



# Blockchain School

# Урок 2. Пишем контракт

*Разбираемся с нюансами синтаксиса Solidity, пишем свой контракт и рассматриваем возможные проблемы безопасности.*

## План

1. Remix IDE
2. Невыполненные транзакции
3. Payable functions
4. Откат изменений
5. События в контрактах
6. Storage контракта
7. Константные и обычные функции
8. Private и ограничение на получение данных
9. Верификация контракта
10. Value overflow
11. Документация по Solidity
12. Задания

# 1. Remix IDE

[Remix](#) – это самая популярная IDE для разработки умных контрактов.

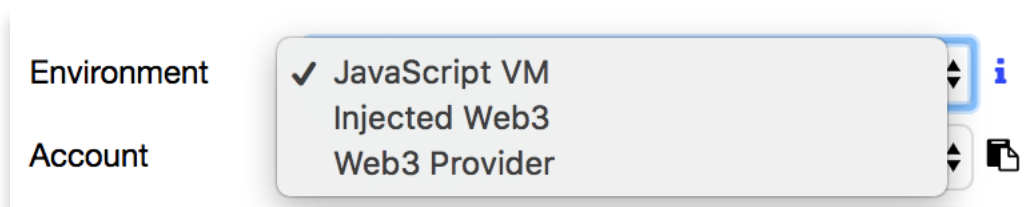
Поддерживает довольно много функций:

- Подключение к указанному RPC провайдеру
- Компиляция кода в байткод / опкоды
- Публикация в Github gist
- Пошаговый дебагер
- Подсчет стоимости исполнения функций в газе
- Сохранение вашего кода в localStorage и др.

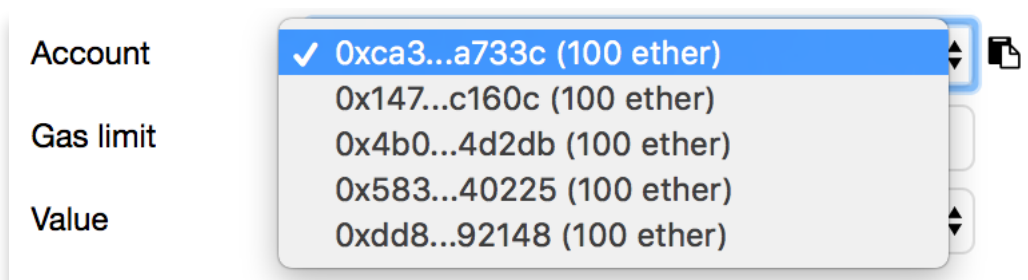
[Больше о его функциях](#)

Мы используем Remix, чтобы эмулировать поведение контракта.

В Environment мы можем выбирать, с какой сетью работать. По умолчанию мы будем работать с JavaScript VM. Но если установить [MetaMask](#)<sup>1</sup>, можно работать с Injected Web3.



Мы будем работать с JavaScript VM, в ней уже есть сгенерированные аккаунты с эфиром на счету. Вам нужно выбрать, с какого аккаунта<sup>2</sup> (эфирного адреса) вы создаете контракт и с какого вызываете функцию.



<sup>1</sup> - Менеджер кошельков, который интегрируется в интерфейс браузера Google Chrome

<sup>2</sup> - Аккаунтом может быть как кошелек, так и контракт

# 1. Remix IDE

Скопируйте этот контракт и вставьте его в Remix:

```
pragma solidity 0.4.19;
contract EthereumCourse {

    mapping (address => bool) public voted;

    string public poll = "Is this course hard or easy for you?";
    string[] public votes;

    event Vote(address voter, string answer);
    event Error(string message);

    function vote(string _answer) public payable returns(bool) {
        if(voted[msg.sender]) {
            Error("You've already voted");
            return false;
        }
        voted[msg.sender] = true;
        votes.push(_answer);
        Vote(msg.sender, _answer);
        return true;
    }
}
```

Когда у вас есть написанный контракт, его нужно задеплоить с одного из аккаунтов. Если у вас написано несколько контрактов, то в выпадающем списке можно выбрать, какой именно контракт вы хотите задеплоить. В нашем случае пока что есть один контракт.

## 1. Remix IDE



The screenshot shows a section of the Remix IDE interface. At the top, there is a dropdown menu with the text 'EthereumCourse' and a small upward/downward arrow icon on the right. Below this dropdown is a light gray rectangular input field. To the right of the input field is a pink button with the text 'Create' in black.

Чтобы создать контракт, нажимаем Create – и получаем интерфейс взаимодействия с контрактом:



The screenshot shows the contract interaction interface in Remix IDE. At the top, there is a header bar with a dropdown menu showing 'EthereumCourse at 0xbbf...732db (memory)' and a file icon. Below the header, there are four rows of interaction options, each with a colored button and a corresponding input field:

- A blue button labeled 'votes' next to a text input field containing 'uint256'.
- A blue button labeled 'voted' next to a text input field containing 'address'.
- A blue button labeled 'poll' next to an empty text input field.
- A pink button labeled 'vote' next to a text input field containing 'string\_answer'.

### Внимание!

Значения в поля нужно писать в кавычках, если это не цифры, а строка или адрес. Каждый раз, когда вы вносите изменения в код, не забывайте создавать контракт заново. Интерфейс не меняется автоматически.

# 1. Remix IDE

Теперь вы можете вызывать функции и просматривать значения переменных, чтобы понять выполняет ли код то, что вы от него ожидаете.

В Remix есть очень удобная консоль, в которой можно посмотреть детали транзакции:

**Status** – Статус транзакции – выполнялась она или нет;

**From** – Адрес, который вызвал транзакцию;

**To** – Функция контракта, которую вызвали;

**Gas** – Лимит газа;

**Transaction cost** – Количество газа, потраченное на транзакцию;

**Hash** – Хэш транзакции;

**Input** – Данные, которые были переданы в функцию (в hex кодировке);

**Decoded input** – Переданные данные в оригинальном представлении;

**Decoded output** – Данные, которые вернула функция;

**Logs** – Ивенты, которые были заэмичены (emitted events);

**Value** – Количество переданного эфира.

## 2. Невыполненные транзакции

Давайте на секунду отойдем от Remix и посмотрим, что происходит в [Etherscan.io](https://etherscan.io).

В нем иногда появляются транзакции с восклицательными знаками. Это транзакции, которые не были выполнены.

0xcb7890015f7310...	4761093	37 secs ago	0x0959f5e8799bde...	→	CryptoKittiesSalesA...	0 Ether
0x4780333baa3e93...	4761093	37 secs ago	0x10d09da2fd14fd...	→	CryptoKittiesCore	0.008 Ether

Условия, при которых не выполнится транзакция:

- Недостаточно газа, чтобы оплатить выполнение транзакции;
- Отправляем эфир в не-payable функцию;
- Обращаемся к функции, которой нет;
- Вызываем функцию, которая кидает ошибку.

### 3. Payable functions

Если мы зададим нашей функции параметр **payable**, то сможем отправлять в контракт эфир.

```
function vote(string _answer) public payable returns(bool) {  
    if(voted[msg.sender]) {  
        return false;  
    }  
    voted[msg.sender] = true;  
    votes.push(_answer);  
    Vote(msg.sender, _answer);  
    return true;  
}
```

Если вы будете пытаться отправить эфир в **не-payable** функцию, транзакция не выполнится. В **payable** функцию можно отправлять 0 эфира.

## 4. Откат изменений

В Solidity есть команда **revert()**, которая позволяет откатить все изменения, которые произошли за время выполнения функции. Если мы отправляем деньги в контракт, но функция завершается **return false**, то деньги все же снимаются со счета отправителя и попадают в контракт. Изменения, которые произошли до **false**, тоже останутся. А если функция завершается **revert()**, то все изменения откатываются и деньги не попадают в контракт.

Этой функцией не стоит злоупотреблять, потому что она не говорит, в каком именно месте произошла ошибка. Если вы используете **true/false**, то перед **false** можете вывести сообщение, из-за чего эта ошибка.

```
event Vote(address voter, string answer);
event Error(string message);

function vote(string _answer) public payable returns(bool) {
    if(voted[msg.sender]) {
        Error("You've already voted");
        return false;
    }
}
```

Вашим заданием было проголосовать, чтобы оценить курс. Вот так будет выглядеть консоль лог, если мы попытаемся проголосовать с одного адреса дважды:

decoded output	{ "0": "bool: false" }
logs	[ { "topic": "08c379a0afcc32b1a39302f7cb8073359698411ab5fd6e3edb2c02c0b5fba8aa", "event": "Error", "args": [ "You've already voted" ] } ]



## 4. Откат изменений

В decoded output мы видим, что функция выполнялась с булевым значением **false** и что у нас есть ивент с названием "Eggo" и строкой "You've already voted".

А теперь давайте заменим **false** на **revert()**

```
function vote(string _answer) public payable returns(bool) {  
    if(voted[msg.sender]) {  
        revert();  
    }  
}
```

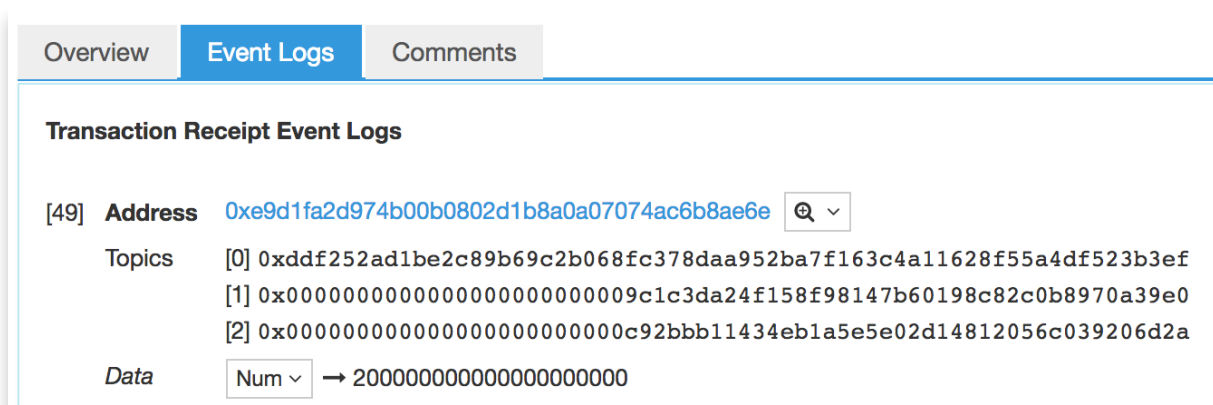
Вы увидите, что покажет консоль, если дважды проголосовать с одного и того же адреса.

## 5. События в контрактах

Ошибку мы вывели с помощью ивента.

Ивенты помогают собирать информацию о том, что происходит в контракте. Обычно мы пишем ивенты после каждого события, которое меняет данные в storage: пользователь отправил деньги, пользователь снял деньги, пользователь сменил данные и т.д. Также очень удобно с их помощью обрабатывать ошибки.

Ивенты можно просматривать не только в консоли, но также в Etherscan.io. Достаточно посмотреть на транзакцию, чтобы понять, что же произошло во время выполнения контракта.



The screenshot shows the 'Event Logs' tab for a transaction on Etherscan.io. The title is 'Transaction Receipt Event Logs'. It displays a single event at index [49]. The 'Address' field shows the contract address: 0xe9d1fa2d974b00b0802d1b8a0a07074ac6b8ae6e. The 'Topics' field lists three topics in hexadecimal: [0] 0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef, [1] 0x00, and [2] 0x00. The 'Data' field shows a value of 2000, with a 'Num' dropdown menu set to 'Num'.

Это данные в 16-ричной кодировке, которые были выведены в ивенте.

Если мы не будем писать ивенты, то отслеживать изменения в контракте будет гораздо сложнее. Нам нужно будет зайти в storage, подумать о том, какие данные могли измениться, посмотреть на них, вспомнить, какими были предыдущие параметры, чтобы узнать изменились ли данные. Ивенты очень упрощают жизнь.

## 6. Storage контракта

Когда вы объявляете переменные в контракте, вы записываете данные в storage контракта. Каждая запись в storage стоит определенное количество газа, которое зависит от типа данных. С этим вы можете разобраться сами, поигравшись в Remix, объявляя какие-то переменные, записывая в них данные и запуская контракт. В консоли будет показано, сколько газа вы потратили.

### Что такое Mapping?

В нашем контракте с голосованием есть такая вот запись:

```
mapping (address => bool) public voted;
```

В нашем случае, маппинг – это хранилище под названием voted, в котором есть ключ (address) и значение (bool). В этом маппинге у нас хранятся адреса всех, кто проголосовал. Когда какой-то адрес вызывает функцию vote, его адрес помещается в маппинг и ему присваивается значение true. Таким образом, к определенному адресу привязывается информации, проголосовали ли с этого адреса или нет.

```
voted[msg.sender] = true;
```

Таким образом, мы можем ввести в интерфейс интересующий нас адрес и узнать проголосовал он уже или нет.

voted

"0x4b0897b0513fdc71"

0: bool: true

**Не забывайте – значения нужно вводить в кавычках!**

## 6. Storage контракта

Можно делать маппинги с разными значениями:

```
mapping (address => unit)
mapping (unit => string)
```

Важно помнить, что когда Solidity принимает в качестве ключа строку (string), то он ее хеширует, поэтому ключом будет хеш. Также, можно делать вложенные маппинги, то есть размещать один маппинг в другом:

```
mapping (address => mapping(unit => bool))
```

В этом случае адрес выступает ключом, а вложенный маппинг – значением.

[Вот тут можно почитать подробнее](#)

## 7. Константные и обычные функции

Когда вы объявляете функцию, вы можете прописывать разные модификаторы, которые ограничивают доступ к ней, или указывать, какие у нее есть возможности. Вы можете писать собственные модификаторы и можете пользоваться уже встроенными в язык.

**Например:**

**view** (от **constant**) – функция не меняет состояние контракта, но она может делать вызовы в другие контракты, может читать из storage и т.д. Но писать в storage не должна, хоть и может.

**pure** – не пишет в storage, не читает из storage, не меняет ничего, просто реализовывает какую-то внутреннюю логику.

Если не прописать такой модификатор, то функция сможет и писать в storage, и читать из него. Если результат работы вашей функции не меняет состояние storage, то Remix предложит добавить ей модификатор **view**. Если она совсем не взаимодействует со storage, то Remix предложит добавить модификатор **pure**.

## 8. Private и ограничение на получение данных

Мы поговорили об ограничениях доступа для функций, а теперь поговорим об ограничениях доступа для пользователей.

Если вы ставите модификатор **public** для переменной или для функции, значит вы считаете, что эти данные должны быть общедоступными. По умолчанию все данные и так **public**, но лучше это указывать явно и вообще всегда выставлять модификаторы доступа. Когда вы своими руками прописываете модификаторы, все делаете это осознанно. Поэтому не будет ситуации, когда вы по невнимательности оставили данные в публичном доступе.

Когда вы ставите модификатор **private**, то функцию или переменную нельзя будет увидеть в интерфейсе Remix, Etherscan.io или MyEtherWallet. При этом storage контракта – это открытая информация и private переменные можно увидеть через него.

Также, для функций можно выставлять модификатор **internal** – это значит, что функцию нельзя вызывать извне, она выполняет какую-то внутреннюю логику. Хорошим тоном считается называть **internal** функции, начиная с нижнего подчеркивания: **\_internalFunction()**.

Отличие private модификаторов от internal в том, что private функции и переменные не наследуются, а internal – наследуются.

## 9. Верификация контракта

Когда вы деплоите контракт в сеть, вы можете верифицировать его или можете этого не делать. Чтобы верифицировать контракт, вам достаточно просто показать код всем в сети. По умолчанию, когда вы деплоите контракт в сеть, вы этого не делаете. Но у вас есть такая возможность. Зайдите на Etherscan.io и посмотрите, сколько всего там контрактов и сколько из них верифицированы.

Вы можете задаться вопросом, почему такой большой разрыв? Мы же тут все честные и все такое, тогда почему почти никто не верифицирует свои контракты?

Во-первых, если в вашем коде все же есть уязвимость, то злоумышленникам будет куда проще ее найти и использовать, когда ваш код открыт. Очень мало людей честно напишут вам *“Ой, ребята, у вас уязвимость, можно украсть все деньги”*. Их просто возьмут и молча украдут.

Во-вторых, компании платят своим программистам за то, что те пишут контракты. Поэтому им не выгодно показывать всем свой код, чтобы другие могли бесплатно пользоваться их разработками. Это касается не только контрактов: мало компаний, которые выкладывают код своего продукта в открытый доступ. Это бизнес, детка.

## 10. Value overflow

Когда вы работаете с числами, вы должны постоянно помнить про особенность типа данных **uint**. В этот тип данных помещается  $((2^{256}) - 1)$  число.

//uint -> uint256 -> unsigned integer 256 -> from 0 to  $((2^{256}) - 1)$

Число  $2^{256}$  эквивалентно нулю.

// $(2^{256}) == 0$

Вставьте в Remix вот этот код:

```
pragma solidity 0.4.19;

contract Overflow {
    function sub(uint _a, uint _b) pure public returns (uint) {
        return _a - _b;
    }
}
```

И попробуйте, к примеру, от нуля отнять десять. Это будет то же самое, что отнять 10 от числа  $(2^{256})$ .

// $0 - 10 = (2^{256}) - 10$

### Внимание!

Допустим, у вас есть такая операция:

**0 - 10 + 20**

Может показаться, что лучший способ избежать overflow, когда мы от меньшего числа отнимаем большее, – это всегда выдавать ошибку. Но это неправильный подход. Математические операции, особенно финансовые, часто состоят не из одного действия. Если в какой-то момент операции у нас получается отрицательное число – это нормально. Нужно проверять итоговое число. Поэтому в нашей операции все в порядке, нет никакого overflow.

Вам необходимо будет придумать, как обойти overflow в задаче, которая дана в конце урока.

## 11. Документация по Solidity

Документация должна стать вашим лучшим другом, вашим спутником и любовницей на время курса. Вы найдете ее по ссылке – <http://solidity.readthedocs.io/en/develop/>

Документация довольно короткая. Ее легко прочесть за несколько дней, если каждый день уделять этому занятию хотя бы по 2 часа. Рекомендую читать только на английском, потому что перевод ужасен и многие вещи поданы слишком сложно.

Также, вам стоит почитать рекомендации для Solidity разработчиков <https://consensys.github.io/smart-contract-best-practices/>

### Вопрос:

#### Можно ли записать сеть своими транзакциями?

Вообще можно. Но минимальная стоимость любой транзакции в сети – 21000 газа. Если у вас достаточное количество эфира и вы готовы потратить его, то вы, конечно, можете создать множество транзакций. Но это не имеет особого смысла, потому что другие участники сети будут видеть ваши дешевые транзакции и думать: *“Ага, я просто поставлю цену за газ больше и моя транзакция выполнится быстрее”*. Поэтому план записать сеть, скорее всего, провалится.



## 12. Задания

### Сделать контракт учета off-chain долгов

#### Протокол займа:

1. Вы встречаетесь с человеком, чтобы дать ему займ;
2. Он отправляет транзакцию на ваш контракт и указывает сумму займа;
3. Вы проверяете, что в контракте действительно появилась такая сумма;
4. Вы передаете деньги.

#### Протокол возврата:

1. Вы встречаетесь с человеком, который хочет вернуть займ;
2. Вы отправляете транзакцию на ваш контракт и указываете сумму возврата;
3. Он проверяет, что в контракте действительно сумма его займа уменьшилась на правильную сумму;
4. Он передает вам деньги.

Делаем это все в <https://remix.ethereum.org> и там же проверяем.

Домашнее задание добавляйте в ответы прямо на сайте курса, чтобы мы ничего не пропустили. По желанию, можете залить готовый код на github и кинуть ссылку в телеграм, чтобы все желающие могли проверить, все ли правильно, и указать на возможные проблемы.

### Внимание!

**Если что-то не работает, а должно, проверьте ставите ли вы кавычки, когда общаетесь с интерфейсом контракта!**

Задавайте вопросы в нашем [Телеграм-чате!](#)

#### Полезные ссылки:

Live Ether Camp – <https://live.ether.camp>

<https://rawgit.com/Ambisafe/etoken-lib/master/utils/kovan.html>

<https://github.com/ethereum/solidity/releases>

<https://github.com/ethereum/wiki/wiki/JavaScript-API>

<https://solidity.readthedocs.io>