Децентрализованные приложения и смарт-контракты (продолжение)

Структура контракта

```
// SPDX-License-Identifier: UNLICENSED
```
тип лицензии

```
pragma solidity (><=)0.7.0;
```
версия компилятора

```
contract NewContract is BaseContract {
```
объявление контракта

```
        uint public a;
        bool b;
        address Vova = 0xAc771378BB6c2b8878fbF75F80880cbdDefd1B1e;
```
глобальные переменные

```
        constructor() {
                .........
        }
```
конструктор – функция, которая выполняется при публикации контракта в блокчейне

```
        function MyFunction() {
                .........
        }
```
методы контракта – функции, вызываемые транзакциями пользователей или другими контрактами

```
}
```

Области видимости глобальных переменных контракта

`public`	поле данных, непосредственно доступное для чтения другим контрактам
`internal`	поле данных, доступ к которому возможен только из контракта или его потомков
`private`	поле данных, доступное только для методов контракта (по умолчанию)

Специальные функции смарт-контракта

`constructor`	**Конструктор смарт-контракта**, выполняется однократно при публикации контракта в блокчейне. Перед выполнением конструктора поля данных инициализируются указанными при их определении значениями либо значениями по умолчанию. **Описание:** **`constructor`**`([<список аргументов>]) { ... }`
`fallback`	**Резервная функция,** выполняется если транзакция вызывает отсутствующую в контракте функцию или вовсе не содержит вызова функции. **Описание:** **`fallback(bytes calldata`** `input) ` **`external`** `[`**`payable`**`]` **`returns(bytes memory`** `output) { ... }`
`receive`	**Receive-функция**, вызывается при переводе средств на адрес контракта без вызова какой-либо функции. Объём газа, доступный данной функции, ограничен 2300 единицами. **Описание:** **`receive() external payable`** `{ ... }`

Модификаторы

Модификатор – функция, выполняемая до или после некоторой другой функции и изменяющая её поведение.

Описание:

```
modifier <Имя>([<параметры>]) {
    <Код_до>
    _;
    <Код_после>
}
```

Имя модификатора указывается в списке атрибутов модифицируемой функции:

```
function doSomething() public <модификатор> {...}
```

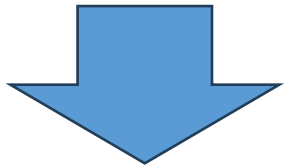
Бинарный интерфейс приложения (ABI)

Бинарный интерфейс приложения (Application Binary Interface, ABI) - последовательность байтов, содержащая в закодированном виде информацию о полях данных контракта с видимостью `public`, о функциях с видимостью `public` и `external`, а также о генерируемых контрактом событиях.

```
function bar(bytes3[2] memory info, uint32 val)
                                public pure returns (uint32) {...}
```

```
{
  "inputs": [                                   Список аргументов функции
   {                                            Первый аргумент
    "internalType": "bytes3[2]",                Тип данных Solidity
    "name": "info",                             Имя аргумента
    "type": "bytes3[2]"                         Тип данных ABI
   },
   {                                            Второй аргумент
    "internalType": "uint32",                   Тип данных Solidity
    "name": "val",                              Имя аргумента
    "type": "uint32"                            Тип данных ABI
   }
  ],
  "name": "bar",                                Имя функции
  "outputs": [                                  Список возвращаемых значений
   {
    "internalType": "uint32",                   Тип данных Solidity
    "name": "",                                 Имя возвращаемого значения
    "type": "uint32"                            Тип данных ABI
   }
  ],
  "stateMutability": "pure",                    Изменение состояния EVM
  "type": "function"                            Тип записи в ABI
}
```

```
uint public value
```



```
function value() public view returns (uint256)
```

```
{
  "inputs": [],
  "name": "value",
  "outputs": [
   {
    "internalType": "uint256",
    "name": "",
    "type": "uint256"
   }
  ],
  "stateMutability": "view",
  "type": "function"
}
```

Функция не принимает аргументов

Имя функции

Список возвращаемых значений

Тип данных Solidity

Имя возвращаемого значения

Тип данных ABI

Изменение состояния EVM

Тип записи в ABI

События (Events)

- **События** - объекты, позволяющие смарт-контрактам записывать информацию в специальную структуру данных, сохраняемую в блокчейне - журнал транзакций EVM.
- Журнал транзакций может считываться внешними приложениями для получения информации о действиях, выполненных контрактом.
- Объявление события:

```
event Имя_события([список сохраняемых данных]);
```

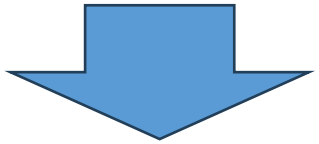
- До трёх полей данных могут быть объявлены с атрибутом **indexed**. Такие поля данных записываются в специальную область, называемую **темой события**. По значению темы внешние приложения могут быстро проводить фильтрацию событий. Остальные (неиндексированные) поля данных записываются в область данных журнала в кодировке ABI.
- Генерация события:

```
emit Имя_события([список сохраняемых данных]);
```

```
event TransferFrom(address indexed from, uint value);

...

emit TransferFrom(msg.sender, msg.value);
```



`logIndex: "0x1"`	Номер записи в журнале
`blockNumber: "0xd"`	Номер блока
`blockHash: "0x740d373c..."`	Хеш блока
`transactionHash: "0x621e7be7..."`	Хеш транзакции, осуществившей запись
`transactionIndex: "0x0"`	Номер транзакции в блоке
`address: "0x0fC5025C... "`	Адрес контракта, разместившего событие
`data: "0x00000000... "`	Данные события (поле value)
`topics: ["0x58becdf9...",` `"0x00000000..."]`	Темы события (хеш имени события и поле from)

```
event TransferFrom(address indexed from, uint value);
```

```
{
  "anonymous": false,                          Анонимное событие или нет
  "inputs": [                                  Перечень записываемых данных
   {                                           Первое поле данных
       "indexed": true,                        Индексируемое поле или нет
       "internalType": "address",              Тип данных Solidity
       "name": "from",                         Имя поля
       "type": "address"                       Тип данных ABI
   },
   {                                           Второе поле данных
       "indexed": false,                       Индексируемое поле или нет
       "internalType": "uint256",              Тип данных Solidity
       "name": "value",                        Имя поля
       "type": "uint256"                       Тип данных ABI
   }

  ],
  "name": "TransferFrom",                      Имя события
  "type": "event"                              Тип записи в ABI
}
```

Обработка ошибок в контрактах

При возникновении ошибки в процессе выполнения функции смарт-контракта:

- работа функции прерывается;
- произведённые изменения не сохраняются в глобальном состоянии EVM;
- соответствующая транзакция признаётся некорректной и не записывается в блокчейн;
- часть газа, израсходованная на выполнение функции до возникновения ошибки, не возмещается отправившему транзакцию пользователю.

Системная ошибка (Panic)

Код	Описание ошибки
0x00	Ошибка, инициированная компилятором
0x01	Вызов функции assert с параметром, равным False
0x11	Арифметическое переполнение / потеря значения
0x12	Деление на ноль
0x21	Преобразование отрицательного или слишком большого числа к типу enum
0x22	Доступ к массиву байтов в хранилище в неправильной кодировке
0x31	Вызов метода pop() для пустого массива
0x32	Выход за границу массива
0x41	Выделение слишком большого блока памяти или создание слишком большого массива
0x51	Вызов через переменную функционального типа, имеющую нулевое значение

Функция assert

Программная генерация системной ошибки:

`assert``(<выражение>)`

генерирует ошибку Panic с кодом 0x01, если значение выражения равно **`False`**.

- Корректно работающий контракт не должен генерировать ошибку Panic даже в случае недопустимых значений параметров.

Программная ошибка (Error)

Программная ошибка – ошибка, генерируемая программой при недопустимых значениях параметров транзакции.

Генерация ошибки Error:

```
require(<выражение>, [<сообщение>])
```

генерирует ошибку Error, если значение выражения равно **`False`**.
В качестве причины выводит сообщение, если оно присутствует.

Пример:

```
require(msg.sender == Owner, "Function only for Owner");
```

Генерирует ошибку, если транзакция инициирована не пользователем с адресом **`Owner`**.

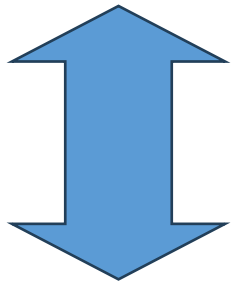
Безусловная генерация ошибки

revert([<сообщение>])

Прерывает выполнение текущей функции, отменяет выполненные ею действия и генерирует ошибку Error с заданным сообщением.

Пример:

```
require(msg.sender == Owner, "Function only for Owner");
```



```
if (msg.sender != Owner)
    revert("Function only for Owner");
```

Пользовательский тип ошибок

Объявление:

error <имя_типа>([<список данных>])

Команда объявляет пользовательский тип ошибок, возвращающий данные, указанные в списке.

Пример:

```
error InsufficientBalance(uint256 available, uint256 required);

...

if (amount > balance[msg.sender])
    revert(InsufficientBalance({available: balance[msg.sender],
           required: amount});
```

Обработка ошибок

Для обработки ошибок, возникших при вызовах внешних функций, используется конструкция **`try ... catch`**:

```
try <вызов> [returns (<возвр_знач>)] {
    <код_если_успешно>
} catch Error(string memory <данные>) {
    <код_если_error>
} catch Panic(uint <данные>) {
    <код_если_panic>
} catch (bytes memory <данные>) {
    <код_если_иное>
}
```