Технология блокчейн
#8

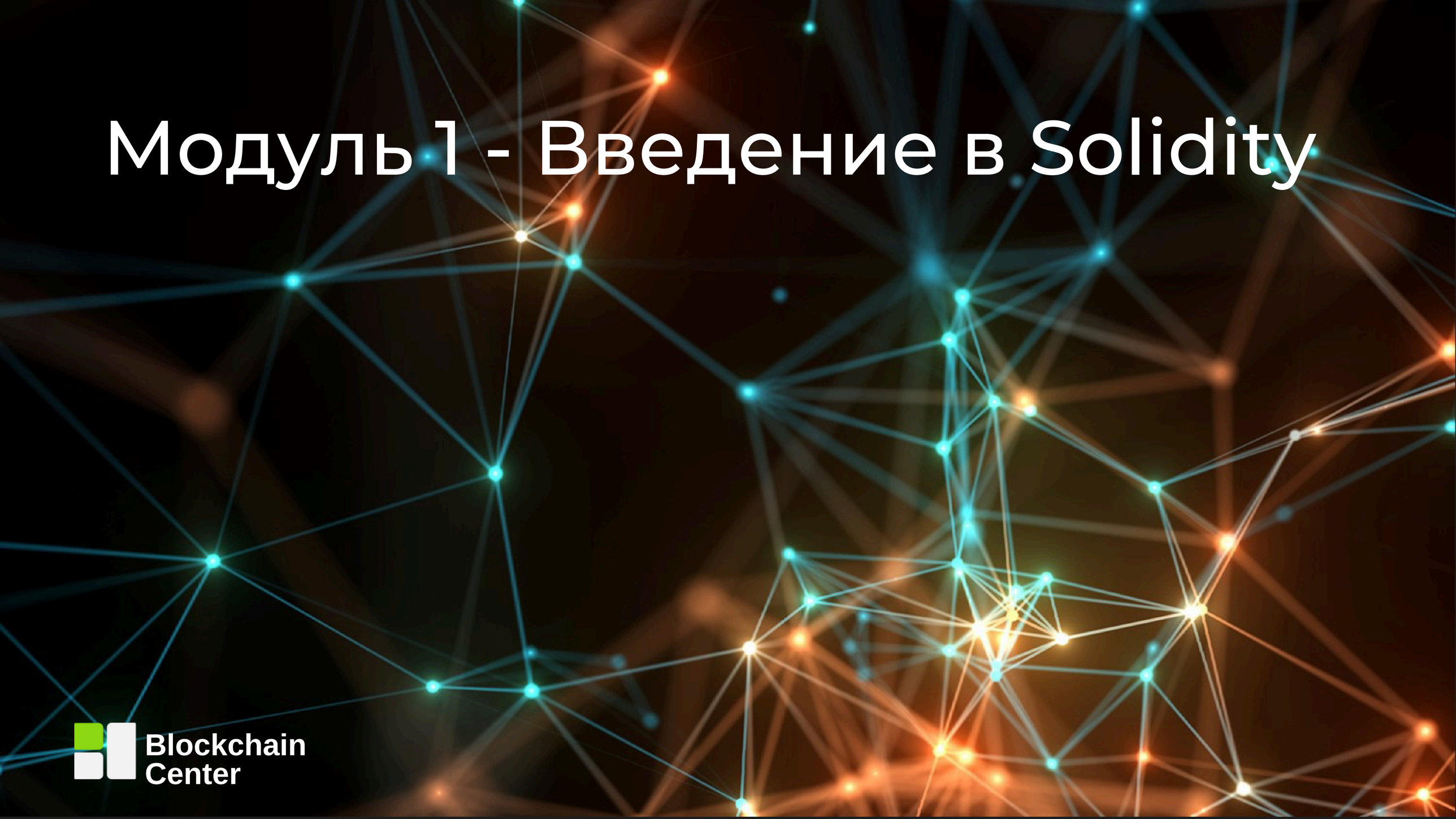
Введение в Solidity

Blockchain Center

Обзор

- Введение в Solidity
- Синтаксис Solidity
- Разработка смарт-контрактов

Модуль 1 - Введение в Solidity



Blockchain Center

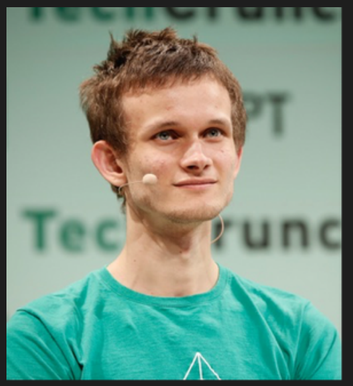
Что такое Solidity?

Solidity - это **объектно-ориентированный язык программирования** для написания смарт-контрактов на блокчейнах, таких как, наиболее известный, **Ethereum**. По своей сути Solidity - это язык программирования, на который оказали сильное влияние JavaScript, Python и C++. Solidity ориентирован на виртуальную машину Ethereum (EVM).

Blockchain Center

Происхождение solidity

Виталик Бутерин, один из **первых сторонников Биткойна**, впервые предложил создать другой протокол для выполнения более сложных смарт-контрактов, чем тот, который используется в сети Биткойн. Структура Биткойна проста, и не зря - он был создан в первую очередь для того, чтобы быть надежными деньгами. Однако для проведения более сложных транзакций, не требующих присутствия посредника для их подтверждения, Бутерин предложил идею Ethereum - сложной системы смарт-контрактов, взаимодействующих друг с другом.



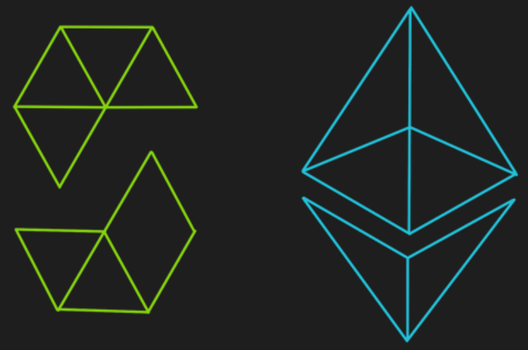
Vitalik Buterin

Gavin Wood

Blockchain Center

Понимание Ethereum и Solidity

Ethereum основан на обширной истории самоотверженной работы известных криптографов, математиков и компьютерных ученых. Более того, он о**бъединяет множество концепций**, начиная от **децентрализации**, неизменяемости, состояний, сетей, теории игр и так далее. Также важно упоминуть о машинах состояний. На пути к расшифровке того, как изучать Solidity, вы столкнетесь с такими терминами, как полнота по **Тьюрингу** и **машины Тьюринга.**



Blockchain Center

Полнота по Тьюрингу

Сам термин "полнота Тьюринга" намекает на происхождение концепции машины Тьюринга. Как вы уже догадались, к этому имел отношение легендарный Алан Тьюринг. В своей работе **1936 года "О вычислимых числах, с приложением к проблеме Энтшейдунга"** Тьюринг представил себе машину, **способную решать любые задачи**, которые **способен решать человек**.

Система правил манипулирования данными считается полной по Тьюрингу, если она может имитировать машину Тьюринга. Идея полноты Тьюринга предшествовала современным компьютерам и появилась до изобретения оперативной памяти.

Blockchain Center

Ethereum и EVM

Ethereum выходит за рамки проверки и хранения данных в распределенных бухгалтерских книгах. С помощью Ethereum можно о**существлять множество различных транзакций** и **высокоуровневых функций**, используя несколько валют или **токенов** (включая Bitcoin) в рамках всей сети. Эта сеть гарантирует, что код выполняется эквивалентно, а результирующие состояния **записываются** и **подтверждаются** посредством консенсуса.

EVM - это **виртуальный стек**, встроенный в каждый полностью участвующий узел сети, или узел Ethereum, который **выполняет байткод контракта**. EVM является **полной системой Тьюринга**, что означает, что он может выполнять **любые логические действия**, связанные с **вычислительными функциями**.

Blockchain Center

Роль Solidity в EVM

Solidity - это **высокоуровневый язык программирования**, который совместим с тем, как люди выражают инструкции - **используя цифры** и **буквы** вместо **двоичного кода**. Если ранние **машины Тьюринга** были основаны на **вводе единиц** и **нулей**, то Solidity **избавляет от такой сложности** и **гуманизирует процесс** ввода с помощью более дружественного кода, который во многом похож на JavaScript.

Смарт-контракты Solidity представляют собой **инструкции**, которые затем **компилируются в байткод EVM**. Узлы в сети Ethereum, как уже упоминалось, запускают **экземпляры EVM**, которые позволяют им **договориться** о выполнении **одного и того же набора инструкций**.

Blockchain Center

Роль Solidity в EVM

Solidity - это **высокоуровневый язык программирования**, который совместим с тем, как люди выражают инструкции - **используя цифры** и **буквы** вместо **двоичного кода**. Если ранние **машины Тьюринга** были основаны на **вводе единиц** и **нулей**, то Solidity **избавляет от такой сложности** и **гуманизирует процесс** ввода с помощью более дружественного кода, который во многом похож на JavaScript.

Смарт-контракты Solidity представляют собой **инструкции**, которые затем **компилируются в байткод EVM**. Узлы в сети Ethereum, как уже упоминалось, запускают **экземпляры EVM**, которые позволяют им **договориться** о выполнении **одного и того же набора инструкций**.

Blockchain Center

История Solidity

Впервые Solidity была предложена доктором **Гэвином Вудом**, первым техническим директором Ethereum, в **2014 году** и доработана **Кристианом Райтвисснером**, который возглавил команду разработчиков для ее продвижения. Ее основная команда до сегодняшнего дня **спонсируется** Ethereum Foundation, а также имеет **широкое сообщество сторонников** и **длинный список** участников, которые продолжают **помогать в ее развитии путем** коммитов, подачи сообщений **об ошибках** и **активного решения проблем** на Github и за его пределами.
Его первой публично названной версией была v0.1.0. С тех пор она претерпела **множество улучшений** и **итераций**.

Christian Reitwiessner

Blockchain Center

Solidity - первый контрактно-ориентированный язык

Контрактно-ориентированный язык отличается от таких в основном объектно-ориентированных языков, как Java и C++, тем, что **акцент** в нем делается на **контрактах** и **функциях**. Solidity типизирован статически. Он также **поддерживает библиотеки**, **наследование** и **другие определяемые пользователем функции**, которые, как правило, **более сложные**. Язык компилирует все инструкции в песочницу байткода, чтобы эти инструкции могли быть прочитаны и интерпретированы в сети Ethereum.

Blockchain Center

Модуль 2 - Основной Синтаксис Solidity

Blockchain Center

Пример простого смарт-контракта

uint public count; // является переменным состоянием и хранится в блокчейне

Функции inc() и dec(); // изметяют состояние count, каждый вызов функции требует газа

Расчет газа:
https://www.evm.codes/

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

    // Function to increment count by 1
    function inc() public {
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {
        // This function will fail if count = 0
        count -= 1;
    }
}
```

Blockchain Center

Переменные

В Solidity существует 3 типа переменных:

локальные
объявлены внутри функции
не хранятся в блокчейне

состояния
объявленные вне функции
хранится в блокчейне

глобальные (предоставляет информацию о блокчейне)

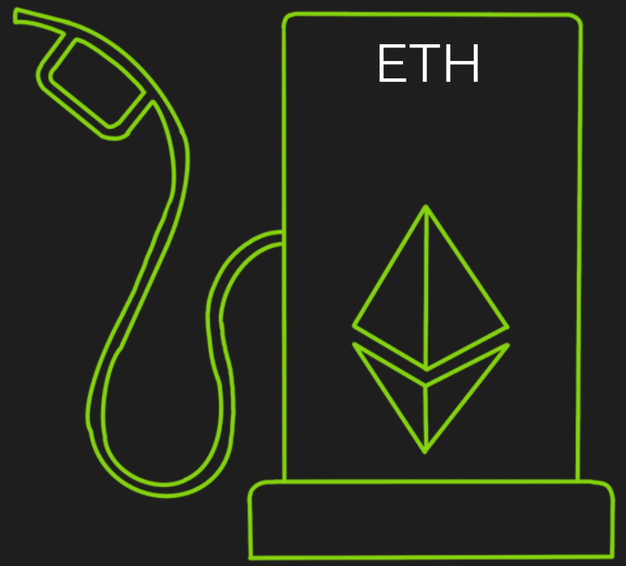
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint public num = 123;

    function doSomething() public {
        // Local variables are not saved to the blockchain.
        uint i = 456;

        // Here are some global variables
        uint timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // address of the caller
    }
}
```

Blockchain Center

Газ

Сколько эфира вам нужно заплатить за транзакцию?
Вы платите за потраченный газ * стоимость газа количество эфира, где

газ - единица вычисления
потраченный газ - общее количество газа, использованного в транзакции
цена газа - сколько эфира вы готовы заплатить за газ.
Транзакции с более высокой ценой газа имеют более высокий приоритет на включение в блок.
Неизрасходованный газ будет возвращен.



Blockchain Center

Лимит газа

Существует 2 верхние границы количества газа, которое вы можете потратить
лимит газа (максимальное количество газа, которое вы готовы потратить на транзакцию, устанавливается вами)
лимит газа в блоке (максимальное количество газа, разрешенное в блоке, устанавливается сетью)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Gas {
    uint public i = 0;

    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

Blockchain Center

Mapping

Map создаются с помощью синтаксиса mapping(keyType => valueType).

KeyType может быть любым встроенным типом значения, байтом, строкой или любым контрактом.

valueType может быть любым типом, включая другое отображение или массив.

Mapping не являются итерируемыми.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public myMap;

    function get(address _addr) public view returns (uint) {
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return myMap[_addr];
    }

    function set(address _addr, uint _i) public {
        // Update the value at this address
        myMap[_addr] = _i;
    }

    function remove(address _addr) public {
        // Reset the value to the default value.
        delete myMap[_addr];
    }
}
```

Nested Mapping

```
contract NestedMapping {
    // Nested mapping (mapping from address to another mapping)
    mapping(address => mapping(uint => bool)) public nested;

    function get(address _addr1, uint _i) public view returns (bool) {
        // You can get values from a nested mapping
        // even when it is not initialized
        return nested[_addr1][_i];
    }

    function set(address _addr1, uint _i, bool _boo) public {
        nested[_addr1][_i] = _boo;
    }

    function remove(address _addr1, uint _i) public {
        delete nested[_addr1][_i];
    }
}
```

Blockchain Center

Array (массив) 1

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Array {
    // Several ways to initialize an array
    uint[] public arr;
    uint[] public arr2 = [1, 2, 3];
    // Fixed sized array, all elements initialize to 0
    uint[10] public myFixedSizeArr;

    function get(uint i) public view returns (uint) {
        return arr[i];
    }

    // Solidity can return the entire array.
    // But this function should be avoided for
    // arrays that can grow indefinitely in length.
    function getArr() public view returns (uint[] memory) {
        return arr;
    }

    function push(uint i) public {
        // Append to array
        // This will increase the array length by 1.
        arr.push(i);
    }
```

Blockchain Center

Array (массив) 2

```
    function pop() public {
        // Remove last element from array
        // This will decrease the array length by 1
        arr.pop();
    }

    function getLength() public view returns (uint) {
        return arr.length;
    }

    function remove(uint index) public {
        // Delete does not change the array length.
        // It resets the value at index to it's default value,
        // in this case 0
        delete arr[index];
    }

    function examples() external {
        // create array in memory, only fixed size can be created
        uint[] memory a = new uint[](5);
    }
}
```

Blockchain Center

Enum

Solidity supports enumerables and they are useful to model choice and keep track of state.

Enums can be declared outside of a contract.

Blockchain Center

```
contract Enum {
    // Enum representing shipping status
    enum Status {
        Pending,
        Shipped,
        Accepted,
        Rejected,
        Canceled
    }

    // Default value is the first element listed in
    // definition of the type, in this case "Pending"
    Status public status;

    // Returns uint
    // Pending  - 0
    // Shipped  - 1
    // Accepted - 2
    // Rejected - 3
    // Canceled - 4
    function get() public view returns (Status) {
        return status;
    }

    // Update status by passing uint into input
    function set(Status _status) public {
        status = _status;
    }

    // You can update to a specific enum like this
    function cancel() public {
        status = Status.Canceled;
    }

    // delete resets the enum to its first value, 0
    function reset() public {
        delete status;
    }
}
```

Structs 1

You can define your own type by creating a struct.

They are useful for grouping together related data.

Structs can be declared outside of a contract and imported in another contract.

```
contract Todos {
    struct Todo {
        string text;
        bool completed;
    }

    // An array of 'Todo' structs
    Todo[] public todos;

    function create(string calldata _text) public {
        // 3 ways to initialize a struct
        // - calling it like a function
        todos.push(Todo(_text, false));

        // key value mapping
        todos.push(Todo({text: _text, completed: false}));

        // initialize an empty struct and then update it
        Todo memory todo;
        todo.text = _text;
        // todo.completed initialized to false

        todos.push(todo);
    }
```

Blockchain Center

Structs 2

```
// Solidity automatically created a getter for 'todos' so
// you don't actually need this function.
function get(uint _index) public view returns (string memory text, bool completed) {
    Todo storage todo = todos[_index];
    return (todo.text, todo.completed);
}

// update text
function updateText(uint _index, string calldata _text) public {
    Todo storage todo = todos[_index];
    todo.text = _text;
}

// update completed
function toggleCompleted(uint _index) public {
    Todo storage todo = todos[_index];
    todo.completed = !todo.completed;
}
```

Blockchain Center

Data Locations - Storage, Memory and Calldata

Variables are declared as either storage, memory or calldata to explicitly specify the location of the data.
storage - variable is a state variable (store on blockchain)
memory - variable is in memory and it exists while a function is being called
calldata - special data location that contains function arguments

```
contract DataLocations {
    uint[] public arr;
    mapping(uint => address) map;
    struct MyStruct {
        uint foo;
    }
    mapping(uint => MyStruct) myStructs;

    function f() public {
        // call _f with state variables
        _f(arr, map, myStructs[1]);

        // get a struct from a mapping
        MyStruct storage myStruct = myStructs[1];
        // create a struct in memory
        MyStruct memory myMemStruct = MyStruct(0);
    }
```

Blockchain Center

Data Locations - Storage, Memory and Calldata

Variables are declared as either storage, memory or calldata to explicitly specify the location of the data.
storage - variable is a state variable (store on blockchain)
memory - variable is in memory and it exists while a function is being called
calldata - special data location that contains function arguments

```
function _f(
    uint[] storage _arr,
    mapping(uint => address) storage _map,
    MyStruct storage _myStruct
) internal {
    // do something with storage variables
}

// You can return memory variables
function g(uint[] memory _arr) public returns (uint[] memory) {
    // do something with memory array
}

function h(uint[] calldata _arr) external {
    // do something with calldata array
}
```

Blockchain Center

Functions 1

There are several ways to return outputs from a function.
Public functions cannot accept certain data types as inputs or outputs

```
contract Function {
    // Functions can return multiple values.
    function returnMany() public pure returns (uint, bool, uint) {
        return (1, true, 2);
    }

    // Return values can be named.
    function named() public pure returns (uint x, bool b, uint y) {
        return (1, true, 2);
    }

    // Return values can be assigned to their name.
    // In this case the return statement can be omitted.
    function assigned() public pure returns (uint x, bool b, uint y) {
        x = 1;
        b = true;
        y = 2;
    }
```

Blockchain
Center

Functions 2

```
// Use destructuring assignment when calling another
// function that returns multiple values.
function destructuringAssignments()
    public
    pure
    returns (uint, bool, uint, uint, uint)
{
    (uint i, bool b, uint j) = returnMany();

    // Values can be left out.
    (uint x, , uint y) = (4, 5, 6);

    return (i, b, j, x, y);
}

// Cannot use map for either input or output

// Can use array for input
function arrayInput(uint[] memory _arr) public {}

// Can use array for output
uint[] public arr;

function arrayOutput() public view returns (uint[] memory) {
    return arr;
}
```

Blockchain Center

Functions 3

```
// Call function with key-value inputs
contract XYZ {
    function someFuncWithManyInputs(
        uint x,
        uint y,
        uint z,
        address a,
        bool b,
        string memory c
    ) public pure returns (uint) {}

    function callFunc() external pure returns (uint) {
        return someFuncWithManyInputs(1, 2, 3, address(0), true, "c");
    }

    function callFuncWithKeyValue() external pure returns (uint) {
        return
            someFuncWithManyInputs({a: address(0), b: true, c: "c", x: 1, y: 2, z: 3});
    }
}
```

View and Pure Functions

Getter functions can be declared view or pure.
View function declares that no state will be changed.
Pure function declares that no state variable will be changed or read.

```
contract ViewAndPure {
    uint public x = 1;

    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }

    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }
}
```

Blockchain
Center

Error Handling

An error will undo all changes made to the state during a transaction.
You can throw an error by calling require, revert or assert.

- require is used to validate inputs and conditions before execution.
- revert is similar to require. See the code below for details.
- assert is used to check for code that should never be false. Failing assertion probably means that there is a bug.

```
contract Error {
    function testRequire(uint _i) public pure {
        // Require should be used to validate conditions such as:
        // - inputs
        // - conditions before execution
        // - return values from calls to other functions
        require(_i > 10, "Input must be greater than 10");
    }

    function testRevert(uint _i) public pure {
        // Revert is useful when the condition to check is complex.
        // This code does the exact same thing as the example above
        if (_i <= 10) {
            revert("Input must be greater than 10");
        }
    }
```

Blockchain Center

Error Handling 2

```
uint public num;

function testAssert() public view {
    // Assert should only be used to test for internal errors,
    // and to check invariants.

    // Here we assert that num is always equal to 0
    // since it is impossible to update the value of num
    assert(num == 0);
}

// custom error
error InsufficientBalance(uint balance, uint withdrawAmount);

function testCustomError(uint _withdrawAmount) public view {
    uint bal = address(this).balance;
    if (bal < _withdrawAmount) {
        revert InsufficientBalance({balance: bal, withdrawAmount: _withdrawAmount});
    }
}
```

Blockchain Center

Error Handling 3

```sol
contract Account {
    uint public balance;
    uint public constant MAX_UINT = 2 ** 256 - 1;

    function deposit(uint _amount) public {
        uint oldBalance = balance;
        uint newBalance = balance + _amount;

        // balance + _amount does not overflow if balance + _amount >= balance
        require(newBalance >= oldBalance, "Overflow");

        balance = newBalance;

        assert(balance >= oldBalance);
    }

    function withdraw(uint _amount) public {
        uint oldBalance = balance;

        // balance - _amount does not underflow if balance >= _amount
        require(balance >= _amount, "Underflow");

        if (balance < _amount) {
            revert("Underflow");
        }

        balance -= _amount;

        assert(balance <= oldBalance);
    }
}
```

Blockchain Center

Events

Events allow logging to the Ethereum blockchain. Some use cases for events are:

Listening for events and updating user interface
A cheap form of storage

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Event {
    // Event declaration
    // Up to 3 parameters can be indexed.
    // Indexed parameters helps you filter the logs by the indexed parameter
    event Log(address indexed sender, string message);
    event AnotherLog();

    function test() public {
        emit Log(msg.sender, "Hello World!");
        emit Log(msg.sender, "Hello EVM!");
        emit AnotherLog();
    }
}
```

Blockchain Center

Практика

Примеры solidity:
https://solidity-by-example.org/

Практика по написанию смарт-контрактов

Blockchain
Center