

Введение в Solidity

Основы разработки на Solidity



Оглавление

Введение	3
Словарь терминов	4
Solidity	4
История языка	4
Чем обусловлен интерес к Solidity?	5
Есть и минусы ...	6
Интерфейс Remix	8
Работа с Remix на примере “Hello world”	10
Конструкции и синтаксис Solidity	13
Практическое задание: смарт-контракт для управления счетами	16
Задачи для закрепления синтаксиса	20
Задание: Создание и Управление Счетом	22
Использованная литература	24

Введение

Добрый день, дорогие Друзья!

На этой лекции мы познакомимся с языком программирования Solidity, узнаем его особенности, области применения, структуру языка и даже напишем наш первый смарт-контракт! Начну сразу с хороших новостей: Solidity сравнительно несложный и отлично подойдёт в качестве первого языка программирования.

Язык Solidity является относительно молодым, однако зарплаты блокчейн разработчиков с опытом 2-3 года часто превышают зарплаты опытных devops-инженеров или опытных python-разработчиков. Индустрия блокчейна постоянно развивается и растёт, а язык Solidity позволяет писать код не только для Ethereum, но и для множества других блокчейнов. Тем не менее квалифицированных специалистов в этой области недостаточно.

Нельзя не отметить, что спрос на разработку на Solidity и блокчейн тесно связан с популярностью криптовалюты. Чем больше возникает "хайпа", тем больше возрастает спрос на специалистов в этой области.

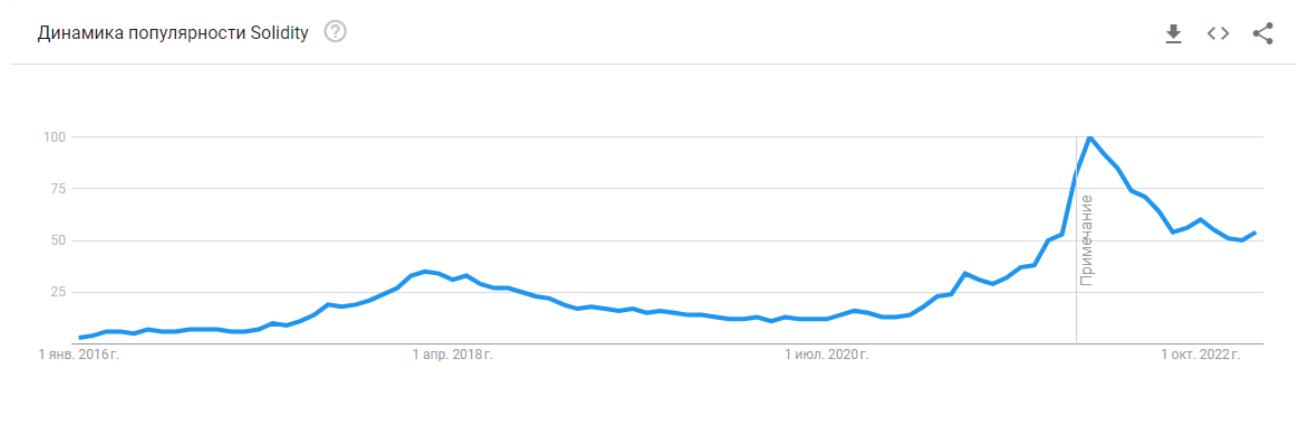


График взят из Google Trends

На этой лекции вы узнаете:

- Историю, особенности языка, преимущества и недостатки.
- Синтаксис языка.
- Напишем первый смарт-контракт.

Solidity

Язык Solidity – это язык программирования, специально разработанный для написания смарт-контрактов на блокчейне Ethereum. Смарт-контракты – это программы, которые выполняются автоматически и без возможности изменений при заданных условиях.

Solidity является статически типизированным языком, схожим с языком программирования JavaScript, синтаксис которого вдохновлён языком C++. Он предлагает богатый набор возможностей для создания различных функций, структур данных, модификаторов доступа и других вещей, которые делают возможной реализацию сложной логики в смарт-контрактах.

Solidity поддерживает объектно-ориентированное программирование, а также возможность создания интерфейсов, наследования и полиморфизма. Эти функции позволяют создавать более абстрактные и гибкие контракты, что облегчает разработку и повторное использование кода.

Одно из ключевых преимуществ Solidity - это возможность использования специальных типов данных, таких как адреса, строки и массивы. Также он позволяет работать с общедоступным реестром данных, называемым хранилищем, где можно сохранять данные на блокчейне.

Solidity также предлагает механизмы событий и модификаторов. События позволяют контракту взаимодействовать с внешним миром и уведомлять стороны о выполнении определённых действий. Модификаторы, с другой стороны, позволяют контролировать доступ к функциям и изменять поведение контракта.

История языка

Работа над разработкой Solidity началась в 2014 году, когда команда Ethereum приступила к созданию своего блокчейна. Основной целью создания этого языка программирования, было обеспечение безопасности смарт-контрактов, которые могут выполняться на платформе Ethereum.

Идея создания Solidity была предложена Гэвином Вудом - бывшим научным сотрудником Microsoft, сооснователем Ethereum, учёным информатиком, а также

создателем Polkadot и Kusama. Однако основная часть разработки языка происходила под руководством Христиана Райтвинзера, а также в команде участвовали Либойс Лавльянд, Алекс Беркман, Якоб Хаун и Пипп Макк.

При создании Solidity разработчики вдохновлялись такими языками программирования, как Python, C++ и JavaScript. Язык также учитывал уникальные возможности, предоставляемые блокчейном Ethereum, включая возможность создания цифровых активов, использование токенов и управление децентрализованными приложениями.

Первая стабильная версия Solidity была выпущена в 2015 году, с тех пор язык активно развивается и постоянно изменяется. Вокруг него сформировалось большое и активное сообщество разработчиков, вносящих вклад в его развитие. На сегодняшний день Solidity является самым популярным языком программирования для написания смарт-контрактов.

Чем обусловлен интерес к Solidity?

Solidity имеет несколько значительных преимуществ, которые делают его популярным и предпочтительным языком для разработки смарт-контрактов на платформе Ethereum. Солидити входит в ТОП самых высокооплачиваемых языков программирования в 2022 году. Вот некоторые из основных преимуществ языка Solidity:

- Специально разработанный для Ethereum:** Solidity является языком программирования, созданным специально для работы с Ethereum и разработки смарт-контрактов на этой платформе. Он предназначен для обеспечения безопасного и надёжного взаимодействия с блокчейном Ethereum.
- Объектно-ориентированный подход:** Solidity поддерживает принципы объектно-ориентированного программирования (ООП), такие как наследование, полиморфизм и инкапсуляция. Это позволяет разработчикам создавать более структурированные и модульные смарт-контракты.
- Широкий набор функциональности:** Solidity предлагает богатый набор функций и возможностей, которые делают его мощным и гибким языком программирования. Он поддерживает различные типы данных, операции и контроль потока

выполнения, что позволяет разработчикам реализовывать сложную логику в смарт-контрактах.

4. Безопасность: Solidity включает в себя множество механизмов безопасности, которые помогают предотвратить уязвимости и атаки на смарт-контракты. Он предоставляет инструменты для проверки границ массивов, защиты от переполнения, контроля доступа к функциям и других видов проверок безопасности.

5. Интеграция с другими языками и инструментами: Solidity хорошо интегрируется с другими языками программирования, такими как JavaScript, Python и Java, и позволяет разработчикам использовать функции и библиотеки из других языков. Кроме того, Solidity поддерживает стандарты токенов ERC-20 и ERC-721, что облегчает создание и взаимодействие с токенами на блокчейне Ethereum.

6. Активное сообщество разработчиков: Solidity имеет мощное сообщество разработчиков, которые охотно делятся опытом, предлагают решения и консультируют других разработчиков. Это делает процесс разработки смарт-контрактов на Solidity более поддерживаемым и доступным.

Есть и минусы ...

Несмотря на множество преимуществ языка Solidity, есть и некоторые минусы, которые следует учитывать при разработке смарт-контрактов на Ethereum. Вот некоторые из них:

1. Отсутствие полной надежности: Solidity является относительно новым языком программирования и все еще находится в процессе развития. Это означает, что иногда могут возникать ошибки и неисправности в функциональности языка, что может привести к непредсказуемому поведению смарт-контрактов.

2. Ограниченная экосистема инструментов: в настоящее время существует ограниченное количество инструментов и библиотек, доступных для разработки смарт-контрактов на языке Solidity. Это уменьшает возможности разработчиков в выборе и использовании современных инструментов разработки и тестирования.

3. Сложность аудита и безопасности: Solidity, как и любой язык программирования, имеет ряд уязвимостей и потенциальных проблем безопасности. Разработчику необходимо быть особенно внимательным и аккуратным, чтобы избежать ошибок, таких как уязвимости реентранции или переполнения стека, которые могут привести к утечке средств или взлому контракта.

4. **Затратность выполнения операций:** выполнение смарт-контрактов на Ethereum требует оплаты за каждую операцию, выполняемую контрактом. Solidity не всегда оптимизирует затраты газа и может быть неэффективным с точки зрения стоимости выполнения крупных и сложных контрактов.
5. **Ограниченнaя поддержка разработчиков:** поскольку Solidity относительно новый язык программирования, количество разработчиков, знакомых с ним, может быть ограничено. Это означает, что, возможно, будет сложнее найти квалифицированных разработчиков для работы с Solidity и решением связанных с ним проблем.

Простые смарт-контракты на Solidity

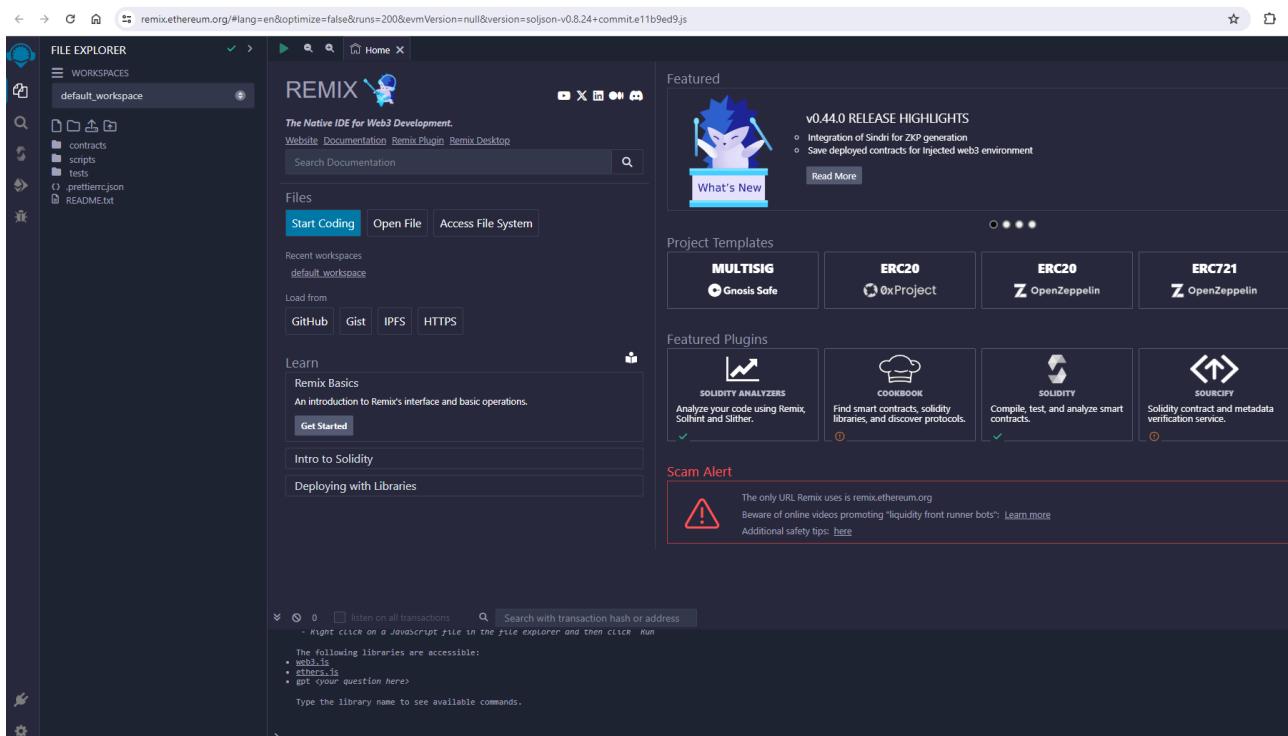
Код на Solidity можно писать в любом текстовом редакторе или интегрированной среде разработки (IDE), которая поддерживает язык программирования Solidity.

Популярные инструменты разработки Solidity включают в себя:

1. **Remix:** онлайн IDE для разработки и тестирования контрактов Solidity. Он предлагает широкий спектр функций, включая отладку контрактов, компиляцию и развёртывание на различных средах выполнения.
2. **Visual Studio Code:** популярный текстовый редактор, который может быть настроен для разработки Solidity с помощью дополнений, таких как "Solidity Visual Studio Code".
3. **Truffle:** фреймворк для разработки децентрализованных приложений (DApps) на базе Ethereum. Он предоставляет инструменты для компиляции, тестирования и развёртывания контрактов Solidity.
4. **IntelliJ IDEA с плагином Solidity:** IntelliJ IDEA - это популярная IDE, которая поддерживает Solidity через плагин, позволяющий разрабатывать и отлаживать контракты Solidity.

Мы рекомендуем Remix для небольших контрактов и для быстрого изучения Solidity.

Интерфейс Remix



На скриншоте представлен пользовательский интерфейс Remix IDE, популярной интегрированной среды разработки для написания смарт-контрактов на языке Solidity.

Вот разъяснения основных элементов интерфейса:

1. File Explorer (Левая панель):

- Создание файла/папки/работы: иконки в верхней части позволяют создавать новые файлы, папки или работы (workspace).
- Contracts: папка, где обычно хранятся файлы смарт-контрактов.
- Scripts: папка для хранения пользовательских скриптов для развёртывания и взаимодействия с контрактами.
- Tests: папка для тестовых скриптов, используемых для тестирования смарт-контрактов.
- README.txt: файл с инструкциями или информацией о проекте.

2. Main Panel (Центральная панель):

- Featured: секция с рекомендациями и новостями от Remix.

- Project Templates: шаблоны проектов для быстрого старта (например, ERC20, ERC721).
- Featured Plugins: рекомендуемые плагины для расширения функциональности IDE, такие как Solidity Analyzers, Cookbook, Solidity и Sourcify.
- Scam Alert: предупреждение о мошенничестве, напоминающее об осторожности при использовании сторонних ресурсов.

3. Terminal (Нижняя панель):

- Терминал: место, где отображаются сообщения компилятора, ошибки, логи транзакций и другая информация.

4. Левый Сайдбар (Левая боковая панель):

- Компилятор Solidity: где вы компилируете свои смарт-контракты.
- Иконки развёртывания и выполнения транзакций: позволяют развёртывать смарт-контракты и выполнять транзакции.

5. Верхняя Панель:

- Home: возврат на главную страницу.
- Search Documentation: поиск по документации.
- Plugin Manager: управление плагинами.
- Settings: настройки IDE.

6. Правый Сайдбар (Правая боковая панель):

- Отображает различные плагины или вкладки, которые могут быть активированы для использования в IDE, такие как анализаторы кода, управление файлами и другие инструменты.

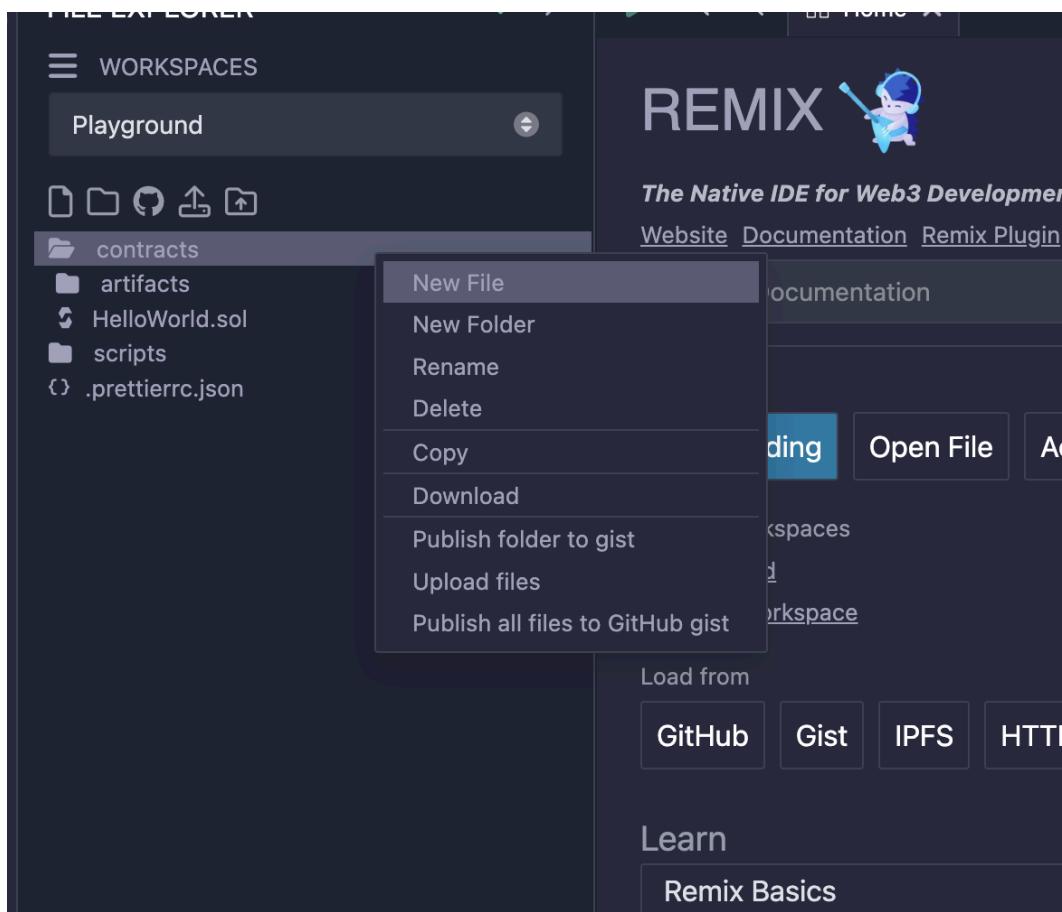
7. Область Уведомлений (Правый верхний угол):

- What's New: показывает последние обновления IDE.
- Сообщения о выпуске: информация о последних релизах Remix IDE.

Работа с Remix на примере “Hello world”

Remix, как мы говорили ранее, работает в формате онлайн IDE и не требует установки. Вам нужно перейти по [ссылке](#) и следовать инструкциям.

- 1) Чтобы начать делать новый контракт, создайте новый файл в папке `\textit{contracts}`, расположенной в левой части экрана. Мы можем видеть это на рисунке ниже.



- 2) Назовем этот новый файл `HelloWorld.sol`. Расширение для файлов Solidity указывается как - `".sol"`. Remix откроет этот новый файл, и мы сможем начать писать смарт контракт.
- 3) Рекомендуется включить указание лицензии, поскольку ваш код будет публичным. Это не обязательно, но компилятор выдаст ошибку, если вы это не сделаете.

```
// SPDX-License-Identifier: MIT
```

- 4) Начнём с указания версии компилятора, которую мы хотим запустить, с помощью директивы pragma, как `pragma solidity ^0.8.7;`



Существует несколько способов определить используемую версию компилятора. Самый простой – указать полную версию, например, 0.8.7. В общем случае, однако, мы хотим использовать любую версию, большую, чем указанная, если она не претерпела изменений, которые могут нарушить обратную совместимость. Для этого мы используем символ "^". Это означает, что компилятор может быть версии 0.8.8, 0.8.9 и т. д., но не версии 0.9.x и выше.

- 5) Чтобы указать, что мы начнём писать новый контракт, мы используем предложение `contract`

```
contract HelloWorld {  
    function helloWorld() external pure returns (string memory) {  
        return "Hello, World!";  
    }  
}
```



```
1 // SPDX-License-Identifier: MIT  
2 pragma solidity >=0.6.12 <0.9.0;  
3 contract HelloWorld {  
4     function helloWorld() external pure returns (string memory) { infinite gas  
5         return "Hello, World!";  
6     }  
7 }  
8  
9  
10  
11
```

Solidity использует синтаксис, основанный на фигурных скобках { }, подобно C, Java, JavaScript и т.д. Всё, что находится внутри contract, будет частью нашего контракта, но компилятор также будет использовать объявления за пределами этой области.

Вы можете подумать, что объявление контракта похоже на объявление нового класса. На самом деле, контракты могут быть инстанцированы другими контрактами, подобно тому как мы делаем это с объектами.

Вот такой наш первый смарт-контракт. Он ничего не сохраняет, ничего полезного не делает, а только говорит: "Привет, Мир!"



Рекомендуется самостоятельно повторить представленный пример смарт-контракта "HelloWorld" в вашей среде разработки. Это позволит вам на практике применить изученные концепции языка Solidity и лучше усвоить основы написания смарт-контрактов. Помните, что практический опыт – ключевой элемент в обучении программированию.

Конструкции и синтаксис Solidity

Конструкции языка Solidity имеют свой специфический синтаксис. Вот некоторые основные элементы синтаксиса Solidity:

```
go

package main

import "fmt"

func isEven(number int) bool {
    return number%2 == 0
}

func main() {
    fmt.Println(isEven(4)) // Выведет: true
    fmt.Println(isEven(5)) // Выведет: false
}
```

```
javascript

function isEven(number) {
    return number % 2 === 0;
}

console.log(isEven(4)); // Выведет: true
console.log(isEven(5)); // Выведет: false
```

```
cpp

#include <iostream>

bool isEven(int number) {
    return number % 2 == 0;
}

int main() {
    std::cout << std::boolalpha;
    std::cout << isEven(4) << std::endl; // Выведет: true
    std::cout << isEven(5) << std::endl; // Выведет: false
    return 0;
}
```

Солидити:

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.7;

contract EvenChecker {
    function isEven(uint number) public pure returns (bool) {
        return number % 2 == 0;
    }
}
```

😄 Как мы видим, язык Solidity не такой сложный, как C++

1. Объявление контракта:

```
1 contract <ContractName> {
2     // Код контракта
3 }
```

Однако не стоит его бояться, вот простенькая программа проверяющая число на четность.

2. Объявление переменных:

При объявлении переменных обязательно нужно указывать их тип.

- **uint** – беззнаковое целое число.

- **int** – знаковое целое число.

- **string** – строка.

- **address** – адрес Ethereum. Это уникальный тип данных, с которым вы, скорее всего, раньше не работали. По отношению к адресам мы можем применять следующие методы: `balance`, `transfer`, `send`, `call`, `callcode`, and `delegatecall`. Пример кода для проверки баланса адреса:

```
1 function getAddressBalance(address x) public view returns(uint) {
2     return address(x).balance;
3 }
```

- **bool** – логическое значение (истина или ложь).

- **mapping** – отображение из одного типа данных в другой.

- **struct** – пользовательский тип данных, который может содержать переменные разных типов.

```
1 <тип данных> <имя переменной>;
```

Пример:

```
1 uint myNumber;
2 string myString;
3 address payable myAddress;
```

3. Объявление функций:

Нужно обязательно указывать тип переменной, которую мы хотим вернуть в результате выполнения функции

- **function** - ключевое слово для объявления функций в Solidity.

- **view** - ключевое слово, указывающее, что функция не изменяет состояние контракта (чтение).

- **pure** - ключевое слово, указывающее, что функция выполняет только вычисления (не взаимодействует с контрактом и не читает состояние).

```
1 <тип данных> <имя функции> (<параметры>) <модификаторы доступа> {
2     // Тело функции
3 }
```

Примеры:

```
1 function add(uint a, uint b) public returns(uint) {
2     uint result = a + b;
3     return result;
4 }
5
6 function getBalance() public view returns(uint) {
7     return address(this).balance;
8 }
```

Практическое задание: смарт-контракт для управления счетами

Цель:

Создать смарт-контракт, который позволяет вести учёт балансов пользователей и проверять, является ли пользователь владельцем контракта.

Задача:

- Объявить контракт с названием `AccountManager`.
- В контракте объявить переменную `owner` типа `address`, которая хранит адрес владельца контракта.
- Создать `mapping` для отслеживания балансов пользователей.
- Написать функцию `isOwner`, которая проверяет, является ли отправитель транзакции владельцем контракта.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract AccountManager {
    address public owner;
    mapping(address => uint) public balances;

    constructor() {
        owner = msg.sender;
    }
}
```

```
function isOwner() public view returns (bool) {  
    // Здесь будет ваш код  
}  
  
// Другие функции и модификаторы по желанию  
}
```

Вопрос:

Какой код нужно вставить в функцию `isOwner`, чтобы она корректно выполняла свою задачу? Выберите один из четырёх вариантов ответов.

Варианты ответа:

- a) `return msg.sender == owner;`
 - Пояснение: этот код правильно сравнивает адрес отправителя (`msg.sender`) с адресом владельца (`owner`).
- б) `return owner;`
 - Пояснение: это неправильно, так как функция должна возвращать логическое значение (`bool`), а не адрес.
- в) `if (msg.sender == owner) return true; else return false;`
 - Пояснение: хотя это работает, код избыточен и может быть упрощён до первого варианта.
- г) `return balances[msg.sender] > 0;`
 - Пояснение: этот код проверяет баланс отправителя, что не относится к проверке на владельца контракта.

Правильный ответ: а)

4. Модификаторы доступа:

- **public** - метод или переменная доступны из любого контракта или адреса.
- **private** - метод или переменная доступны только внутри текущего контракта.
- **internal** - метод или переменная доступны только внутри текущего контракта и его производных контрактов.

- **external** - метод может быть вызван только внешним контрактом или извне текущего контракта.

5. Условные операторы:

- **if-else** - условный оператор, который выполняет блок кода, если условие истинно, иначе выполняет другой блок кода.

- **switch** - условный оператор, который позволяет выбрать один из нескольких вариантов для выполнения.

```
1 if (<условие>) {  
2     // Блок кода, выполняющийся, если условие истинно  
3 } else {  
4     // Блок кода, выполняющийся, если условие ложно  
5 }
```

Пример:

```
1 if (a > b) {  
2     // Код  
3 } else {  
4     // Код  
5 }
```

6. Циклы:

- **for** - цикл, который выполняет блок кода заданное количество раз.

```
1 ` ` `  
2 for (<инициализация>; <условие>; <шаг>) {  
3     // Блок кода, выполняющийся в каждой итерации  
4 } ` ` `
```

- **while** - цикл, который выполняет блок кода, пока условие истинно.

```
1  ```
2  while (<условие>) {
3      // Блок кода, выполняющийся, пока условие истинно
4  }```
5
```

- **do-while** - цикл, который выполняет блок кода, а затем проверяет условие.

```
1  ```
2  do {
3      // Блок кода, выполняющийся минимум один раз
4  } while (<условие>);
5  ````
```

Исключения:

- **try-catch** - блок кода, который позволяет перехватывать и обрабатывать исключения.

```
1 try {
2     // Код, который может вызвать исключение
3 } catch {
4     // Обработка исключения
5 }
```

События:

- **event** - ключевое слово для объявления событий, которые могут быть вызваны в контракте и прослушиваться внешне.



Задачи для закрепления синтаксиса

Задачи могут показаться лёгкими, так как вы ранее изучали другие языки программирования, но решение этих задач поможет вам быстрее привыкнуть к языку Solidity:

Задание 1: использование циклов для агрегации данных.

Цель: научиться использовать циклы для агрегации данных в Solidity.

Задача: Создать смарт-контракт, который управляет списком целых чисел.

Реализовать функцию, которая использует цикл для подсчёта суммы всех чисел в списке.

Шаги:

- Объявить динамический массив `uint` для хранения списка чисел.
- Написать функцию `addNumber(uint number)`, которая добавляет число в массив.
- Реализовать функцию `sumNumbers()`, которая возвращает сумму всех чисел в массиве. Для этого использовать цикл `for` или `while`, перебирающий элементы массива и суммирующий их значения.

Ответ:

Для решения данной задачи вам потребуется создать смарт-контракт на Solidity, который будет управлять списком целых чисел. Вот пример кода с контрактом, функцией для добавления числа в массив и функцией для подсчёта суммы всех чисел в массиве с использованием цикла:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.7;
3
4 contract NumberList {
5     uint[] public numbers;
6
7     function addNumber(uint number) public {
8         numbers.push(number);
9     }
10
11    function sumNumbers() public view returns (uint) {
12        uint sum = 0;
13        for (uint i = 0; i < numbers.length; i++) {
14            sum += numbers[i];
15        }
16        return sum;
17    }
18 }
19
```

В этом контракте объявляется динамический массив `numbers`, в который можно добавлять целые числа с помощью функции `addNumber`. Функция `sumNumbers` использует цикл `for` для перебора элементов массива и подсчёта их суммы. После прохождения цикла возвращается общая сумма чисел.

Теперь у вас есть контракт, который позволяет управлять списком целых чисел и выполнять операции агрегации данных, такие как подсчёт суммы.

Задание 2: работа с переменными

Цель: научиться использовать и манипулировать различными типами переменных в Solidity.

Задача: создать контракт, в котором будут объявлены переменные типов `'uint'`, `'int'`, `'bool'`, и `'address'`. Реализовать функции для изменения и чтения значений этих переменных, демонстрируя основные операции над данными.

Ответ:

Для выполнения задания по работе с переменными в Solidity, вам необходимо создать контракт, в котором будут объявлены переменные различных типов и реализовать функции для их изменения и чтения значений. Вот пример кода с контрактом, включающим переменные типов `uint`, `int`, `bool` и `address`, а также функции для манипулирования этими переменными:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.7;
3
4 contract VariablesContract {
5     uint public uintVariable;
6     int public intVariable;
7     bool public boolVariable;
8     address public addressVariable;
9
10    constructor() {
11        uintVariable = 10;
12        intVariable = -5;
13        boolVariable = true;
14        addressVariable = msg.sender;
15    }
16
17    function setUintVariable(uint newValue) public {
18        uintVariable = newValue;
19    }
20
21    function getIntVariable() public view returns (int) {
22        return intVariable;
23    }
24
25    function toggleBoolVariable() public {
26        boolVariable = !boolVariable;
27    }
28
29    function setAddressVariable(address newAddress) public {
30        addressVariable = newAddress;
31    }
32 }
33

```

В этом контракте объявлены переменные типов uint, int, bool и address, а также реализованы функции для изменения и чтения значений этих переменных. Конструктор устанавливает начальные значения переменных. Функция setUintVariable изменяет значение переменной типа uint, функция getIntVariable возвращает значение переменной типа int, функция toggleBoolVariable инвертирует значение переменной типа bool, а функция setAddressVariable устанавливает новое значение переменной типа address.

Таким образом, вы сможете управлять переменными различных типов в контракте и выполнять основные операции над данными с их помощью.

Задание 3: создание и использование массивов

Цель: освоить работу с массивами в Solidity, включая динамические операции.

Задача: разработать контракт с массивом целых чисел. Написать функции для добавления элементов в массив и извлечения элемента по индексу, показывая, как управлять массивами в Solidity.

Ответ:

Вот пример контракта на языке Solidity, который выполняет указанные задачи:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract ArrayExample {
5     uint[] public integerArray;
6
7     function addElement(uint _element) public {
8         integerArray.push(_element);
9     }
10
11    function getElement(uint _index) public view returns (uint) {
12        require(_index < integerArray.length, "Index out of
13        bounds");
14        return integerArray[_index];
15    }
16 }
```

В данном примере контракт `ArrayExample` содержит массив `integerArray`, в который мы можем добавлять элементы при помощи функции `addElement` и извлекать элемент по индексу с помощью функции `getElement`. Важно убедиться, что перед извлечением элемента по индексу мы проверяем, что индекс находится в пределах массива.

Задание 4: контроль доступа и модификаторы

Цель: понять и правильно применять модификаторы доступа для управления видимостью функций и переменных.

Задача: создать контракт с функциями, имеющими различные модификаторы доступа (`public`, `private`, `internal`, `external`). Продемонстрировать, как каждый модификатор влияет на доступность функций в разных контекстах.

Ответ:

Вот пример контракта на Solidity, который демонстрирует различные модификаторы доступа:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract AccessControlExample {
5     uint private privateValue;
6     uint internal internalValue;
7     uint public publicValue;
8
9     constructor(uint _initialValue) {
10         privateValue = _initialValue;
11         internalValue = _initialValue;
12         publicValue = _initialValue;
13     }
14
15     function getPrivateValue() private view returns (uint) {
16         return privateValue;
17     }
18
19     function getInternalValue() internal view returns (uint) {
20         return internalValue;
21     }
22
23     function getPublicValue() public view returns (uint) {
24         return publicValue;
25     }
26
27     function setPrivateValue(uint _newValue) private {
28         privateValue = _newValue;
29     }
30
31     function setInternalValue(uint _newValue) internal {
32         internalValue = _newValue;
33     }
34
35     function setPublicValue(uint _newValue) public {
36         publicValue = _newValue;
37     }
38 }
39

```

В данном контракте AccessControlExample у нас есть три переменные `privateValue`, `internalValue` и `publicValue`, каждая из которых сопровождается функцией чтения и функцией записи. Модификаторы доступа `private`, `internal`, `public` определяют, кто и как может вызывать эти функции.

Модификатор `private` обеспечивает доступ только внутри контракта, `internal` позволяет доступ из дочерних контрактов и из контрактов, находящихся в одном файле, а модификатор `public` делает функции доступными для всех.

Задание 5: управление потоком – условные операторы

Цель: изучить и применить условные операторы в Solidity для контроля потока выполнения.

Задача: разработать функции, использующие `'if-else'` и `'switch'` для выполнения различных операций в зависимости от заданных условий.

Ответ:

Для выполнения задания 5 по управлению потоком с использованием условных операторов в Solidity, вы можете создать следующий контракт, который содержит функции с условными операторами `if-else` и `switch`:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract FlowControlExample {
5
6     // Функция с использованием оператора if-else
7     function checkNumber(uint num) public pure returns (string memory) {
8         if (num > 0) {
9             return "Число положительное";
10        } else {
11            return "Число не положительное";
12        }
13    }
14
15    // Функция с использованием оператора switch
16    function getDay(uint day) public pure returns (string memory) {
17        string memory result;
18        switch (day) {
19            case 1:
20                result = "Понедельник";
21                break;
22            case 2:
23                result = "Вторник";
24                break;
25            case 3:
26                result = "Среда";
27                break;
28            case 4:
29                result = "Четверг";
30                break;
31            case 5:
32                result = "Пятница";
33                break;
34            case 6:
35                result = "Суббота";
36                break;
37            case 7:
38                result = "Воскресенье";
39                break;
40            default:
41                result = "Некорректный день недели";
42                break;
43        }
44        return result;
45    }
46 }
47 }
```

Этот контракт содержит две функции: `checkNumber`, которая использует оператор `if-else` для проверки числа на положительность, и `getDay`, которая использует оператор `switch` для возвращения дня недели по номеру.

Такой контракт поможет вам понять и применить условные операторы в Solidity для управления потоком выполнения программы.

Задание 6: циклы в Solidity

Цель: овладеть использованием циклических конструкций для повторяющихся операций.

Задача: написать функцию, в которой будут использоваться различные виды циклов (`'for'`, `'while'`, `'do-while'`), чтобы выполнить серию операций несколько раз.

Ответ:

Вот пример функции на Solidity, которая демонстрирует использование различных циклов:

```

1 pragma solidity ^0.8.0;
2
3 contract LoopExample {
4     // Использование цикла for для выполнения операций несколько раз
5     function forLoop() public pure returns (uint) {
6         uint sum = 0;
7         for (uint i = 0; i < 10; i++) {
8             sum += i;
9         }
10        return sum; // Возвращает сумму чисел от 0 до 9
11    }
12
13    // Использование цикла while для выполнения операций до тех пор,
14    // пока условие истинно
14    function whileLoop() public pure returns (uint) {
15        uint sum = 0;
16        uint i = 0;
17        while (i < 10) {
18            sum += i;
19            i++;
20        }
21        return sum; // Возвращает сумму чисел от 0 до 9
22    }
23
24    // Использование цикла do-while для выполнения операций хотя бы
25    // один раз
25    function doWhileLoop() public pure returns (uint) {
26        uint sum = 0;
27        uint i = 0;
28        do {
29            sum += i;
30            i++;
31        } while (i < 10);
32        return sum; // Возвращает сумму чисел от 0 до 9
33    }
34 }
35

```

Этот контракт содержит три функции, каждая из которых использует разный тип цикла для вычисления суммы чисел от 0 до 9. Вы можете использовать этот пример как основу для создания более сложных функций, которые выполняют повторяющиеся операции с использованием циклов в Solidity. Удачи в освоении циклов!

Задание 7: обработка исключений

Цель: узнать, как обрабатывать ошибки и исключения в Solidity.

Задача: реализовать функцию с блоками `try-catch`, которая попытается выполнить операцию, потенциально вызывающую ошибку, и корректно обработает возникшее исключение.

Ответ:

В Solidity обработка исключений осуществляется с помощью блоков try и catch. Вот пример функции, которая демонстрирует этот механизм:

```
1 pragma solidity ^0.8.0;
2
3 contract ExceptionHandling {
4     // Эта функция вызывает другую функцию, которая может выбросить
5     // исключение
6     function callFunctionThatMayThrow() public returns (string
7         memory) {
8         try this.functionThatThrows() {
9             // Если нет ошибки, выполнится этот блок
10            return "Function executed successfully";
11        } catch {
12            // Если функция выбросит исключение, выполнится этот
13            // блок
14            return "An error occurred";
15        }
16    }
17    // Эта функция специально выбрасывает исключение для
18    // демонстрации
19    function functionThatThrows() public pure {
20        // Здесь может быть код, который потенциально может вызвать
21        // ошибку
22        revert("Error thrown intentionally");
23    }
24 }
```

В этом примере functionThatThrows специально выбрасывает исключение с помощью revert. Функция callFunctionThatMayThrow пытается вызвать functionThatThrows и обрабатывает возникшее исключение в блоке catch. Это позволяет программе продолжать работу, несмотря на возникновение ошибки. Используйте этот пример как основу для создания своих функций с обработкой исключений в Solidity.

Задание 8: работа со структурами и перечислениями

Цель: изучить применение структур данных и перечислений в контексте смарт-контрактов.

Задача: создать структуру данных и перечисление, используемые в функциях контракта, чтобы демонстрировать, как они могут упростить и структурировать данные.

Ответ:

В Solidity структуры данных и перечисления позволяют организовать и управлять данными более эффективно. Вот пример контракта, который использует структуру и перечисление:

```

1 pragma solidity ^0.8.0;
2
3 // Объявление перечисления для статусов задач
4 enum TaskStatus {
5     ToDo,
6     InProgress,
7     Done,
8     Blocked
9 }
10
11 // Объявление структуры для представления задачи
12 struct Task {
13     string description;
14     TaskStatus status;
15 }
16
17 contract TaskManager {
18     // Массив для хранения задач
19     Task[] private tasks;
20
21     // Функция для добавления новой задачи
22     function addTask(string memory _description) public {
23         tasks.push(Task({
24             description: _description,
25             status: TaskStatus ToDo
26         }));
27     }
28
29     // Функция для изменения статуса задачи
30     function updateStatus(uint _taskId, TaskStatus _status) public {
31         tasks[_taskId].status = _status;
32     }
33
34     // Функция для получения информации о задаче
35     function getTask(uint _taskId) public view returns (Task memory)
36     {
37         return tasks[_taskId];
38     }

```

В этом примере:

- TaskStatus - перечисление, которое определяет возможные статусы задачи.
- Task - структура, которая содержит описание задачи и её статус.
- TaskManager - контракт, который использует массив tasks для хранения задач и предоставляет функции для управления этими задачами.

Этот пример демонстрирует, как структуры и перечисления могут быть использованы для создания чёткой и упорядоченной структуры данных в смарт-контрактах. Используйте этот код как основу для разработки своих контрактов с применением структур и перечислений.

Задание 9: создание и использование событий

Цель: освоить механизм событий в Solidity для взаимодействия контракта с внешним миром.

Задача: определить одно или несколько событий в контракте и вызвать их в различных функциях, демонстрируя, как события могут быть использованы для отслеживания важных изменений и действий.

Ответ:

События в Solidity играют ключевую роль в логировании и информировании о действиях, происходящих внутри смарт-контрактов. Вот пример контракта, который демонстрирует использование событий:

```

1 pragma solidity ^0.8.0;
2
3 contract EventExample {
4     // Объявление события для логирования изменений статуса задачи
5     event TaskStatusChanged(uint taskId, string newStatus);
6
7     // Структура для задачи
8     struct Task {
9         string description;
10        string status;
11    }
12
13    // Массив для хранения задач
14    Task[] public tasks;
15
16    // Функция для добавления задачи, которая также вызывает событие
17    function addTask(string memory _description) public {
18        tasks.push(Task({description: _description, status:
19            "ToDo"}));
20        // Вызов события после добавления задачи
21        emit TaskStatusChanged(tasks.length - 1, "ToDo");
22    }
23
24    // Функция для изменения статуса задачи, которая также вызывает
25    // событие
26    function updateTaskStatus(uint _taskId, string memory
27        _newStatus) public {
28        tasks[_taskId].status = _newStatus;
29        // Вызов события после изменения статуса задачи
30        emit TaskStatusChanged(_taskId, _newStatus);
31    }
32 }
```

В этом примере:

- event TaskStatusChanged - событие, которое логирует изменение статуса задачи.
- addTask - функция для добавления новой задачи в массив tasks и вызова события TaskStatusChanged.
- updateTaskStatus - функция для изменения статуса задачи и вызова события TaskStatusChanged.

События могут быть отслежены внешними слушателями, что позволяет клиентским приложениям реагировать на изменения в контракте. Это делает события мощным инструментом для создания интерактивных и реактивных децентрализованных

приложений. Используйте этот код как основу для включения событий в ваши смарт-контракты.

Задание 10: взаимодействие контрактов

Цель: понять, как различные контракты могут взаимодействовать и вызывать функции друг друга.

Задача: создать два контракта, где один будет вызывать функцию другого. Это поможет понять, как контракты могут работать вместе в более сложных системах.

Ответ:

Взаимодействие контрактов в сети Ethereum позволяет создавать сложные децентрализованные приложения. Вот пример двух контрактов, где один контракт вызывает функцию другого:

```
1 pragma solidity ^0.8.0;
2
3 // Контракт, который будет вызываться
4 contract CalledContract {
5     uint public storedData;
6
7     function set(uint x) public {
8         storedData = x;
9     }
10 }
11
12 // Контракт, который будет вызывать функцию другого контракта
13 contract CallingContract {
14     CalledContract public calledContract;
15
16     constructor(address _calledContractAddress) {
17         calledContract = CalledContract(_calledContractAddress);
18     }
19
20     function callSetFunction(uint _x) public {
21         calledContract.set(_x);
22     }
23 }
```

В этом примере:

- `CalledContract` содержит функцию `set`, которая позволяет изменять значение переменной `storedData`.
- `CallingContract` содержит ссылку на `CalledContract` и функцию `callSetFunction`, которая вызывает функцию `set` контракта `CalledContract`.

Чтобы использовать `CallingContract` для вызова функции в `CalledContract`, вам нужно сначала задеплоить `CalledContract`, получить его адрес, а затем использовать этот адрес при создании `CallingContract`. Это позволяет `CallingContract` вызывать функцию `set` и изменять состояние `CalledContract`.

Этот пример иллюстрирует основы взаимодействия контрактов в Ethereum и может служить отправной точкой для создания более сложных систем, где контракты взаимодействуют друг с другом.

Итоги урока:

На этом уроке мы не только узнали:

- Особенности языка программирования Solidity, области применения, его преимущества и недостатки.
- Особенности работы с интерфейсом Remix.
- Конструкции и синтаксис языка, правила написания смарт-контрактов.

А также решили множество задач, которые послужат крепкой основой для дальнейшего изучения языка.

Использованная литература

<https://soliditylang.org/>