address with the name "**myAddress**"
- Add a constructor that initializes "**myValue**" to 5 and "**myAddress**" to the address of the contract deployer.
- Also, make sure the contract allows you to receive some ETH during deployment
- Add a function that allows you to change the value of "**myValue**" and that can only be called externally
- Create a function that contains one argument with the name "**withdrawAddress**" and that allows you to send the entire contract balance to "withdrawAddress". Make sure to use a low level call to forward the contract balance





Solidity Types

Reference Types

Reference Types

- Ref. types: structs, arrays, bytes, strings and mappings
- Pass a reference - the value of the variable can be modified through different names
- The location of reference variables needs to be specified - except for state variables, they are storage by default:
  - memory: temporary, lifetime is limited to external function call, low gas cost, use for intermediate calculation and store the result in storage
  - calldata: special data location for function arguments, non-persistent, non-modifiable, behaves like memory
  - storage: area where state variables are stored, persistent storage area (lifetime of the contract), high gas cost – should be avoided if possible

Reference Types – Arrays

- Can have compile-time fixed size (uint[3] myArr) or dynamic size (uint[ ] myArr)
- A getter is created for public arrays - only a single array element can be returned => the index needs to be specified on the getter function
- Member functions for all arrays: length
- Additional members for dynamic storage arrays and bytes: push(), push(value) and pop()
- Memory arrays can be created with the new operator:
  uint[ ] memory myArr = new uint[ ](7)
  myArr[0] = 1...
  they cannot be resized (the .push() function is not available)
- Array literals can be used to assign values to a fixed size array: *uint8[3] memory myArr = [1, 2, 3]*

Reference Types - Assignments

- Assignments between storage and memory always create a copy
- Assignments from memory to memory create a reference
- Assignments from storage to a local storage variable create a reference
- All other assignments to storage always create a copy

```
uint[] arr1; //data location is storge - can be omitted

function arrayTest(uint[] memory arr2) public {
    arr1 = arr2; //this creates an independant copy

    uint[] storage arr3 = arr1; //assigns a reference
    arr3[0] = 5; //modifies arr1 through arr3
    console.log("Value of arr1[0]: ", arr1[0]);
    arr1[1] = 10; //modifies arr3 through arr1
    console.log("Value of arr3[1]: ", arr3[1]);
}
```

Reference Types – Strings & Bytes

- Dynamically sized special arrays - for raw data (bytes) or UTF-8 encoded strings
- String don't have the length function and individual elements cannot be accessed by index
- Apart from string.concat(s1, s2...), there are no string manipulation functions
- Using strings is expensive and should be avoided
- bytes is similar to bytes1[ ] but tightly packed (bytes1[ ] adds 31 padding bytes in memory between elements) - bytes is cheaper than bytes1[ ]
- If the length can be limited to a certain number of bytes, it is better to use one of the value types: bytes1... bytes32 , because they are cheaper

Reference Types – Structs & Mappings

Struct:

- User defined types that group several variables
- Types in a struct are initialized with their default values

Mapping:

- Mappings are like hash tables - each value is pre-initialized
- Usage: mapping(keyType => valueType) myMapping
- Example: **mapping(address => uint) public balances**;
- Mappings are accessed like arrays, but there are no index-out-of-bounds exceptions and all possible key/value pairs are already initialized => any possible key can be accessed

Reference Types – Mappings

- Use mappings rather than arrays, because they are cheaper
- The keyType cannot be a struct, mapping or array (bytes and string are allowed)
- Mappings always have storage as data location
- Mappings cannot be used as parameter or return value for public functions
- A getter is created for mappings that are used as public state variable - the keyType needs to be used as parameter for the getter
- Mappings and structs are often used together
- Mappings cannot be iterated

Special Variables

msg:

- Provides data about the current message call
- msg.sender
- msg.value
- msg.data

block:

- Provides general information about the blockchain
- block.timestamp - in seconds since unix epoch
- block.chainid
- block.basefee
- block.number
- blockhash(uint blocknumber)

Reference Types – Struct & Mapping Example

```
contract Bank {
    struct Deposit {
        uint amount;
        uint timestamp;
    }

    struct AccountDetail {
        uint balance;
        uint numDeposits;
        mapping(uint => Deposit) deposits;
    }

    mapping(address => AccountDetail) public accounts;

    function deposit() payable public {
        ...
    }

    function withdraw(uint amount) public {
        ...
    }
}
```

Example: Reference Types

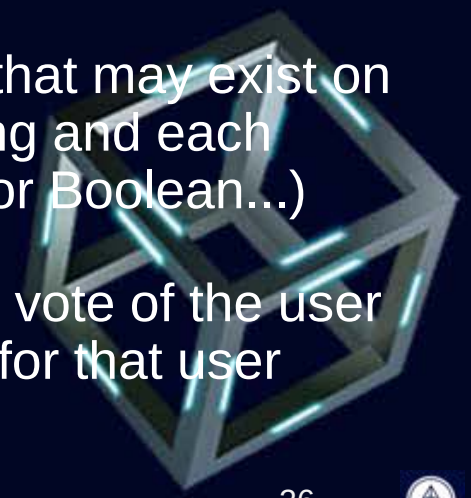
Create a smart contract that contains the following functionalities:

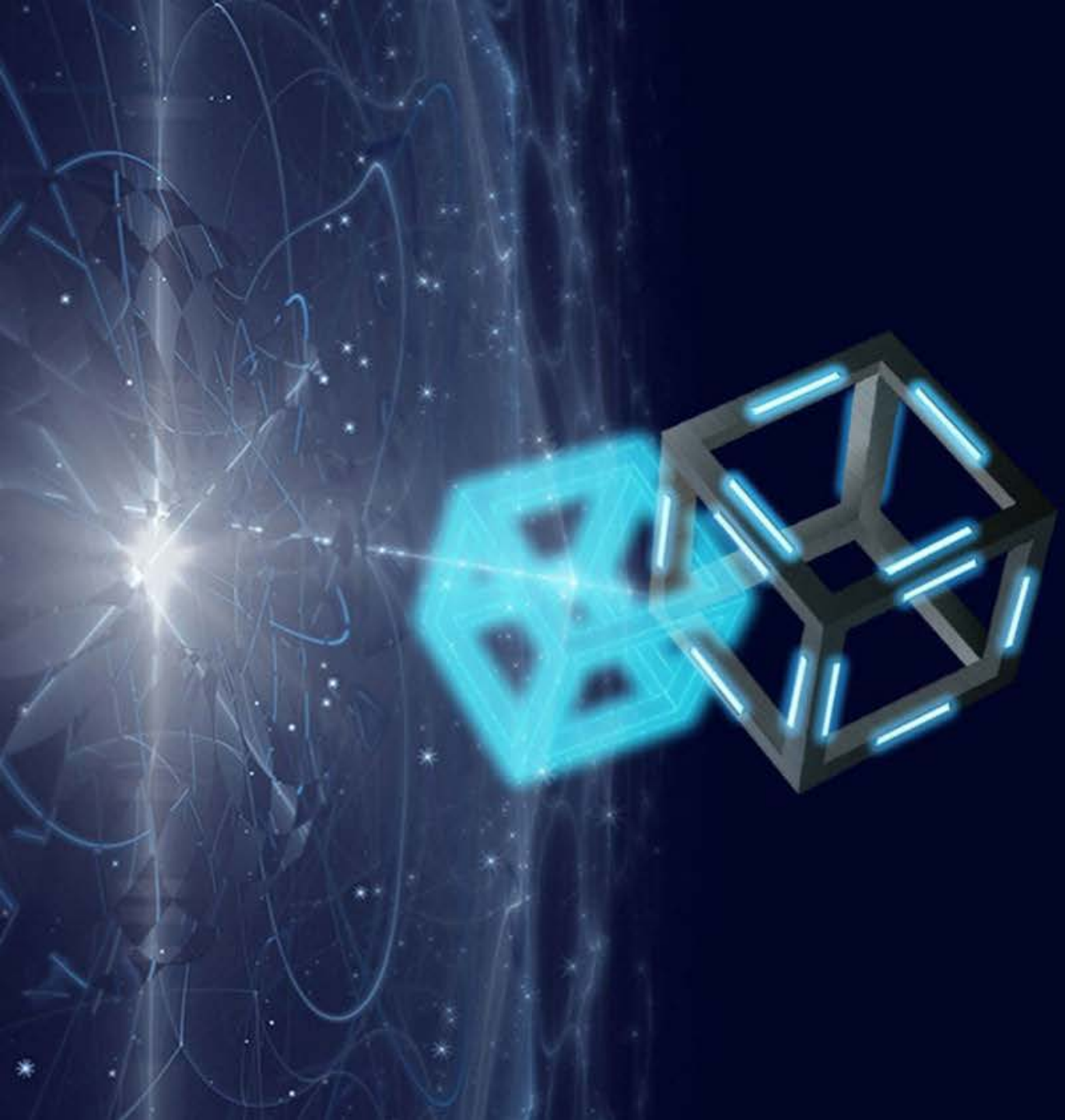
- Create 2 state variables: a string and a dynamic array that is initialized with the values 1, 2 and 3
- Create a struct (**Deposit**) that contains 2 types for amount and timestamp
- Create a mapping that manages the balances for all users and create a second mapping that stores only the latest deposit for each user
- Create a function (**changeValues**) that modifies the string state variable and that allows to modify the first value of the array state variable via a another locally defined array.
- Create a function (**deposit**) that allows to make a deposit and that updates the user balance and the latest deposit.

Exercise

Create a smart contract that contains the following functionalities:

- Create a public state variable (**arr1**) that is a dynamic array of signed integers with the values -1, 0 and 1
- Create a function that allows you to change the values of **arr1** through another array that needs to be declared in that function. You are not allowed to manipulate the values of arr1 by directly accessing the values of arr1, like: arr1[0] = 5;
- Create a struct (**Voter**) that defines 2 types: a type that indicates if the voter has already voted (**hasAlreadyVoted**), and a **uint** type that indicates the **vote** (vote = 1, 2, 3...)
- Create a mapping (**voters**) that assigns a **Voter struct** to any possible address that may exist on the Ethereum network. All those addresses need to be accessible on the mapping and each address needs to point to a struct that is initialized with its default values (false for Boolean...)
- Create a function that allows anyone to vote. The function takes 1 argument: the vote of the user (**uint userVote**) and updates the corresponding values of the "**voters**" mapping for that user





Error Handling Functions, Modifiers & Events

Error Handling

- Transactions are atomic - they either fail or succeed as a whole
- All errors and exceptions revert the state of the contract to its initial values

- ***require*(bool condition, [string memory message])** - typically used to validate input parameters, reverts if the condition is not met. Returns remaining gas

- ***revert*(string memory reason) or revert MyCustomError()** - aborts execution and reverts state changes. Using custom errors is much cheaper than providing information in a string - a detailed explanation of the error can be provided via NatSpec comments. Returns remaining gas

- ***assert*(bool condition)** - used for internal errors, if such an error occurs, the code is probably flawed and needs to be fixed. Reverts if the condition is not met. Consumes all gas.

  Assert is also triggered, if: division or modolo by zero, out of bounds index is accessed, convert a value too big to enum...

Error Handling Example

```
/// the provided amount must be 1 ETH
/// @param provided The amount provided by the user
error WrongAmount(uint provided);

function buySomethingFor1ETH() public payable {
    if (msg.value != 1 ether) revert WrongAmount(msg.value);

    // Alternative way to do it:
    require(msg.value != 1 ether, "Incorrect amount.");

    // Perform the purchase...
}
```

Example: Error Handling

Create a smart contract that contains the following functionalities:

- Create a custom error (**WrongAmount**) that returns the amount provided by the user
- Create a function that reverts (using **require**) if the transferred value is different than 1 ETH
- Create a function that reverts (using the **custom error**) if the transferred value is different than 1 ETH
- Create a function that uses an **assert** statement to check the value of an invariant

Functions

- Modify state => require a transaction – costs a fee gas
- Read state => message call is performed (against a single node) - is free
- **View** functions only read from state, but don't modify state
- **Pure** functions neither read nor modify state
- View functions can call other view and pure functions
- Pure functions can only call other pure functions
- The names of return parameters can be omitted
- Either explicitly assign return variables and then leave the function (without a return statement), or provide one or more return values with the **return** statement

Read & Write Functions

Statements that are considered to modify state (function cannot be view or pure):

- Modifying a state variable
- Emitting an event
- Creating a contract
- Using **selfdestruct**
- Sending Ether
- Calling another function that modifies state

Statements that are considered to read state (function cannot be pure):

- Accessing any state variables
- Accessing **address.balance**
- Accessing any of the members of **block or msg**

Read & Write Functions Examples

```
function funcWrite(uint8 val) public {
    myInt = val;
}

function funcPure(uint a, uint b) internal pure returns (uint sum, uint prod) {
    sum = a + b;
    prod = a * b;
    //return (a+b, a*b);
}

function funcView() public view returns (uint) {
    (uint add, uint multiply) = funcPure(3, 5);
    return myInt + add + multiply;
}
```

Function Visibility & Overloading

Function visibility:

- ***public:*** can be called internally and externally
- ***external:*** can be called externally and from other contracts. Cheaper than public => use external if you are sure the function will only be called externally.
- ***private:*** can only be called from within the contract
- ***internal:*** can only be called from within the contract and from derived contracts. Calling functions internally is cheaper and more efficient than calling them externally

Function overloading:

- A contract can have multiple functions with the same name but with different parameters

```
function myFunc(uint value) public view returns (uint) {...
function myFunc(uint value, bool flag) public view returns (uint) {...
```

Special Function

getter:

- The compiler automatically creates a getter function for all public state variables
- A getter function for an array returns only one element of the array – the element corresponding to the provided index
- To call a getter function from another contract, use: ***someContract.myStateVar();***

constructor:

- Called only once during deployment of the contract
- Arguments can be provided
- Initialize state variables - for example the owner of the contract
- Needs to marked as **payable** if funds should be transferred during deployment

Special Function – selfdestruct & receive

selfdestruct:

- renders the contract unusable
- transaction history remains on the blockchain
- specify address to which the remaining funds will be sent => ***selfdestruct(owner);***

receive:

- A contract can have 1 receive function. The receive function cannot have arguments, cannot return anything and must have external visibility and be payable.
- ***receive() external payable {...}***
- receive() is executed on plain Ether transfers
- The receive function can only rely on 2300 gas if **send or transfer** is used => not many operations can be performed (for example: emitting an event)

Special Function – fallback

- A contract can have 1 fallback function that has external visibility
- ***fallback() external [payable] - or:***
  fallback (bytes calldata input) external [payable] returns (bytes memory output)
- The fallback function is executed on a call to the contract if none of the other functions match the given function signature
- The fallback function can receive data, but in order to also receive Ether it must be marked **payable**
- If **receive()** does not exists, but a payable fallback function exists, the fallback function will be called on a plain Ether transfer
- If neither a receive nor a payable fallback function is present, the contract cannot receive Ether

Special Function Examples

```
constructor(uint8 valueForMyInt) payable {
    owner = payable(msg.sender);
    myInt = valueForMyInt;
    direction = Directions.Right;
}

function destroySmartContract() public onlyOwner {
    selfdestruct(owner);
}

receive() external payable {
    emit Received(msg.sender, msg.value);
}

fallback() external {
    myInt = 55;
}
```

Modifiers

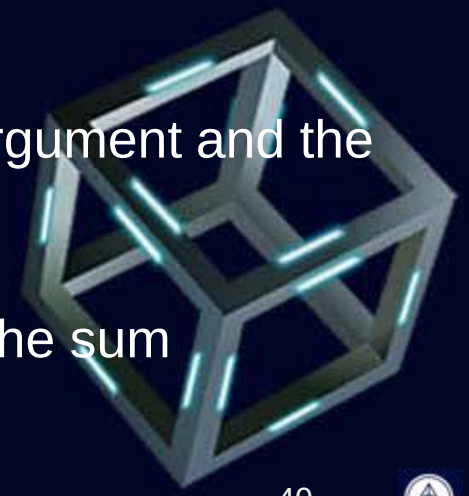
- Modifiers are used to add reusable functionality to functions in a declarative manner
- Typically used to check specific conditions before executing the function
- Arguments can be provided
- Multiple modifiers can be applied to a function – separated by whitespace and executed in the order presented
- The symbol _ in the modifier is replaced with the function body

```
address payable public owner;

modifier onlyOwner() {
    require(msg.sender == owner);
    _;
}

function destroySmartContract() public onlyOwner {
    selfdestruct(owner);
}
```

Example: Functions & Modifiers

Create a smart contract that contains the following functionalities:

- Create 2 modifiers: One that provides access only to the contract owner and one that allows the execution of a function only after a specific time in the future.
- Add a constructor that initializes the contract owner and a **timeLimit** variable to 1 minute after the deployment of the contract .
- Create a function that changes the value of a state variable (**value**) and that can only be executed by the contract owner and earliest 1 minute after deployment of the contract.
- Create a function that takes a **uint** as argument and returns the product of the argument and the "**value**" state variable.
- Create a function that takes **2 uint's** as arguments and returns the product and the sum of those arguments.



Exercise

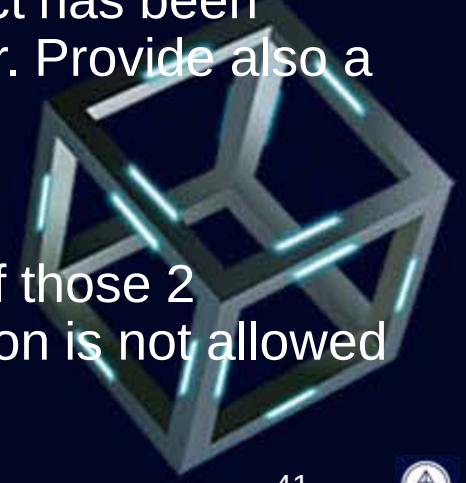
Create a smart contract that contains the following functionalities:

- Create a state variable of type string with the name "**myString**". Anyone should be allowed to read the value of "**myString**"

- Create a function that allows to modify the value of "**myString**". Only the contract owner is allowed to call that function and it should only be possible to call that function within the first 2 minutes after the contract has been deployed.

- If the contract owner tries to call the function 2 minutes (or later) after the contract has been deployed, abort the transaction and revert all state changes using a custom error. Provide also a detailed description of the error using **NatSpec** comments.

- Create a function that takes 2 arguments of type **uint** and that returns the sum of those 2 arguments. The function must be callable internally and externally and the function is not allowed to read from state.

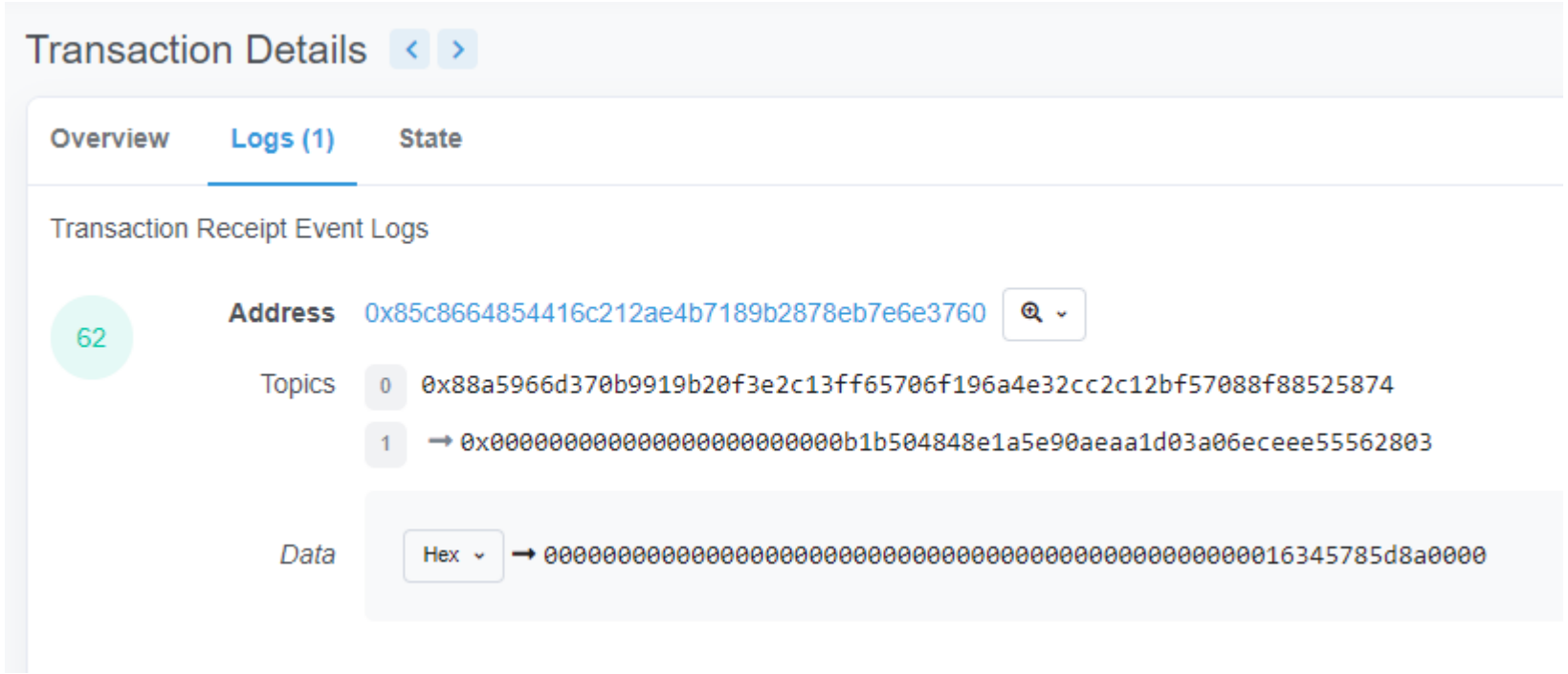


Events

- State changing functions cannot return values (they return a txn hash) - that's why we use events
- Events allow to log specific information - they are also used as cheap data storage (storing data on the blockchain is very expensive)
- Event arguments are stored in **the transaction log** - a special data structure on the blockchain
- Those logs are associated with the address of the contract
- Log data cannot be accessed from within the contract
- Client applications can subscribe and listen to events (read the log data) and react accordingly
- Up to 3 parameters can be indexed and searched for later - they get added to a special data structure called **topics**

Events Example

```
event Received(address indexed from, uint amount);

receive() external payable {
    emit Received(msg.sender, msg.value);
}
```



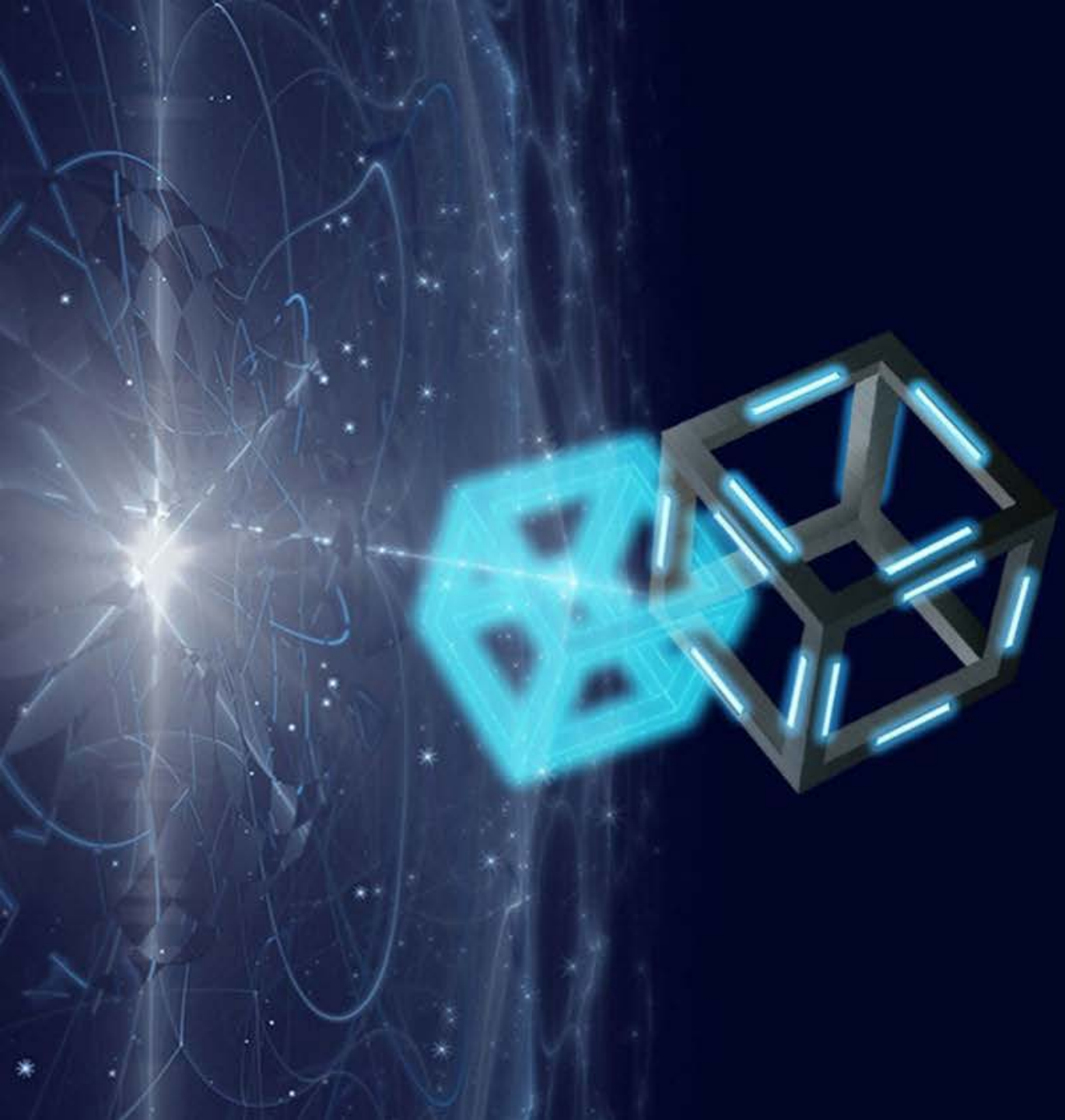
The “from" address is indexed and therefore listed as **Topic** - together with topic0 for the event signature

https://goerli.etherscan.io/tx/0x62c667f1cf3128756d115b31a1a8782b59a33c8333380f090ac92a71bca21064

Example: Events

Create a smart contract that contains the following functionalities:

- Create a private state variable that can store the balances of thousands of addresses
- Create an event (**Deposit**) that will be emitted whenever a user deposits any funds. The event should log the user address, which should be searchable and the deposited amount.
- Create a function (**deposit**) that allows users to deposit funds. Revert the functions if no funds are deposited. Update the user balance and emit the "**Deposit**" event.



Inheritance, Libraries, Interfaces and Abstract Contracts

Importing Files

- Re-use existing smart contracts and libraries, organize code, make code easier to maintain
- **import "filename"** => all global members of the specified file are imported into the current global scope - this pollutes the global namespace and is not recommended
- **import * as someNamespace from "filename"** => all global members are accessible via the provided namespace
- **import {function1 as alias, function2} from "filename"** => import only required members - an alias can be provided

Inheritance

- Multiple inheritance is supported
- Use "**is**" to derive from one or more contracts: **contract C is A, B** =>
  B is the most derived contract
- A derived contract can access all non-private members
- Use the **virtual** keyword on a function to allow overriding
- Use the **override** keyword to override a virtual function from a derived contract
- A function in a lesser derived contract can be called by specifying the contract:
  ContractName.functionName()
- To call the function one level higher up in the inheritance hierarchy, use: **super. functionName()**

Inheritance Example

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract TestSolidity is ERC20 {

    constructor() payable ERC20("name", "symbol") {
        _mint(msg.sender, 1000 * 10**decimals());
    }

    // overide function from ERC20
    function name() public view virtual override returns (string memory) {
        return "TEST";
    }
}
```

Example: Inheritance

Create a smart contract that contains the following functionalities:

- Create a contract **C2** that inherits from the existing contract **C1** and overrides the function **f3**
- Create a contract **C3** that defines a function **f2** with the same signature as the one defined in **C1**
- Create a contract **C4** that inherits from **C2 and C3** (in that order)
- Add a function (**callF3**) to **C4** that returns the result of **f3** – what is the return value and why?
- Add a function (**callSuperF3**) to **C4** that returns the result of the call to: **super.f3()** – what is the return value and why?

Libraries

- Defined by "**library**" keyword
- Provide re-usable set of functions - extend functionality of smart contract
- Cannot have state variables
- Cannot receive Ether
- Code is executed in the context of the calling contract - storage from calling contract can be accessed (only if state variables are explicitly supplied)
- The directive: ***using libraryName for someType;*** attaches the library functions to someType.
- First parameter of library function is often called "**self**" if the function can be seen as a method of that type

Library Example

```
library Search {
    // usage: uint[] data = [1,2,3]; => data.indexOf(3);
    function indexOf(uint[] memory self, uint value) internal pure returns(uint) {
        for(uint i = 0; i < self.length; i++)
            if(self[i] == value) return i;

        return type(uint).max;
    }
}

contract LibTest {
    using Search for uint[];
    uint[] data;

    function ReplceOrAddValue(uint oldValue, uint newValue) public {
        uint index = data.indexOf(oldValue);
        if(index == type(uint).max)
            data.push(newValue);
        else
            data[index] = newValue;
    }
```

Interfaces

- Defined by "**interface**" keyword
- Cannot have any functions implemented
- All functions must be external - they are virtual by default
- Cannot have state variables, constructor and modifier
- Can inherit from other interfaces
- A contract implements an interface by using the "**is**" keyword:
  contract SomeContract is SomeInterface

Interface Example

```
interface IERC20 {
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);

    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol

Abstract Contracts

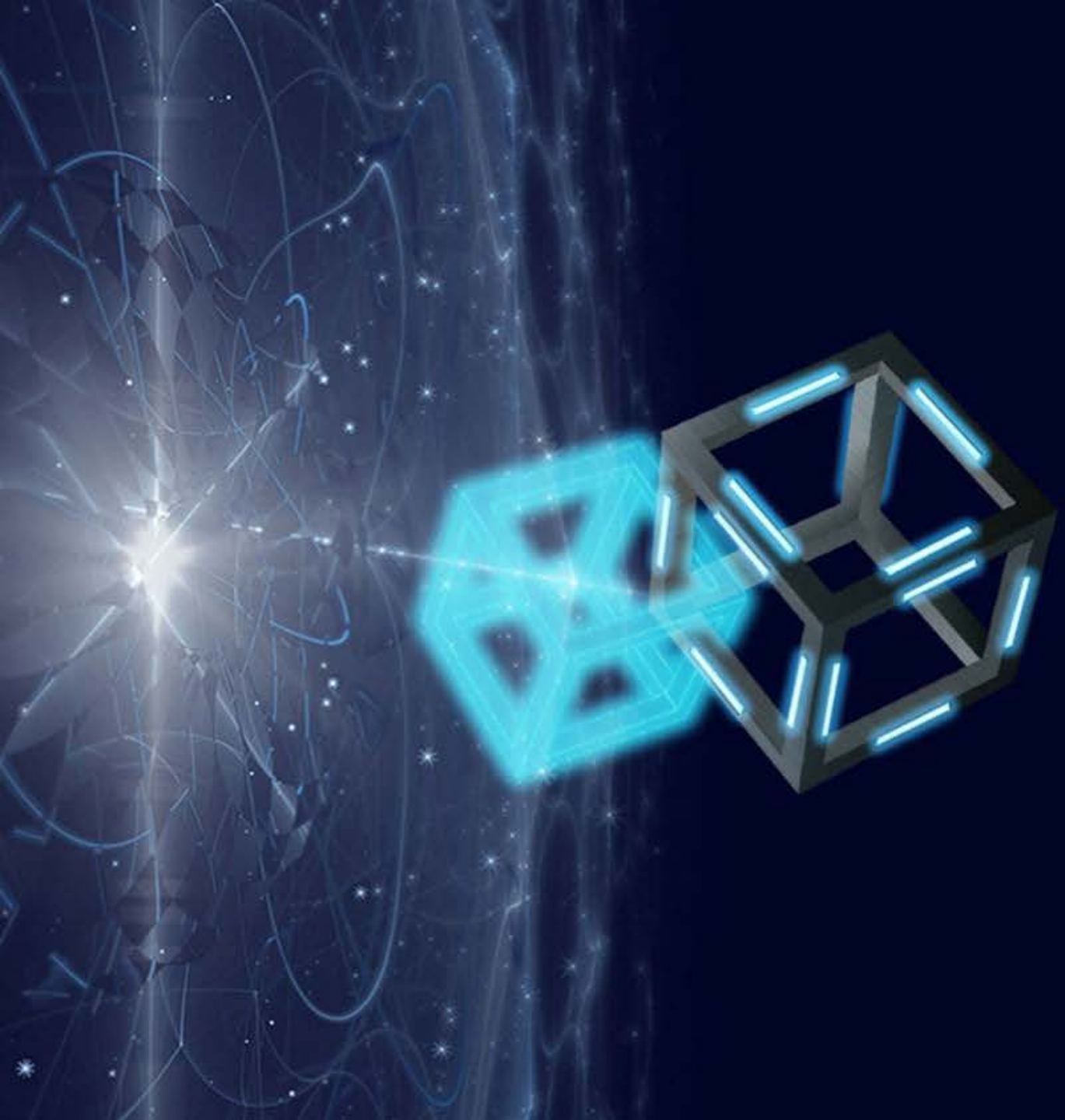
- Defined by "abstract" keyword
- A contract needs to marked as abstract when one or more functions are not implemented
- A contract can be marked as abstract even if all functions are implemented
- An abstract contract cannot be instantiated directly
- Abstract contracts are used as a base class
- A contract inherits from an abstract contract by using the "is" keyword:
  contract SomeContract is MyAbstractContract

Abstract Contracts Example

```
contract ERC20 is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;

**********************************************************************************

abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Context.sol



Voting Smart Contract

Voting Smart Contract

Requirements:

- During contract creation, all proposals are created and the end of the election is defined
- Each proposal has an Id and holds the number of votes received
- The contract has an owner (address that creates the contract) who can give other users the right to vote
- Each user (who has the right to vote) can vote only once and only before the election has ended
- After each vote, an event is emitted with the address of the voter and the voted proposal
- At any time, anyone can check the currently leading proposal and its current number of votes

Voting Smart Contract

Exercise:

- Add an ***endElection*** function that can only be called by the contract owner and after the election has ended

- Revert with a custom error of type: ***EndElectionAlreadyCalled*** if the ***endElection*** function is called more than once

- Emit an ***ElectionEnded*** event when the ***endElection*** function is called and provide the name of the winning proposal and the vote count as event arguments