

# Основы разработки на Solidity

Основы разработки на Solidity



# Оглавление

Введение	3
<b>Типы данных</b>	<b>3</b>
Пример:	3
Модель памяти	4
Пример:	5
Типы данных	6
Практическая задача: Разработка Умного Хранилища данных	8
Value types	8
Reference types	9
Массивы (Arrays)	10
<b>Структуры (Structs)</b>	<b>10</b>
<b>Отображения (Mappings)</b>	<b>11</b>
Практическая Задача: Создание Системы Управления Учебным Центром	12
<b>Структура контракта</b>	<b>13</b>
Функции	15
Аналогия:	17
Практическое Применение:	17
Модификаторы	17
Ошибки	18
События	18
<b>Обработка ошибок</b>	<b>18</b>
<b>Использованная литература</b>	<b>19</b>

# Введение

На прошлом уроке мы узнали, как смарт-контракты обрабатываются виртуальной машиной Ethereum и ознакомились с основами синтаксиса Solidity. Цель сегодняшнего урока:

- освоить синтаксис solidity, научиться читать код контрактов;
- подробно изучить среду выполнения контрактов;
- научиться размещать, вызывать и отлаживать контракты.

На лекции мы изучим способы хранения переменных, структуру контрактов и в процессе напишем несложный контракт. Вперёд к знаниям!

## Типы данных

Solidity это язык со статической строгой типизацией, это значит, что на момент компиляции программы типы всех переменных должны быть явным образом заданы и существуют ограничения на приведение типов переменных.

Ключевое отличие Solidity от других языков как, например, JavaScript – это отсутствие концепции null или undefined. В Solidity каждая переменная инициализируется пустым значением, которое зависит от её типа. Это означает, что переменные всегда имеют некое начальное значение, даже если оно не было явно задано.

Можно провести аналогию с игрой в карты, где каждая карта (переменная) должна быть из определённой колоды (типа данных). Solidity не позволит вам играть картой, которой нет в колоде, в отличие от языков, где такие невидимые или неопределённые карты допустимы.

### Пример:

Давайте рассмотрим пример. Если вы объявите переменную типа uint, она автоматически инициализируется значением 0. В других языках, таких как JavaScript, неинициализированная переменная будет иметь значение undefined или

null.

```
1 uint myNumber; // автоматически инициализируется как 0 в Solidity
```



Используйте `require` для проверки входных данных функции. Это упростит отслеживание ошибок и убережёт вас от множества проблем при работе со сложными типами данных.

## Модель памяти

### Модель памяти в Solidity и Ethereum Virtual Machine (EVM)

В Solidity область хранения данных разделена на три ключевые зоны: *стек (stack)*, *память (memory)* и *состояние EVM (state)*. Каждая из этих зон имеет уникальные особенности:

- **Стек** — это основное место для хранения временных переменных. Он работает по принципу LIFO (последний пришёл — первый ушёл).
- **Память** — используется для хранения данных во время исполнения контракта. Это похоже на оперативную память в компьютере.
- **Состояние EVM** — это долгосрочное хранилище данных, которые сохраняются между транзакциями и блоками.

Можно сравнить *стек*, *память* и *состояние EVM* с разными формами хранения на кухне. *Стек* — это рабочая поверхность, где вы временно размещаете ингредиенты (переменные), *память* — это открытые полки, где вы держите ингредиенты во время приготовления блюда (время выполнения контракта), а *состояние EVM* — это холодильник, где хранятся продукты (данные) на долгий срок.

В глобальном смысле для переменных есть всего три области хранения — стек (*stack*), память (*memory*) и состояние EVM (*state*). Размер машинного слова EVM составляет 32 байта (256 бит). EVM построена по гарвардской архитектуре — код и данные размещены в разных адресных пространствах и код не может быть модифицирован в рамках выполнения. Единственный способ исполнить код из переменных — программно разместить новый контракт и вызвать его.

EVM является стековой машиной, значительная часть переменных хранится в стеке. Формально размер стека не ограничен, однако, поскольку каждая операция `push` сжигает 3 газа, то фактически он ограничен размером газа на блок (30 000 000 единиц) - 10 000 000 значений.

Кроме того, во время выполнения, каждому контракту выделяется независимое адресное пространство с произвольным доступом. Запись значений осуществляется командой `mstore`, которая также стоит 3 газа, поэтому фактический размер памяти ограничен также 10 000 000 значений.

Самым дорогим по газу местом для хранения является EVM state – одна операция записи командой `sstore` может стоить до 20 тыс. единиц газа, что в теории даёт возможность записи 1 500 значений. Стоимость выполнения команды `sstore` варьируется в зависимости от следующих факторов: записывается нулевое или ненулевое значение, изменяется ли фактическое хранимое значение, использовался ли адрес ранее во время текущего выполнения контракта. Самым дорогим вариантом является запись нового ненулевого значения - 20 тыс. газа. При освобождении ячейки EVM state газ, напротив, освобождается.

## Пример:

Представьте, что вы записываете значение в *состояние EVM* (через операцию `sstore`). Это похоже на сохранение файла на жёсткий диск компьютера. Данная операция требует значительного количества газа, особенно если вы сохраняете новое ненулевое значение.

```
1 uint public data;  
2  
3  
4 function updateData(uint _data) public {  
5     data = _data; // Здесь используется операция sstore  
6 }  
7
```

## Типы данных

В Solidity выделяются два вида типов данных – типы, передающиеся по значению (value types) и по ссылке (reference types). Как следует из названия, одни переменные всегда передаются только по значению, т. е. при использовании

переменной в качестве параметра функции, создаётся копия. Вторые, напротив, только по ссылке.

Простые задачи и примеры решений:

### Задание 1:

Создайте смарт-контракт, который хранит числовое значение и позволяет его изменять.

Добавьте функцию для чтения этого значения.

Убедитесь, что типы данных и доступ к переменным корректно объявлены.

```
1 pragma solidity ^0.8.7;
2
3
4 contract SimpleStorage {
5     uint private storedData;
6
7     function set(uint x) public {
8         storedData = x;
9     }
10
11
12     function get() public view returns (uint) {
13         return storedData;
14     }
15 }
16 }
17
```

### Задание 2:

Создайте смарт-контракт с закрытой переменной и функцией для её изменения.

Добавьте модификатор, который разрешает изменение переменной только владельцем контракта.

Проверьте работоспособность контракта с разными аккаунтами.

```

1 pragma solidity ^0.8.7;
2
3 contract Ownable {
4     address public owner;
5
6     constructor() {
7         owner = msg.sender;
8     }
9
10    modifier onlyOwner() {
11        require(msg.sender == owner, "Not owner");
12        _;
13    }
14
15    function changeOwner(address newOwner) public onlyOwner {
16        owner = newOwner;
17    }
18 }
19

```

### Задание 3:

Создайте контракт для хранения массива чисел.

Реализуйте функцию добавления элемента в массив с учётом оптимизации газа.

Добавьте функцию для просмотра всего массива.

```

1 pragma solidity ^0.8.7;
2
3 contract GasOptimizedStorage {
4     uint[] public numbers;
5
6     function addNumber(uint number) public {
7         numbers.push(number);
8     }
9
10    function getNumbers() public view returns (uint[] memory) {
11        return numbers;
12    }
13 }
14

```

## Практическая задача: разработка умного хранилища данных

### Задание:

Создать смарт-контракт под названием SmartDataStorage, который будет использоваться для хранения, обновления и извлечения данных различных типов.

Контракт должен включать следующие функции:

- `storeNumber(uint number)`: добавляет числовое значение в массив. Должна учитывать оптимизацию газа.
- `retrieveNumbers()`: возвращает массив всех сохранённых чисел.
- `storeString(string memory text)`: хранит строку. Доступ к этой функции должен быть ограничен только владельцем контракта.
- `retrieveString()`: возвращает последнюю сохранённую строку.
- `changeOwnership(address newOwner)`: позволяет текущему владельцу передать права новому владельцу. Должна использовать модификатор доступа.

Управление доступом: только владелец контракта может добавлять или изменять строки.

Оптимизация газа: убедитесь, что операции с массивами и строками эффективно используют газ.

### Ответ:

Для выполнения данного задания нужно написать смарт-контракт на языке Solidity. Вот пример кода смарт-контракта SmartDataStorage, удовлетворяющего всем требованиям:



```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4
5 contract SmartDataStorage {
6     uint[] private numbers;
7     string private storedString;
8     address private owner;
9
10
11     constructor() {
12         owner = msg.sender;
13     }
14
15
16     function storeNumber(uint number) public {
17         numbers.push(number);
18     }
19
20
21     function retrieveNumbers() public view returns (uint[] memory) {
22         return numbers;
23     }
24
25
26     function storeString(string memory text) public {
27         require(msg.sender == owner, "Only contract owner can store
string");
28         storedString = text;
29     }
30
31
32     function retrieveString() public view returns (string memory) {
33         return storedString;
34     }
35
36
37     function changeOwnership(address newOwner) public {
38         require(msg.sender == owner, "Only contract owner can change
ownership");
39         owner = newOwner;
40     }
41 }
42

```

Этот код включает все необходимые функции и управление доступом. Функция storeNumber добавляет числовое значение в массив, retrieveNumbers возвращает


массив чисел, `storeString` хранит строку (только для владельца), `retrieveString` возвращает строку, а `changeOwnership` позволяет текущему владельцу передать права новому.

Использование операций с массивами и строками в этом контракте производится эффективно, что помогает оптимизировать использование газа.

## Value types

Рассмотрим подробнее некоторые из value types.

- Знаковое/беззнаковое целое число (signed/unsigned integer) – задаётся ключевыми словами **int8..int256** / **uint8..uint256**, соответственно, с шагом 8 бит. Обозначения **int** и **uint** являются псевдонимами для **int256** и **uint256**.
- Булевское значение (boolean) - задаётся ключевым словом **boolean**, хранит значение **true** / **false**.
- Адрес (address) – задаётся ключевым словом **address** либо **address payable**, содержит 20-байтовое значение адреса в сети ethereum, предоставляет функции такие как **balance()**, **transfer()**, **send()**, **call()** и т. д.
- Массив байт фиксированного размера (bytes) – задаётся ключевыми словами **bytes1 bytes32** и хранит последовательность байтов длиной от 1 до 32.

 В solidity нет встроенного типа данных, обеспечивающего работу с плавающей точкой (floating point). Для работы с дробными числами применяются либо вычисления с фиксированной точкой, либо механизм вычислений с сохранением отдельно числителя и знаменателя.

## Reference types

У каждой переменной ссылочного типа есть дополнительный атрибут, указывающий место размещения. У переменных есть три места размещения - **memory**, **storage** и **calldata**.

**Calldata** означает, что переменная будет доступна только для чтения, этот атрибут используется для параметров функций, вызываемых извне. При использовании переменной этого типа не произойдёт фактического копирования данных из транзакции.

**Memory** означает, что переменная будет размещена в памяти.

**Storage** означает, что переменная будет размещена в EVM state.

Существуют правила копирования данных:

- при присвоении значений между **storage** и **memory** будет создана копия данных;
- при присвоении между **memory** и **memory** произойдёт создание ссылки;
- все присвоения переменной **storage** осуществляются копированием данных.

## Массивы (Arrays)

Массивы в Solidity — это как шкафы с множеством ячеек для хранения элементов одного типа. Они бывают двух видов:

Статические массивы: похожи на шкафы с фиксированным количеством ячеек. Их размер известен заранее и не меняется.

Динамические массивы: это как шкафы с расширяемыми полками. Можно добавлять сколько угодно элементов, а массив "растет" для их размещения.

Пример:

```
1 ```solidity
2 uint[5] staticArray; // Статический массив из 5 элементов
3 uint[] dynamicArray; // Динамический массив, может расти
4 ```
5
```

Массивы хранят свои элементы последовательно, как книги на полке. Также можно создавать многомерные массивы, что похоже на использование нескольких шкафов одновременно.

## Структуры (Structs)

Структуры в Solidity позволяют создавать свои собственные сложные типы данных. Это как конструктор, из которого можно построить здание с разными комнатами (атрибутами). Каждая комната служит для определенной цели, но все они объединены в один объект.

Пример:

```
1  ```solidity
2  struct Person {
3      string name;
4      uint age;
5  }
6  Person person1; // Создание объекта структуры
7  ```
8
```

В структурах нельзя использовать рекурсию, что можно сравнить с запретом на строительство бесконечно высокого здания.

## Отображения (Mappings)

Отображения — это как секретные ящики, каждый из которых открывается уникальным ключом. Они предназначены для эффективного хранения и быстрого доступа к данным. Проведем аналогию: представьте себе большую библиотеку, где каждая книга имеет уникальный номер. Отображение позволяет вам мгновенно найти нужную книгу по ее номеру, вместо того чтобы искать ее по всей библиотеке.

Пример:

```
1 ```solidity
2 mapping(address => uint) public balances; // Отображение балансов
3 balances[msg.sender] = 100; // Присваивание значения по ключу
4 ```
5
```

В отображении, ключом может быть любой простой тип, а значением — любой тип данных, включая другие отображения и массивы.



Типы данных в Solidity должны быть выбраны осторожно и с учетом их использования в контракте. Неправильный выбор может привести к неэффективному использованию газа и даже к уязвимостям в безопасности.

## Практическая задача: создание системы управления учебным центром

Описание задачи:

Студентам предстоит разработать смарт-контракт для управления курсами и учетом студентов в учебном центре. Контракт должен обеспечивать добавление новых курсов, регистрацию студентов на курсы и отслеживание их успеваемости.

Спецификация:

Структура Course:

- Название курса (string).
- Преподаватель (адрес).
- Список студентов (массив адресов).

Структура Student:

- Имя студента (string).
- Список курсов, на которые студент записан (массив названий курсов).
- Оценки по курсам (отображение из названия курса в оценку).

Функционал контракта:

- `addCourse(string memory courseName, address teacher)`: добавление нового курса.
- `registerStudent(string memory studentName)`: регистрация нового студента.
- `enrollStudentToCourse(string memory studentName, string memory courseName)`: запись студента на курс.
- `assignGrade(string memory studentName, string memory courseName, uint grade)`: присвоение оценки студенту за курс.
- `getStudentCourses(string memory studentName)`: возвращает список курсов, на которые записан студент.
- `getCourseStudents(string memory courseName)`: возвращает список студентов, записанных на курс.

### **Ответ:**

Для данной задачи нужно создать смарт-контракт на языке Solidity, который будет управлять курсами и студентами в учебном центре. Ниже представлен пример такого контракта:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract EducationalCenter {
5     struct Course {
6         string courseName;
7         address teacher;
8         address[] students;
9         mapping(address => uint) grades;
10    }
11
12    struct Student {
13        string studentName;
14        string[] coursesEnrolled;
15        mapping(string => uint) grades;
16    }
17
18    mapping(string => Course) courses;
19    mapping(string => Student) students;
20
21    function addCourse(string memory courseName, address teacher)
22    public {
23        require(courses[courseName].teacher == address(0), "Course
24        already exists");
25        courses[courseName] = Course(courseName, teacher, new
26        address[](0));
27    }
28
29    function registerStudent(string memory studentName) public {
30        require(bytes(students[studentName].studentName).length ==
31        0, "Student already exists");
32        students[studentName] = Student(studentName, new string[]
33        (0));
34    }
35
36    function enrollStudentToCourse(string memory studentName, string
37    memory courseName) public {
38        require(bytes(students[studentName].studentName).length !=
39        0, "Student does not exist");
40        require(courses[courseName].teacher != address(0), "Course
41        does not exist");
42        students[studentName].coursesEnrolled.push(courseName);
43        courses[courseName].students.push(msg.sender);
44    }
45
46    function assignGrade(string memory studentName, string memory
47    courseName, uint grade) public {
48        require(courses[courseName].teacher == msg.sender, "Only
49        teacher can assign grades");
50        students[studentName].grades[courseName] = grade;
51        courses[courseName].grades[msg.sender] = grade;
52    }
53
54    function getStudentCourses(string memory studentName) public
55    view returns (string[] memory) {
56        return students[studentName].coursesEnrolled;
57    }
58
59    function getCourseStudents(string memory courseName) public view
60    returns (address[] memory) {
61        return courses[courseName].students;
62    }
63 }

```

Этот смарт-контракт содержит структуры `Course` и `Student` для хранения информации о курсах и студентах соответственно, а также функции для добавления курсов, регистрации студентов, записи студентов на курсы, присвоения оценок и получения списков курсов студента и студентов на курсе.

## Структура контракта в Solidity

Смарт-контракты в Solidity похожи на рецепты в кулинарии. Как и рецепт, который содержит ингредиенты и шаги приготовления, смарт-контракт содержит переменные и функции, которые определяют его поведение и взаимодействие.

### Переменные Состояния (State Variables)

Переменные состояния – это как ингредиенты в рецепте. Они сохраняются в блокчейне и доступны в любое время, точно так же, как ингредиенты хранятся на кухне.

Пример:

```
1  ```solidity
2  contract Recipe {
3      uint public sugarGrams; // Количество сахара в граммах
4  }
5  ```
6
```

Видимость переменных:

- **Internal:** это как ингредиенты, доступные только в вашей кухне. Они доступны внутри контракта и его наследниках.
- **Private:** это особые ингредиенты, которые вы держите в секрете даже от своих помощников. Только внутри контракта, где они объявлены.
- **Public:** публичные ингредиенты, как рецепт на витрине. Они доступны всем, и Solidity автоматически создает функцию для их чтения.



## Функции

Функции – это как шаги в рецепте. Они описывают, что нужно делать с ингредиентами для получения желаемого результата.

Пример:

```
1  ```solidity
2  function addSugar(uint grams) public {
3      sugarGrams += grams; // Добавляем сахар
4  }
5
```

## Модификаторы и События

Модификаторы – это особые условия приготовления. Например, "перемешивайте только деревянной ложкой". Они добавляют дополнительные условия к функциям.

События в Solidity – это как звонок таймера на кухне, сигнализирующий о важном событии, например, о том, что блюдо готово.

## Ошибки

Обработка ошибок в смарт-контракте похожа на исправление ошибок при приготовлении. Если что-то идет не так, вы останавливаетесь и исправляете ошибку, прежде чем продолжить.



При проектировании смарт-контракта важно тщательно продумать структуру переменных и функций, так как они определяют логику и безопасность контракта.

# Функции

Для определения функции используется следующий синтаксис:

**function** <идентификатор>([<список параметров>]) <**internal** | **private** | **public** | **external**> [**pure** | **view**] [(модификаторы)] [**returns** (<список типов возвращаемых значений>)]

При задании функции требуется в явном виде задать её видимость посредством одного из ключевых слов.

- **internal**, **private** – задают режим видимости функций аналогично переменным состояниям.
- **external** – часть интерфейса контракта, функция может быть вызвана извне, но при этом не может быть вызвана изнутри контракта напрямую.
- **public** – часть интерфейса контракта, функция может быть вызвана и извне, и изнутри контракта.

Также возможно указать один из трёх типов функции:

- **view** – функция может читать переменные состояния, но не может модифицировать их.
- **pure** – в соответствии с [определением](#), чистая функция всегда возвращает одно и то же значение для одинаковых аргументов, а также не имеет побочных эффектов. Чистая функция в solidity не может читать и модифицировать переменные состояния, получать баланс смарт-контракта функцией `address(this).balance` и вызывать функции не являющиеся чистыми.
- **payable** - функция может принимать на баланс смарт-контракта ETH.

Кроме того, в смарт-контракте могут быть определены несколько специальных функций:

- **fallback** – функция, выполняемая в том случае, если произошёл вызов неизвестного метода. Функция не имеет параметров и не возвращает значений. Контракт может содержать только одну функцию `fallback`. Функция должна иметь модификатор видимости `external` и может иметь тип `payable`.
- **receive** – функция, выполняемая в случае когда к контракту обратились с пустым вызовом. Это происходит, например, если смарт контракт указать получателем обычного перевода. Функция не имеет параметров и не возвращает значений. Контракт может содержать только одну функцию

receive. Функция должна иметь тип payable и модификатор видимости external

- **constructor** – функция, однократно исполняемая при размещении контракта. Контракт может иметь только один конструктор. Конструктор может быть internal или public. Код конструктора используется только для инициализации и не размещается в блокчейне.

Если в контракте нет функции receive, то при попытке отправки на него ETH будет исполнена функция fallback, в случае наличия у неё типа payable. Если же у контракта нет ни функции receive ни функции fallback типа payable, отправить на контракт ETH будет невозможно.

Fallback применяется для приёма эфира на счёт контракта, обработки некорректных транзакций.

### Создание Функции:

Это как здороваться с каждым, кто вас вызывает. Простая и чистая функция, не меняющая и не читающая состояние контракта.

Функция с Payable:

```
1 function deposit() public payable {  
2   // теперь здесь можно хранить ETH  
3 }  
4
```

Это как положить деньги в ваш контрактный "кошелек".

Можно сравнить создание функций в смарт-контракте с организацией концерта: каждая функция – это разный музыкальный номер. Некоторые номера открыты для всех (public), другие – только для VIP-гостей (external), а некоторые – личные репетиции за закрытыми дверями (private и internal).

Используйте различные типы функций для создания гибкого и безопасного смарт-контракта. Например, используйте *internal* функции для повторного использования кода внутри контракта, *public* или *external* для взаимодействия с пользователем, и *payable* для управления финансовыми транзакциями.

## Модификаторы

Модификаторы являются ключевым элементом языка Solidity и используются для изменения поведения функций или контрактов. Они представляют собой специальные функции, которые позволяют добавить дополнительную функциональность к другим функциям.

Модификаторы могут быть применены к функциям и могут проверять определенные условия перед выполнением целевой функции. Если условия, заданные модификатором, выполняются, то функция будет вызвана, в противном случае, выполнение функции будет прервано с ошибкой.

Модификаторы могут быть полезны для множества задач, таких как проверка прав доступа, проверка условий, обработка ошибок, логирование и другие. Они помогают повысить безопасность и эффективность кода, а также упростить его чтение и понимание.

Знак подчёркивания это специальный символ, используемый внутри модификаторов. Он используется чтобы связать модификатор с кодом основной функции и указать, где должен быть выполнен код основной функции.

## Ошибки

Solidity позволяет определять пользовательские типы ошибок для дальнейшего использования.

**error** <идентификатор>([список параметров]);

Ошибки возвращаются как результат выполнения контракта с помощью ключевого слова **revert** и могут использоваться для передачи значений.

## События

Смарт-контракт может записывать журнал (log) действий в состояние блокчейна, это делается с помощью событий (events). Когда событие порождается (emit), оно вместе с аргументами записывает в журнале (log).

События в смарт-контрактах Solidity можно сравнить с отправкой специальных газетных выпусков в блокчейн. Каждый выпуск (событие) содержит важную информацию о происходящем в контракте.

## Зачем нужны События?

- События служат мостом между блокчейном и внешним миром. Они сообщают внешним приложениям о важных моментах в жизни контракта, например, о завершении транзакции или изменениях в данных.

## Как Работают События?

- При возникновении события в контракте, оно записывает данные в журнал блокчейна. Это как размещение объявления в газете: информация доступна всем, но не может быть изменена после публикации.

## Пример Использования Событий:

- Рассмотрим смарт-контракт, который отслеживает покупки. Каждый раз, когда кто-то делает покупку, контракт может "издать" событие, информирующее о совершенной транзакции.

```
1 pragma solidity ^0.8.7;
2
3 contract Marketplace {
4     event Purchase(address indexed buyer, uint amount);
5
6     function buyItem(uint amount) public {
7         // Логика покупки
8         emit Purchase(msg.sender, amount); // Генерация события
9     }
10 }
11
```



События в Solidity — это как фотографии важных моментов, сделанные на вечеринке. Они захватывают и сохраняют особые мгновения (существенные изменения в контракте), которые можно вспоминать (просматривать) и обсуждать позже, но которые нельзя изменить после того, как были сделаны (записаны).

# Обработка ошибок

В Solidity реализован механизм поддержания консистентности состояния EVM. Транзакции являются атомарными - если транзакция по какой-либо причине не выполняется, то все изменения, которые были сделаны в состоянии откатываются. В основе механизма обработки ошибок лежат исключения. Исключения могут возникать автоматически, например, в случае деления на ноль или попытки обращения к элементу за пределами массива. Также исключения могут генерироваться пользовательским кодом с помощью функций `assert()`, `require()` и оператора `revert`.

Функция `assert(bool)` единственным параметром принимает булевское значение и генерирует исключение если это значение ложно.

Функция `require(bool, string)` также генерирует исключение типа `Error(string)`, если булевское значение было ложным.

`Assert()` и `require()` имеют два технических отличия. Во-первых `require()` обеспечивает гибкость, позволяя вернуть дополнительную информацию об ошибке. Во-вторых `require()` останавливает выполнение функции, при этом тратится только фактически израсходованный газ, в то время как `assert()` полностью спишет весь доступный для выполнения газ (`gaslimit`).

Семантически `assert()` используется только для внутренних проверок, например консистентности данных, а `require()` - для валидации пользовательского ввода. Это учитывается статическими анализаторами кода.

Оператор `revert` позволяет вернуть ошибку, определённую пользователем и, кроме того, обеспечивает большую гибкость в использовании.

## Итоговое практическое задание: разработка контракта для организации событий

### Описание задачи:

Вам предстоит создать смарт-контракт, который позволяет пользователям создавать, просматривать и бронировать места на различные события (концерты, конференции, мастер-классы и т. д.). В задаче есть подсказка, попробуйте сначала решить без нее.

Спецификации контракта:

Структура Event:

- Название события (string)
- Дата и время (uint, представляющее timestamp)
- Цена билета (uint)
- Общее количество мест (uint)
- Количество доступных мест (uint)
- Список участников (массив адресов)

Функционал контракта:

- `createEvent (string memory name, uint dateTime, uint ticketPrice, uint totalSeats)`: создание нового события. Доступно только владельцу контракта.
- `getEvents()`: возвращает список всех событий.
- `bookTicket(uint eventId) payable`: бронирование билета на событие. Функция должна быть `payable`, чтобы принимать ETH в качестве оплаты билета.
- `cancelEvent(uint eventId)`: отмена события. Доступно только владельцу контракта.
- `refundTicket(uint eventId, address attendee)`: возврат средств за билет. Доступно только владельцу контракта.

Конструктор контракта:

- Назначает `msg.sender` владельцем контракта.

**Цель:** дополнить смарт-контракт, реализовав описанные функции, а также обеспечив безопасность и корректную обработку ошибок.

**Ответ:**

Для выполнения данного практического задания по разработке контракта для организации событий в блокчейне, вам потребуется написать код на языке Solidity. Ниже приведен пример реализации контракта, удовлетворяющего приведенным спецификациям:

Этот контракт позволяет создавать события, просматривать список всех событий, бронировать билеты на события, отменять события и организовывать возврат средств за билеты. Владелец контракта может управлять событиями и билетами, а участники могут забронировать билеты на доступные мероприятия.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract EventContract {
5     address public owner;
6     uint public eventCount;
7     mapping(uint => Event) public events;
8
9     struct Event {
10         string name;
11         uint dateTime;
12         uint ticketPrice;
13         uint totalSeats;
14         uint availableSeats;
15         address[] attendees;
16         mapping(address => bool) hasAttended;
17     }
18
19     constructor() {
20         owner = msg.sender;
21         eventCount = 0;
22     }
23
24     modifier onlyOwner() {
25         require(msg.sender == owner, "Only owner can call this
function");
26         _;
27     }
28
29     function createEvent(string memory name, uint dateTime, uint
ticketPrice, uint totalSeats) public onlyOwner {
30         events[eventCount] = Event(name, dateTime, ticketPrice,
totalSeats, totalSeats, new address[]());
31         eventCount++;
32     }
33
34     function getEvents() public view returns (Event[] memory) {
35         Event[] memory allEvents = new Event[](eventCount);
36         for (uint i = 0; i < eventCount; i++) {
37             allEvents[i] = events[i];
38         }
39         return allEvents;
40     }
41
42     function bookTicket(uint eventId) public payable {
43         require(eventId < eventCount, "Invalid event ID");
44         Event storage selectedEvent = events[eventId];
45         require(selectedEvent.availableSeats > 0, "No available
seats");
46
47         selectedEvent.attendees.push(msg.sender);
48         selectedEvent.availableSeats--;
49     }
50
51     function cancelEvent(uint eventId) public onlyOwner {
52         require(eventId < eventCount, "Invalid event ID");
53         Event storage selectedEvent = events[eventId];
54         selectedEvent.availableSeats = 0;
55     }
56
57     function refundTicket(uint eventId, address attendee) public
onlyOwner {
58         require(eventId < eventCount, "Invalid event ID");
59         Event storage selectedEvent = events[eventId];
60         require(selectedEvent.hasAttended[attendee] == false,
"Attendee has already attended");
61
62         for (uint i = 0; i < selectedEvent.attendees.length; i++) {
63             if (selectedEvent.attendees[i] == attendee) {
64                 selectedEvent.attendees[i] =
selectedEvent.attendees[selectedEvent.attendees.length - 1];
65                 selectedEvent.attendees.pop();
66                 selectedEvent.availableSeats++;
67                 break;
68             }
69         }
70     }
71 }
72

```



## Итоги лекции:

- Мы ознакомились с моделью памяти В Solidity, поняли что область хранения данных разделена на три зоны: *стек (stack)*, *память (memory)* и *состояние EVM (state)*, рассмотрели уникальные особенности каждой зоны.
- Из материалов лекции поняли, что такое массивы, структуры и отображения в языке Solidity, изучили их функционал.
- Познакомились со структурой контракта и ее составляющими, а также выполнили не сложные, но очень полезные практические задания.

Спасибо за Внимание! Увидимся на следующей лекции!

## Использованная литература

1. <https://docs.soliditylang.org>
2. <https://ethereum.github.io/yellowpaper/paper.pdf>

-----