Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Воронежский государственный технический университет»

Кафедра инноватики и строительной физики
им. профессора И.С. Суровцева

БЛОКЧЕЙН - ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к выполнению лабораторных работ
по дисциплине «Blockchain - технологии»
для студентов направления 27.03.05 «Инноватика»
(профиль «Инновационные технологии») всех форм обучения

Воронеж 2021

УДК 336.74:004
ББК 65.050.253

Составители:
канд. техн. наук Д.В. Сысоев,
докт. физ.-мат. наук П.А. Головинский

Блокчайн - технологии: методические указания к выполнению лабораторных работ по дисциплине «Blockchain - технологии» для студентов направления 27.03.05 «Инноватика» (профиль «Инновационные технологии») всех форм обучения / ФГБОУ ВО «Воронежский государственный технический университет»; сост.: Д.В. Сысоев, П.А. Головинский. Воронеж: Изд-во ВГТУ, 2021. 25 с.

Предназначены для проведения лабораторных работ по дисциплине «Blockchain - технологии» для студентов направления 27.03.05.

Методические указания подготовлены в электронном виде и содержатся в файле LabRab-BchT.pdf.

Ил. 47. Библиогр.: 5 назв.

УДК 336.74:004
ББК 65.050.253

Рецензент – Н.В. Акамсина, канд. техн. наук, доцент кафедры систем управления и информационных технологий в строительстве

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

Лабораторная работа №1

Знакомство с Remix - web-средой Solidity IDE.

Цель работы: изучить и закрепить на практике возможности среды разработчика смарт- контрактов Remix.

Результат: практические навыки работы с инструментами среды разработчика смарт- контрактов Remix.

Теоретическая справка:

Обзор среды Remix. Откроем Remix по адресу: http://remix.ethereum.org

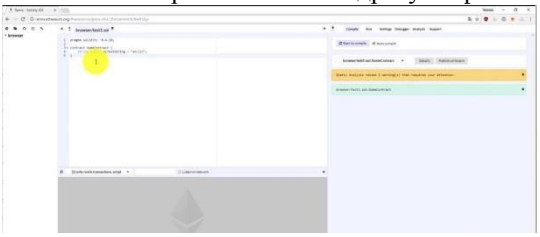


Рис.1.

Remix - это интегрированная среда разработки, запускаемая непосредственно в браузере. Она поддерживает весь функционал, обычно ожидаемый от сред разработки, отличается тем, что работает в браузере, а также снабжена механизмом эмуляции блокчейна, в котором можно запускать распределенные приложения или контракты Solidity (смарт- контракты), а также проводить их отладку и тестирование.

Как добавлять новые и просматривать существующие файлы в среде Remix. Обратимся к левой стороне Remix.

Рис.2.

Здесь расположен файловый менеджер, с помощью которого можно создавать новые файлы, переименовывать их, после чего они становятся доступными.

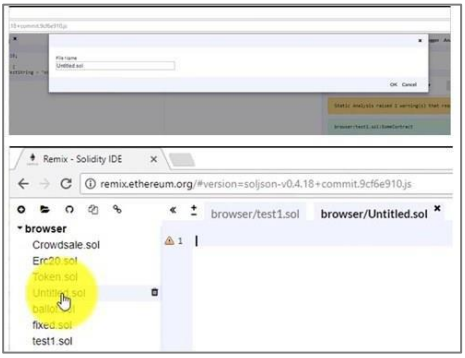


Рис.3.

В главном окне после этого можно редактировать файлы. Закроем это окно и удалим файл.

Рис.4.

Можно также добавлять файлы с компьютера, либо напрямую импортируя их, либо публикуя на общедоступном ресурсе.

Рис.5.

Кроме того, можно скопировать все файлы в другой браузер Solidity, а также подключиться к локальному хосту.

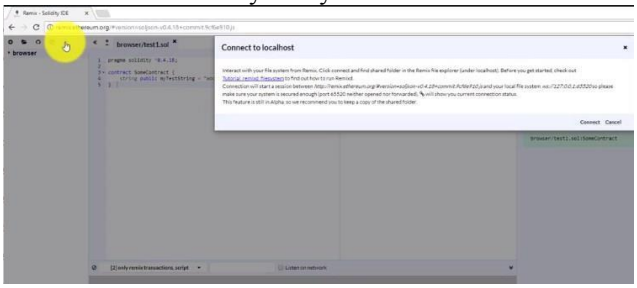


Рис.6.

Главное окно редактирования. В центре расположено основное окно для редактирования кода, позволяющее добавлять необходимые функции.

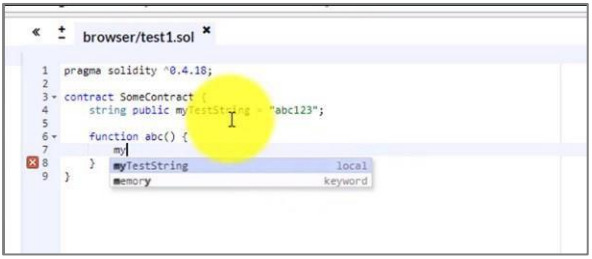


Рис.7.

Кроме того, поддерживается функция автоматического дополнения - после набора нескольких символов система предлагает подсказки, например, что *myTestString* - это переменная.

Информация о контрактах на вкладке Compilation. В правой части находятся несколько опций для взаимодействия со смарт-контрактом и получения информации о нем. Сверху расположены вкладки для компиляции (Compile), запуска (Run), изменения настроек (Settings), отладки (Debugger), анализа (Analysis) и получения поддержки (Support).

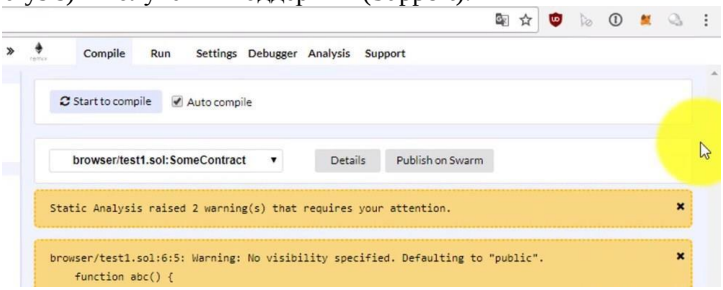


Рис.8.

Вкладка Compile. В большинстве случаев при работе в среде Remix действует настройка автоматической компиляции Auto-Compile - код компилируется по мере ввода текста. Чуть ниже приводятся результаты статического анализа кода (подробнее с ними можно ознакомиться на вкладке Analysis).

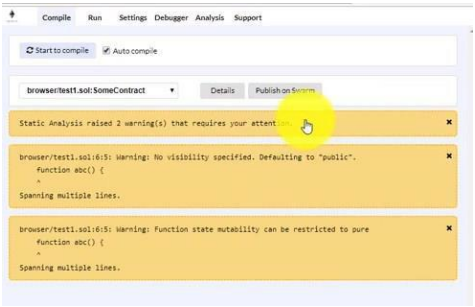


Рис.9.

Например, для этой функции не объявлена область видимости, кроме того, она вообще не содержит инструкций и может быть превращена в так называемую *чистую* функцию.

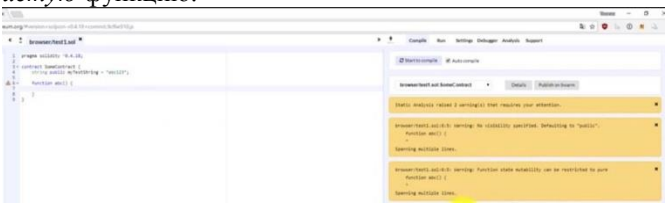


Рис.10.

В случае, если в файле Solidity описано более одного контракта, здесь можно выбрать нужный контракт

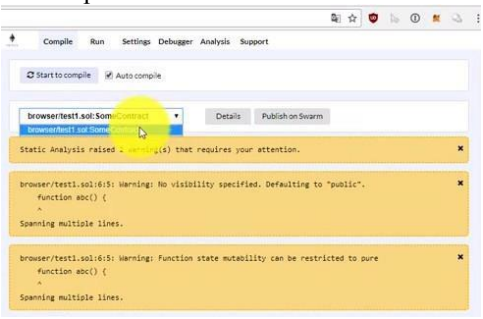


Рис.11.

и получить подробную информацию о нем: название контракта, метаданные, байткод - так что видно, что контракт действительно компилируется в браузере, и можно получить доступ к байткоду.

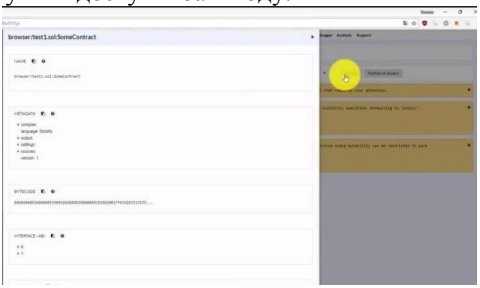


Рис.12.

Кроме того, доступен двоичный интерфейс для приложений (API), можно скопировать участок кода в буфер обмена и использовать его для взаимодействия с контрактом посредством Mist или другого веб-кошелька для среды Ethereum.

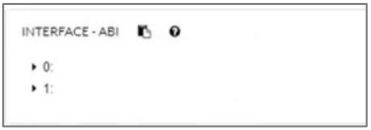


Рис.13.

Также можно использовать этот код Web3 для непосредственной отправки контракта в Geth, и просматривать множество других сведений, например, коды операций или исполняемый байткод.

Все, что нужно знать о запуске файлов Solidity в среде Remix. Вкладка Run. Это опция выбора среды запуска; виртуальная машина Java будет обеспечивать эмуляцию среды блокчейна прямо в браузере.

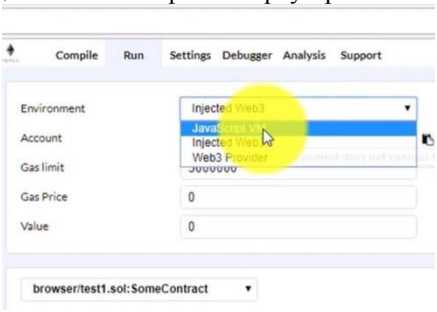


Рис.14.

В эмулированной среде можно использовать несколько счетов, с сотней единиц эфира на каждом, и, конечно, создавать, размещать и запускать контракты.

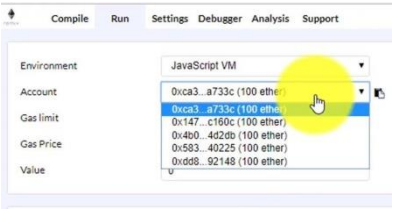


Рис.15.

Все они будут работать в эмулированной среде блокчейна, то есть не будет подключения к MetaMask или реальному блокчейну - все происходит только в браузере.

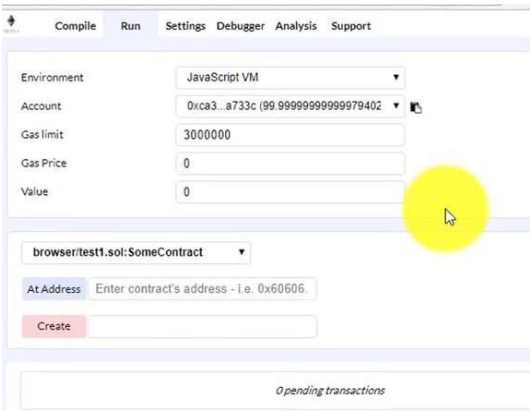


Рис.16.

Если закрыть и открыть браузер, вся информация будет потеряна.

Для каждой транзакции, предусмотренной контрактом, можно установить свой предел расхода газа. Например, в функции *abc* (сделаем ее публичной) переменная *myTestString* меняет свое значение с "abc123" на "bbbccc". Теперь опубликуем контракт.

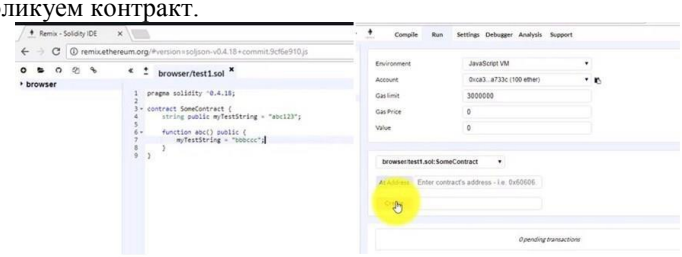


Рис.17.

Видно, что переменная *myTestString*. имеет значение "abc123".

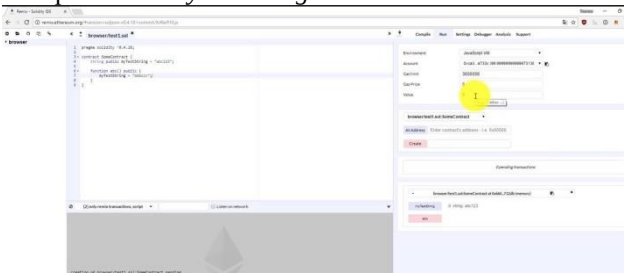


Рис.18.

Затем можно вызвать функцию *abc*, щелкнув по ее названию. Для конкретной транзакции можно установить предел расхода газа, цену за единицу газа, а также отправить вместе с транзакцией некоторое количество валюты, например, единицу эфира.

Журнал событий в Remix

Выполним вызов функции. При вызове функции в нижней части окна отображаются все задействованные транзакции.

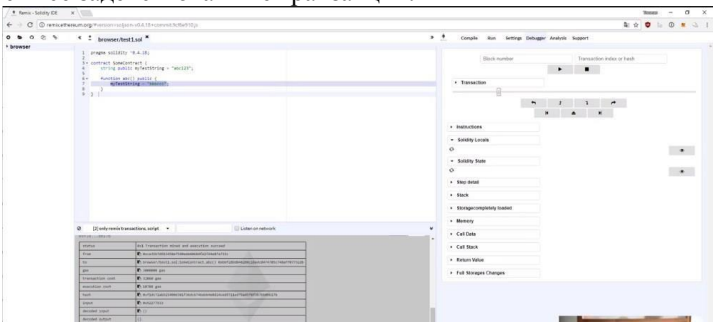


Рис.19.

Этот функционал особенно удобен при работе со сложными функциями из большого числа шагов или циклов.

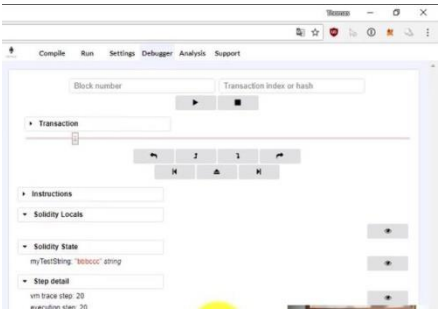


Рис.20.

На этапе отладки можно узнать, что последняя операция кода не была выполнена или что кое-где необходимо заменить знак "меньше или равно" знаком "больше или равно" до отправки транзакции в реальный блокчейн.

Таковы типичные случаи применения отладчика. Иногда приходится иметь дело с обработкой исключений, в этих случаях отладчик позволяет быстрее установить участок возникновения исключения.

Краткий обзор инструментов для статического анализа. Вкладка Analysis содержит инструменты статического анализа.

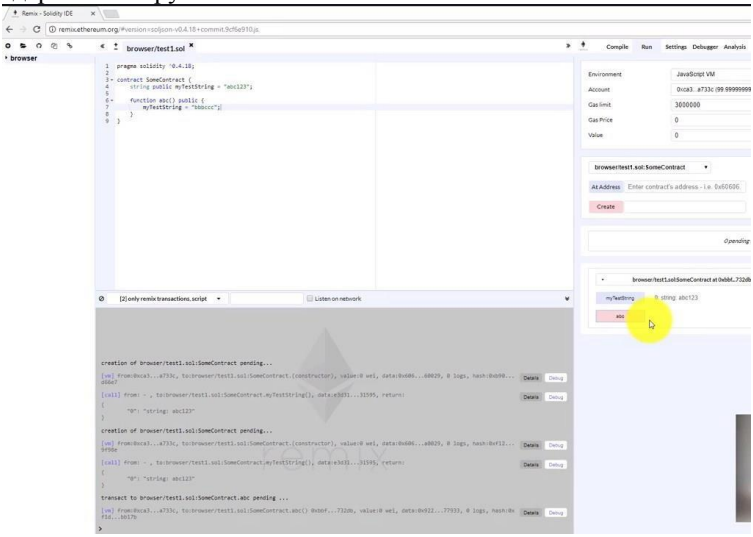


Рис.21.

Сейчас транзакция ожидает обсчета, а затем она будет обработана сетью блокчейна. Также можно получить больше информации о транзакции. Для каждой конкретной транзакции можно увидеть, с какого адреса и на какой она была отправлена. Напомним, что сейчас все транзакции выполняются в эмулированной среде блокчейна, и существуют только внутри этого браузера. Можно увидеть установленный предел расхода газа. Также видны отправляемые в транзакции данные, которые могут взаимодействовать со смарт-контрактом по этому адресу.

Рис.22.

Инструментарий для отладки. В среде Remix отладчик (вкладка Debugger) поддерживает последовательную работу по шагам.

Рис.23.

Очень удобно иметь отладчик, включенный непосредственно в код. Можно устанавливать точки прерывания или воспользоваться кнопкой отладки, чтобы увидеть, как проходит транзакция после отправки в сеть блокчейна.

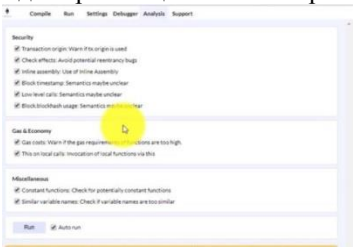


Рис.24.

При статическом анализе код просматривается на предмет ошибок описания рабочей среды, и, в случае их нахождения, выводятся предупреждения. Например, для каких-то функций не описаны требования по расходу газа, или объявлены неиспользуемые переменные. В качестве предупреждений могут выводиться сообщения о том, что код использует временную метку блока, например, now, основанную на временной метке блока майнера, вместо временной метки блока из блокчейна.

Таковы типичные предупреждения, выдаваемые инструментарием для статического анализа. Если вы уверены в своих действиях, их можно игнорировать, но зачастую они полезны при указании на опечатки или использование циклов, когда без них можно обойтись, поэтому в общем случае статическим

анализом не стоит пренебрегать.

Помощь и поддержка. На **вкладке Support** можно найти прямые ссылки на чат сообщества Remix, где можно пообщаться на тему Remix или задать вопросы участникам этого сообщества.

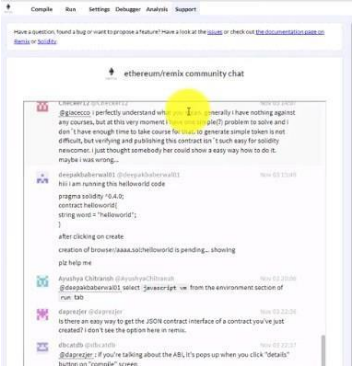


Рис. 25.

Новый интерфейс окна среды Remix выглядит следующим образом:

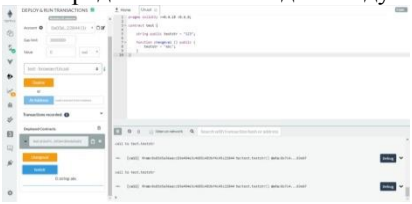


Рис.26.

Задание

1. Ознакомьтесь с возможностями, режимами работы и инструментами IDE Remix.
2. Для эксперимента используйте следующий простой смарт контракт:

```
pragma solidity >=0.4.18 <0.6.0; contract test {
string public teststr = "123";
function changeval () public { teststr = "abc";
}
}
```

Контрольные вопросы

1. Какие инструменты IDE Remix Вы знаете?
2. Что такое газ в Ethereum и зачем он нужен?
3. Что представляет собой смарт-контракт?

Лабораторная работа №2
Язык программирования смарт-контрактов Solidity

Цель работы: изучить и закрепить на практике основные синтаксические конструкции языка программирования Solidity.

Результат: файлы с примерами простых контрактов согласно задания.

Теоретическая справка:

Эта работа полностью посвящена изучению среды программирования Solidity. Однако перед тем, как погрузимся в ее изучение, давайте поговорим о том, какого рода распределенное *приложение* нам предстоит написать. В ходе этого курса будем создавать распределенный веб-кошелек с высокой степенью надежности. Его *надежность* обеспечена самой структурой блокчейна. Распределенность же означает, что к кошельку возможен *доступ* нескольких пользователей, однако распоряжаться хранящимися в кошельке средствами можно только при наличии соответствующего доступа.

Как же работает распределенный кошелек? У такого кошелька есть некоторый баланс, и несколько пользователей могут подключаться к нему и отправлять средства на другие счета. Эти *операции* отслеживаются через транзакции в блокчейне. Рассмотрим подробнее на примере. В нашем кошельке пять тысяч единиц эфира, и *адрес 0xdeadbeef1* отправляет *запрос* на перевод одной тысячи единиц эфира на другой *адрес 0xdeadbeef2*. Кошелек должен будет проверить, что отправитель уполномочен распоряжаться хранящимся в кошельке эфиром. Если такие *права* есть, то на *адрес 0xdeadbeef2* отправится перевод тысячи единиц эфира. Кроме того, это событие будет зарегистрировано, о транзакции будет составлен отчет.

Для программирования данного примера в блокчейне будем использовать *язык программирования* высокого уровня Solidity. Solidity очень похож на JavaScript, но есть и различия. То, что в других языках называется классами, в среде Ethereum называется контрактами. Контракт может состоять из нескольких функций. Еще есть события, описывающие историю работы распределенного приложения, а также другие структуры, такие как библиотеки. Наш сравнительно простой распределенный веб-кошелек будет состоять из нескольких функций. Две функции будут реализовывать *управление учетными записями* пользователей - одна позволит добавлять адреса, уполномоченные распоряжаться эфиром в кошельке, а другая - отзывать *доступ* к кошельку. Специальная *функция* будет позволять отправлять эфир из кошелька другим пользователям сети; кроме того, должен быть и способ получать эфир, это будет реализовано так называемой функцией запасания эфира. Еще кошелек будет регистрировать и отправлять события трех видов. Одно событие будет срабатывать при отправке эфира, второе - при редактировании авторизованного на распоряжение эфиром адреса, а третье

- при отзыве у такого адреса прав.

Классы контрактов, функций и условий

В этой работе будут раскрыты следующие вопросы: как создавать простые контракты, как выглядят структура и функции контракта, а также как

использовать условия в среде Solidity. Начнем с простого контракта. Понятие контракта в среде Solidity похоже на понятие класса в других языках программирования. Контракт задается словом *contract*, а затем ему необходимо дать имя. Пусть это будет *SimpleDapp*. Итак, структура простого контракта - это имя, *служебное слово contract*, и логика контракта, заключенная в *фигурные скобки*.

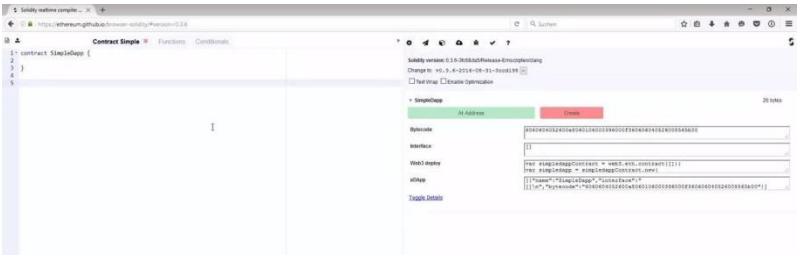


Рис.27.

Обратите внимание, что в этом же файле хранится еще один контракт.

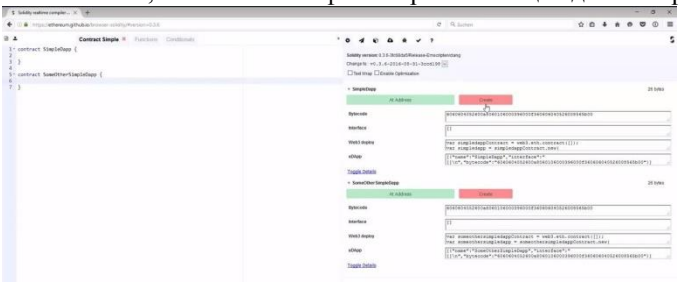


Рис.28.

В правой части браузера Solidity видно, что *компиляция* и *отладка* контрактов происходят *по* мере их написания, что очень удобно. Можно создавать контракт. Ему сразу задается определенный *адрес*. Наш контракт снабжен только функцией запасания эфира, и не умеет ничего другого. Обратите внимание, что раздел с интерфейсным кодом пуст.

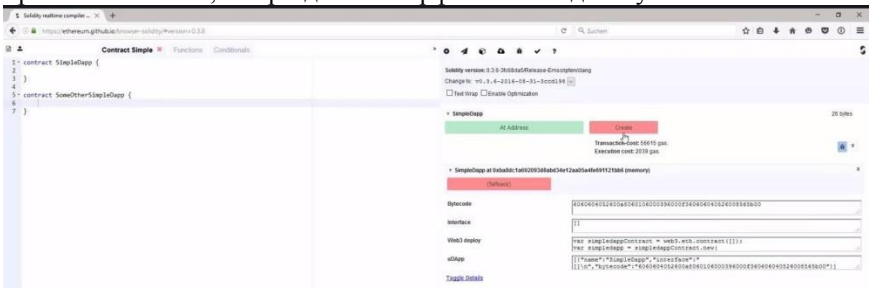


Рис.29.

Добавим в наше *приложение* переменную типа *uint*.

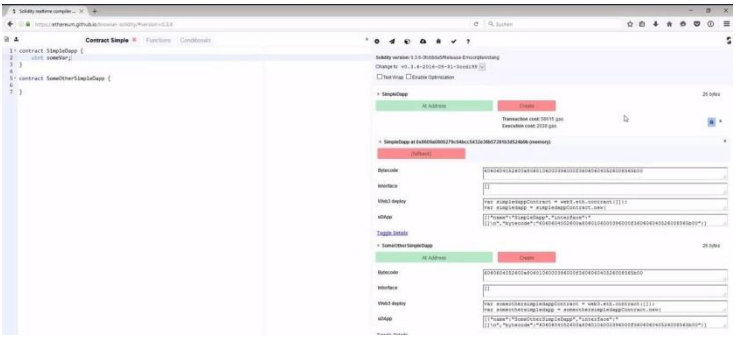


Рис.30

Если обратить внимание на раздел создания контракта в правой части браузера, можно заметить, что *доступ* к некоторым переменным *по* умолчанию ограничен. Чтобы взаимодействовать с этими переменными из других контрактов, необходимо задать ряд функций. Попробуем скомпилировать наш простой контракт и вставить его на вкладке *Functions*.

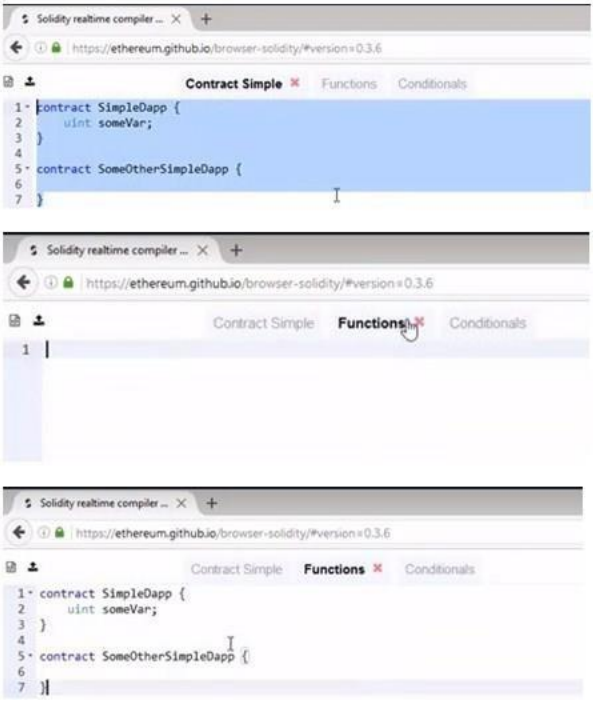


Рис.31.

Создадим две функции для работы с этой переменной. В среде Solidity одни функции могут возвращать некоторые значения, а другие - нет. Зададим переменную *someVar*. Она будет брать *значение* другой переменной *myVar*. Еще потребуется *функция*, возвращающая *значение someVar*.

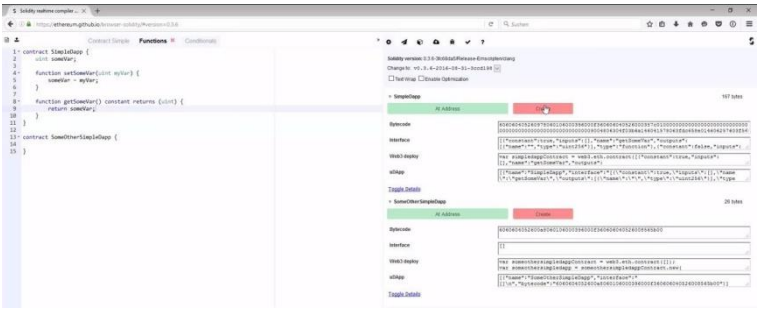


Рис.32.

Все эти функции относятся к постоянным функциям, которые могут только возвращать значения. Они не могут изменять данные в блокчейне и поэтому для их обсчета не требуется тратить газ.

Посмотрим внимательнее, как работает *приложение SimpleDapp*. Сейчас *функция getSomeVar* возвращает наше число. *По*умолчанию эта *переменная* равна нулю.

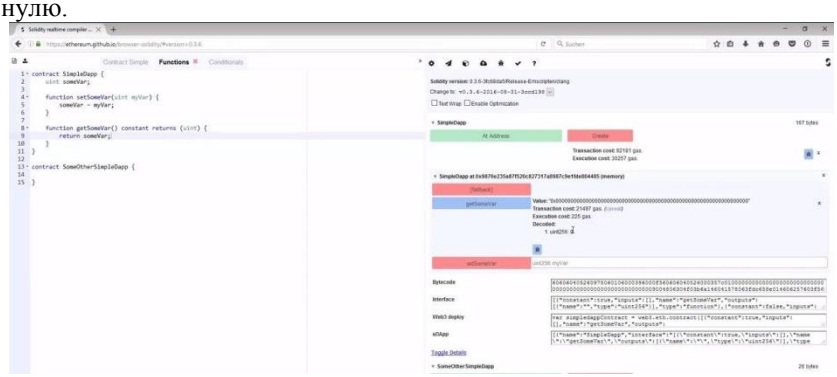


Рис.33.

В среде Solidity переменные инициализировать не нужно, некоторые значения им заданы *по* умолчанию. Теперь попробуем задать этой переменной другое *значение*, например, 4. *getSomeVar* теперь возвращает четыре.

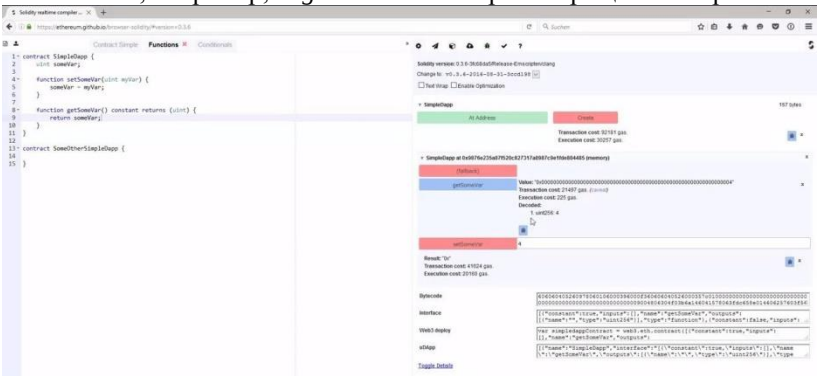


Рис.34.

Работа с функциями в Solidity не отличается от других языков программирования. Например, можно задать другую функцию, которая будет умножать нашу переменную на четыре и задавать какое-нибудь *значение* другой переменной. Теперь *функция getSomeVar* возвращает шестнадцать.

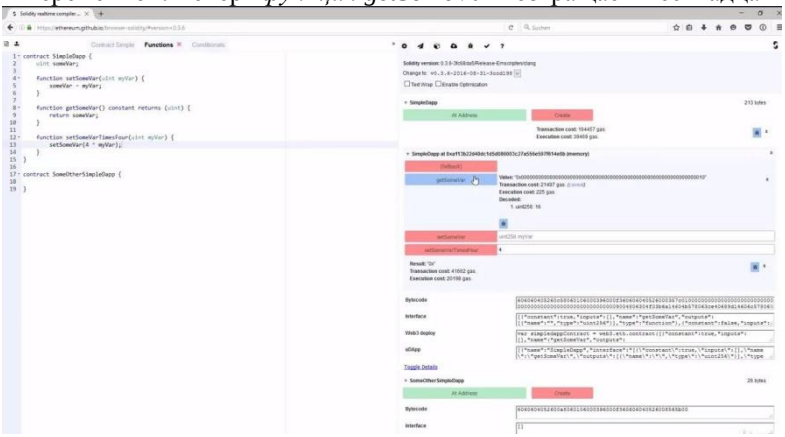


Рис.35.

В среде Solidity содержимое контракта может обращаться к другому контракту. Давайте создадим контракт

SomeOtherSimpleDapp, который будет возвращать *значение someVar* из контракта *SimpleDapp*.

Конструктор в среде Solidity работает примерно так же, как и любой другой *конструктор*. Он будет вызываться, когда контракт создается, а его *функция* называется так же, как и сам контракт. Вот простой *конструктор* для контракта *SomeOtherSimpleDapp*. Поработаем с контрактом *SimpleDapp*. Создадим переменную типа *SimpleDapp*, чтобы при ее инициализации во время создания контракта она сразу получила тип *SimpleDapp*. Это будет работать, если напрямую указать *адрес* контракта, являющегося источником типа. Можем создать еще одну копию *SimpleDapp*.

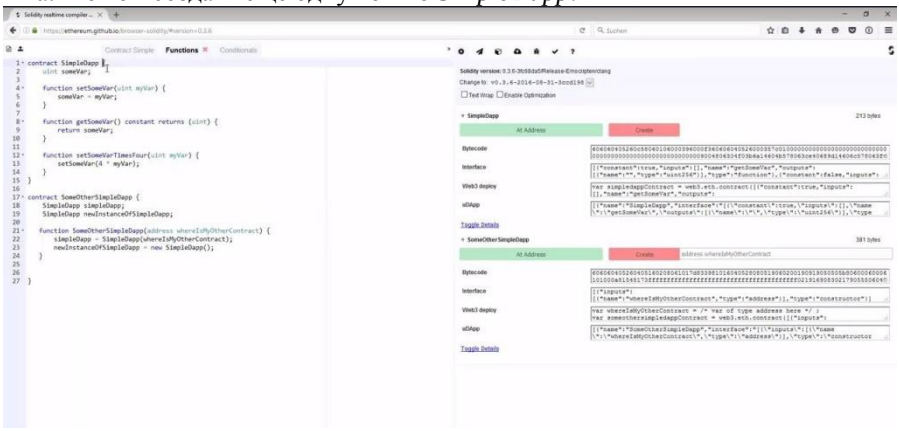
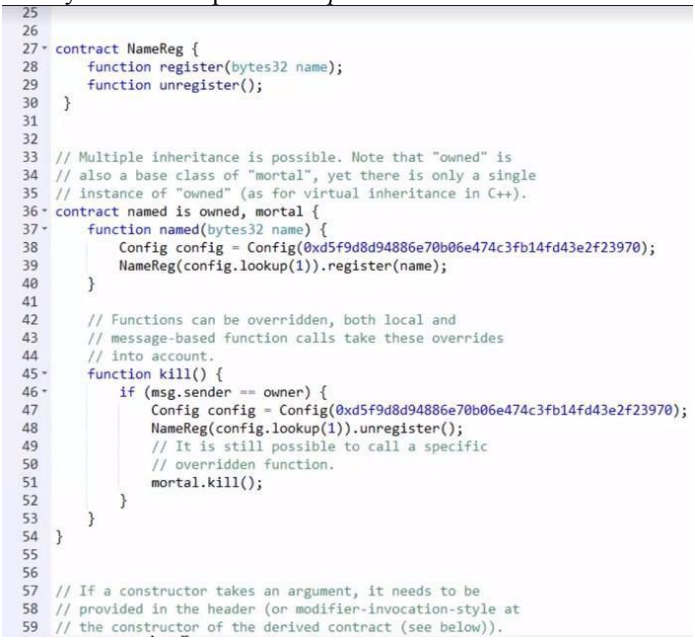


Рис.36.

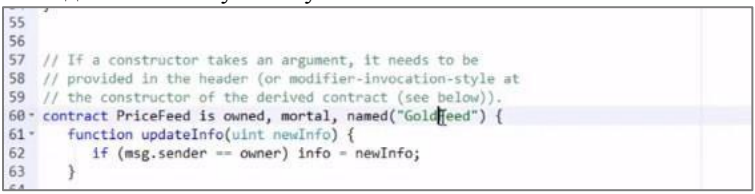
Посмотрим, как можно взаимодействовать с контрактом. В случае с первым контрактом *SimpleDapp* имеем дело со ссылкой на контракт, созданный выше. Как же это работает? Мы создали контракт *SimpleDapp*. У него есть некоторый *адрес*. Сейчас *функция getSomeVar* возвращает ноль, поскольку мы не указали конкретный *адрес*.



```
25
26
27  contract NameReg {
28      function register(bytes32 name);
29      function unregister();
30   }
31
32
33  // Multiple inheritance is possible. Note that "owned" is
34  // also a base class of "mortal", yet there is only a single
35  // instance of "owned" (as for virtual inheritance in C++).
36  contract named is owned, mortal {
37      function named(bytes32 name) {
38          Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
39          NameReg(config.lookup(1)).register(name);
40      }
41
42      // Functions can be overridden, both local and
43      // message-based function calls take these overrides
44      // into account.
45      function kill() {
46          if (msg.sender == owner) {
47              Config config = Config(0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970);
48              NameReg(config.lookup(1)).unregister();
49              // It is still possible to call a specific
50              // overridden function.
51              mortal.kill();
52          }
53      }
54  }
55
56
57  // If a constructor takes an argument, it needs to be
58  // provided in the header (or modifier-invocation-style at
59  // the constructor of the derived contract (see below)).
```

Рис.37.

Рассмотрим следующий контракт *PriceFeed*. Он является дочерним сразу для трех контрактов. Один из них - это *owned*, основной контракт. Второй - это *mortal*, а третий - *named*, напрямую задающий имя. Если *конструктор* требует *аргумент*, например, контракт *named*: имя контракта, имя конструктора - это требование должно быть упомянуто в заголовке.



```
55
56
57  // If a constructor takes an argument, it needs to be
58  // provided in the header (or modifier-invocation-style at
59  // the constructor of the derived contract (see below)).
60  contract PriceFeed is owned, mortal, named("GoldFeed") {
61      function updateInfo(uint newInfo) {
62          if (msg.sender == owner) info = newInfo;
63      }
```

Рис.38.

В контракте *PriceFeed* есть и другая логика. Посмотрим, что произойдет, если другой контракт вызовет *PriceFeed* и его функцию *kill*. Вернемся к контракту owned с конструктором. Есть контракт *mortal*, являющийся дочерним для *owned*, а также *функция kill*. Кроме того, есть контракты *Base1* и *Base2*. Оба они являются дочерними для *mortal*, оба наследуют функцию *kill* и вызывают в конце унаследованную процедуру mortal.kill. Контракт *Final* является дочерним для *Base1* и *Base2*.

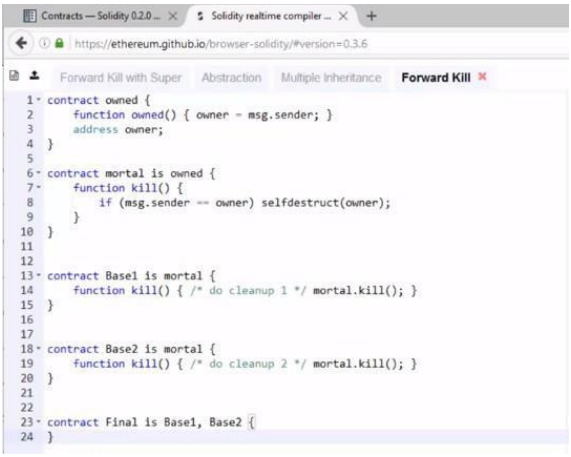


Рис.39.

Что произойдет, если другой контракт вызовет функцию Final.kill Она вызовет функцию Base2.kill, а до контракта *Base1* цепочка вызовов так и не дойдет. Эту проблему можно обойти с помощью служебного слова *Super*. Мы имеем дело с двумя классами, *Base1* и *Base2*, и вместо вызова функции mortal.kill будем вызывать super.kill. Этот способ вызова гарантирует, что если какой-либо другой контракт будет вызывать final.kill, то вначале он вызовет функцию *kill* из контракта *Base1*, Base1.kill, а затем - следующий контракт в графе наследования, то есть *Base2*. Поэтому будет вызвана *функция* Base2.kill, то есть *вызов функции* super.kill обеспечивает *вызов функции* mortal.kill.

Стоит отметить, что *компилятор* не знает, *по* какому адресу хранится библиотека. Поэтому эти адреса должны быть указаны, чтобы связующая процедура могла к ним обратиться. Можно также указывать адреса вручную. Чтобы узнать больше о библиотеках, ознакомьтесь с документацией Solidity - в разделе *Contract and* Libraries.

Типы переменных, массивы, структуры и ассоциации.

Эта работа посвящена основным типам переменных, структурам, ассоциациям и способам их применения в среде Solidity. В среде Solidity есть булев тип, тип целых чисел и тип неотрицательных целых чисел, байты, строки, массивы, структуры, ассоциации и *пользовательский тип*. Величины с фиксированной запятой еще не внедрены. Посмотрим в деталях, как работают основные типы. Их описание можно найти в документации Solidity, в разделе Types.

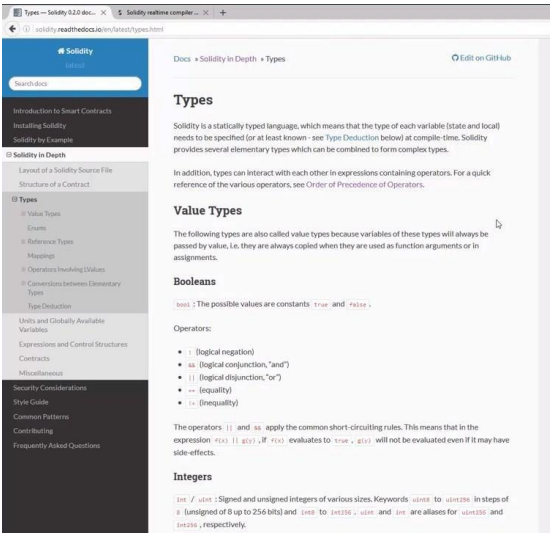


Рис.40.

Переменные булевого типа могут принимать два значения: правда или *ложь*, как и в других языках программирования. Численные величины бывают типа *int* - с поддержкой знака минус, или *uint* - только для неотрицательных значений. Для их определения есть *целый* ряд ключевых слов, начиная от uint8 и до uint256 с шагом в 8. Это верно и для переменных целочисленного типа.

Рассмотрим пример. В конструкторе, записанном в нашем контракте, назначим несколько величин. Переменной *myBool* будет присвоено *значение* "правда", переменной *myUint8* - максимальное *значение* для типа *uint8*, *значение* "X" - символьным переменным. Есть также типы массивов символов и строки. В данном случаем две функции, возвращающие значения для символьных и численных переменных. *Переменная myUint256* возвращает 256 *бит* неотрицательного целого числа, а *функция getMyBytes32* - 32 бита символов.

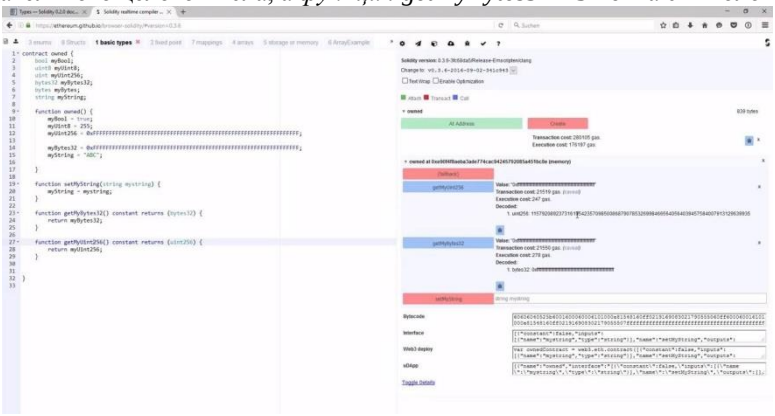


Рис.41.

Скопируем пример с раздела в *браузер* Solidity. В контракте этот тип назван *ActionChoices* и включает в себя значения переменных *GoLeft, GoRight, GoStraight* и *SitStill*. Для внутренней обработки Solidity переводит подобные массивы в тип *uint*, сначала применяя тип *uint8*, и расширяя его до *uint16* и так далее *по* мере роста числа вариантов значения переменных. Скомпилируем код и создадим контракт.

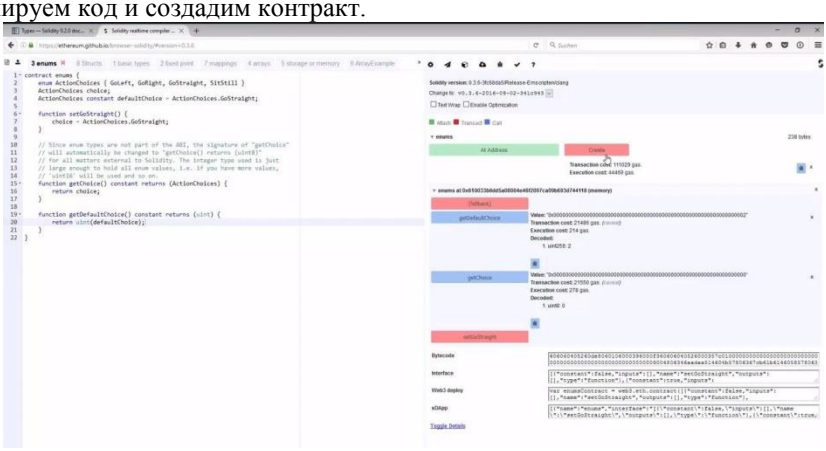


Рис.42.

В нем две функции - *getChoice* и *getDefaultChoice*. Обратим внимание на возвращаемую величину. *Функция getChoice* возвращает выбор действий; для целей внутренней обработки *список* действий, из которых можно выбирать, будет храниться в типе *uint8*, поскольку *список* состоит всего из четырех позиций. Одновременно с этим *функция getDefaultChoices* будет возвращать результат типа *uint256*, поэтому если потребуется работать с типом полученного результата, его нужно будет переопределить в тип *uint256*. Посмотрим на полученный результат в правой части браузера Solidity. *Функция getDefaultChoice* возвращает результат типа *uint256*, и этот результат - два. (Обратите внимание на порядок перечисления вариантов выбора в определении типа.) *Функция getChoice* возвращает результат типа *uint8*, в данном случае - ноль. Напомним, что все вводимые переменные и функции имеют некоторые значения *по* умолчанию, в данном случае *значение по* умолчанию - ноль. *Значение по* умолчанию для булевых переменных - это "*ложь*".

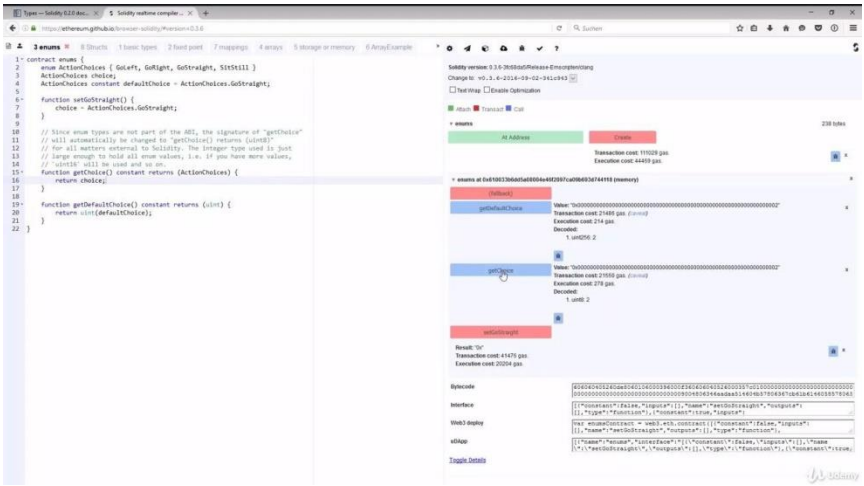


Рис.43.

Давайте сделаем выбор *goStraight* с помощью функции *setGoStraight* и проверим, каков будет результат работы функции

getChoice. Теперь он тоже равен двум.

Массивы в среде Solidity бывают фиксированные и динамические.

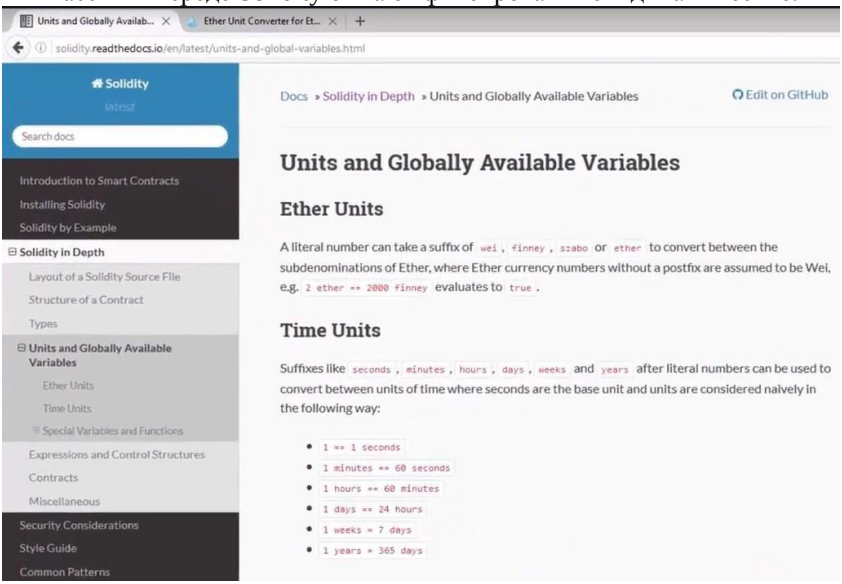


Рис.44.

Прежде чем начать программировать, стоит ознакомиться с используемыми в среде Ethereum единицами измерений. На веб - сайте http://ether.fund/tool/converter расположен очень удобный конвертер единиц.

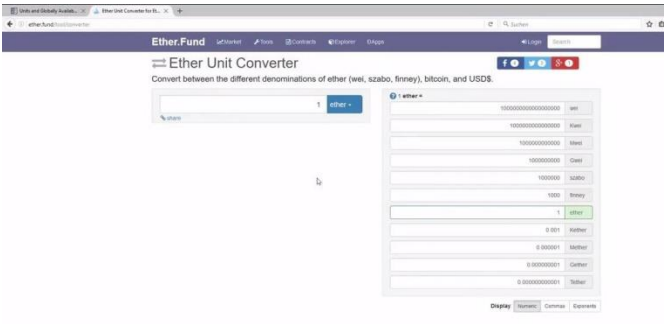


Рис.45.

Самой маленькой единицей измерения эфира является вей (wei). Один квинтиллион веев составляют эфир (ether). Финни (finney) меньше эфира, а сабо (szabo) меньше финни. Один эфир - это тысяча финни, а один финни - это тысяча сабо, и так далее. В коде Solidity можно использовать эти названия как служебные слова для определения единиц измерений, например, можно написать "2 ether == 2000 finney", и система вернет *значение* "*истина*".

Что касается единиц измерения времени, то минимальной величиной в Ethereum является секунда (seconds). Как и в случае с валютой, работают служебные слова, например, можно написать "1 minutes is 60 seconds".

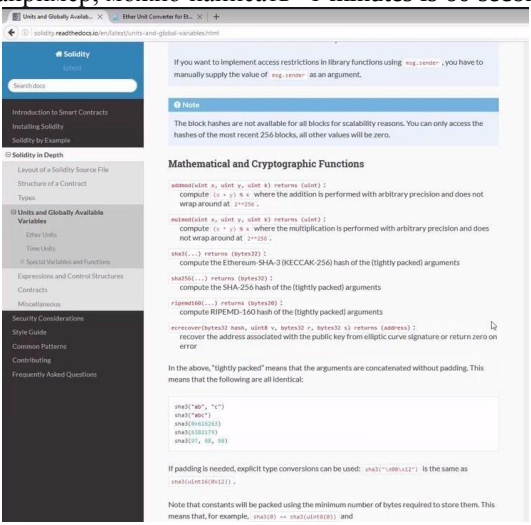


Рис.46.

У типа *address* есть некоторые особые члены. Один из них - это *balance*, возвращающий количество эфира на балансе у этого адреса, выраженное в веях. Другой - это *send*. С помощью функции *send* можно отправить этому адресу эфир. *Функция send* вернет *значение* "*истина*" в случае успешного перевода средств.

Наконец, у типа *contract* тоже есть два члена. Первый - это *this*, *указатель* на *адрес* текущего контракта. Второй - это *функция* self-destruct. Ее

вызов уничтожит контракт и отправит оставшийся эфир получателю, указанному в качестве параметра функции address *recipient*.

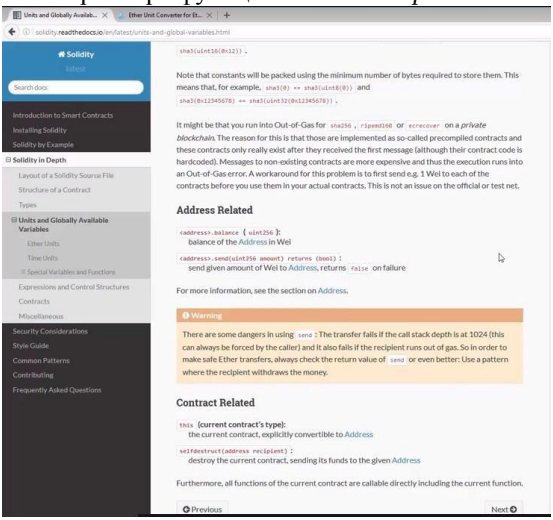


Рис.47.

Задание

1. Создайте следующие смарт-контракты на языке Solidity:

3. Контракт с тремя функциями, одна задает значение переменной (используя механизм передачи параметра), вторая - удваивает значение переменной, а третья - его возвращает.

4. В одном файле два контракта: первый - из предыдущего задания, второй - содержит функцию, возвращающую значение переменной из первого контракта.

5. Контракт, в котором используется условный оператор.

6. Контракт с массивами.

Контрольные вопросы

1. Структура контракта?
2. Единицы и глобальные переменные в языке Solidity
3. Создание контракта
4. Область видимости
5. Геттер-функции
6. Модификаторы функций
7. Константы
8. Функции
9. События
10. Наследование
11. Абстрактные контракты
12. Интерфейсы
13. Библиотеки?
14. Типы данных в языке Solidity?

Библиографический список

1. Могайар У. Блокчейн для бизнеса. – М: Издательство «Эксмо», 2018. – 224 с.
2. Фергюсон Н., Шнайер Б. Практическая криптография: Пер. с англ. – М.: Издательский дом “Вильямс”, 2004. — 432 с.
3. Головинский, П.А. Блокчейн и криптовалюты [Текст] : учебно-справочное пособие для преподавателей и студентов высших учебных заведений / ФГБОУ ВО "Воронеж. гос. техн. ун-т". - Воронеж : Цифровая полиграфия, 2019. - 88 с.
4. Свэн М. Блокчейн. Схема новой экономики; перевод, оформление, издание – М,: Издательство «Олимп – Бизнес», 2017. – 240 с.
5. Максуров, А.А. Блокчейн, криптовалюта, майнинг: понятие и правовое регулирование : монография / А. А. Максуров. - Блокчейн, криптовалюта, майнинг: понятие и правовое регулирование ; 2024-05-18. - Москва : Дашков и К, 2021. - 212 с.

БЛОКЧЕЙН - ТЕХНОЛОГИИ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к выполнению лабораторных работ
по дисциплине «Blockchain - технологии»
для студентов направления 27.03.05 «Инноватика»
(профиль «Инновационные технологии») всех форм обучения

Составители:
канд. техн. наук Д.В. Сысоев,
докт. физ.-мат. наук П.А. Головинский

Подписано к изданию _____.
Уч.-изд. л. _____.