Solidity Cheatsheet

for version 0.4.25

TOPMONKS

bold orange = syntax. orange = example. *italics* = placeholder. Things in [] are optional. Things in {} are mandatory.

Pragma

pragma solidity ^{version}; pragma solidity ^0.4.25;

Compiler version to use.

Comments

```
// one liner
/* multi liner */
/// Natspec one liner
/** Natspec multi liner **/
```

Natspec is Doxygen-like syntax for documenting functions, arguments etc.

Simple data types

***type* [public] [constant] *name*;**
uint public amount;
delete amount; - deletes variable content

Getters are automatically generated for public vars. constant vars have to be initialized on declaration.

Basic types

bool, int(int256 alias), uint(uint256 alias), byte(bytes1 alias)

Ints and uints also as int8 - int256 in steps of 8. Bytes also as bytes1 - bytes32 in steps of 1.

String

string [storage|memory] [public] *name*;
string memo;

Bytes

bytes [storage|memory] [public] *name*;
bytes public myBytes;

myBytes.length; - get bytes length

myBytes.push(hex"ff"); - append byte

Address type

address [public] *name*;
address addr;

addr.balance
addr.transfer(100) - sends 100 wei, forwards 2300 gas, throws on error
addr.send(100) - sends 100 wei, forwards 2300 gas, returns true/false
addr.call('*func_signature*', [*argument*, ...]) - calls *func_signature* function and passes rest of the arguments to it
addr.delegatecall('*func_signature*', [*argument*, ...]) - same as call but evaluation context is set to current context

Structs

struct *name* { [*member*; ...] };

```
struct Person {
  string name;
  int age;
}
```

Person customer;
customer = Person({ name: 'Mr. Nobody', age: 1 });
customer2 = Person('Mr. Nobody', 1);
customer.name;
customer.age = 99;

Arrays

***type*[*size*] [memory|storage] [public] name;**
int[5] myArray;
int[] anotherArray; - dynamic array

delete myArray; - clears array's content
myArray.length; - get array's length
myArray.push(1); - push new element to array
myArray[3]; - get element
myArray[3] = 8; - set element
uint[] memory myArray = new uint[](5); - memory array with size determined at runtime. Memory arrays can't have dynamic size

Mappings

mapping(*key_type* => *value_type*) [public] name;
mapping (address => int) funds;

funds[address] = 33; - set value
funds[address]; - retrieve value

Simple types can be used as keys. Any types can be used as values.

All possible mapping keys always exists and have a default byte value of all zeroes.

Can't be iterated, checked for length, retrieve which keys were set etc. Also can't be created in memory.

Flow control

if (*condition*) { ... };
if (*condition*) { ... } else { ... };
for (*declaration*; *condition*; *expression*) { ... };
for (uint counter; counter < 10; counter++) { ... };
while (*condition*) { ... };
do { ... } while (*condition*);
***condition* ? *expression* : *expression*;**

continue, break and return can be used to influence flow.

Enums

enum *name* { [*state*, ...] };
enum Mood { Happy, Anxious, Sad };

Mood dogMood;
dogMood = Mood.Happy;

Imports

import "*path*";
import * as *name* from "*path*";
import { *name* as *alias*|*name*, ... } from "*path*";
import "*path*" as *namespace*;

All paths are relative.

Contract declaration

```
contract Memo {
  string public memo;
  address owner;

  constructor(string _memo) public {
    memo = _memo;
    address = msg.sender;
  }

  function changeMemo(string newMemo) public onlyOwner {
    memo = newMemo;
  }

  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }
}
```

`Memo memo = new Memo("my precious memo");` - create new contract
`Memo memo = Memo(existingMemoAddress);` - intialize already existing contract
`this;` - current contract
`address(this);` - converts contract to address
`selfdestruct(ownerAddress);` - destroys contract and sends it's funds to address. Must be called from contract.

Functions

`function name([argument, ...]) [visibility] [view|pure] [payable] [modifier, ...] [returns([argument, ...])];`

```
function setName(string name) public { ... };
function getName() view public returns(string name) { ... };
function compute() private returns(int num1, int num2) { ... };
```

`compute();` - function call
`compute.value(300).gas(4000)();` - call a func, send some ether along, and set it's gas limit

Visibility can be **`public`**, **`private`**, **`internal`**, **`external`**

`view` func doesn't modify blockchain anyhow. **`pure`** func doesn't modify or read blockchain.

`payable` func can receive ether.

Inheritance

`contract name is [ancestor, ... ] { ... };`

`super.function();` - call func from first ancestor
`SpecificAncestor.function();` - call func from specific ancestor

Ancestor contructors

`contract Derived is Base(7) { ... };` - pass a static value to ancestor constructor
`constructor(uint _y) Base(_y * _y) public {};` - pass a dynamic value
All ancestor constructors are called upon contract initialization and all needs to have theirs arguments set in one of the ways above.

require / revert / assert

`require(condition,[message])` - use for public facing assertions
`revert([message])`
`assert(condition)` - use for internal assertions
All of these throw an error and revert current transaction.

Inline assembly

`assembly { ... }` - block with inline assembly to get closer to EVM

Deprecations

`constant` - function modifier
`year` - time literal suffix
`throw` - alias to revert
`block.blockhash(block_number);` - replaced by global **`blockhash(block_number);`**
`msg.gas` - replaced by global **`gasleft();`**
`suicide` - alias to **`selfdestruct(address)`**
`keccak256(a, b)` - deprecated in favor of`keccak256(abi.encodePacked(a, b))`
`callcode`

Also contructors written as functions with same name as contract.

Literals and globals

wei, **finney**, **szabo**, **ether** - literal number suffixes for transfering numbers to wei

```
1000 finney == 1 ether
```

seconds, **minutes**, **hours**, **days**, **weeks** - literal number suffixes for transfering numbers to miliseconds

```
60 seconds == 1 minutes
```

block.{method} - current block info

```
block.number
block.gaslimit
```

... and more

msg.{method} - various calldata `msg.gas`

```
msg.sender
```

... and more

tx.{method} - current transaction `tx.gasprice`

```
tx.origin
```

abi.{method} - encoding methods `abi.encode(arguments)`

```
abi.encodePacked(arguments)
```

... and more

Cryptographic funcs

```
keccak256([argument, ...]);
sha256([argument, ...]);
ripemd160([argument, ...]);
```

Others

this; - current contract

now; - current time

gasleft(); - remaining gas

Function Modifiers

```
modifier name([argument, ...]) { ... _; ... }
```

```
modifier onlyOwner {
  require(msg.sender == owner);
  _;
}
```

_; - marks where modified function call will run.

Usage

```
function a(int num) public onlyOwner { ... }
function b(int num) public onlyOwner, secondModifier { ... }
```

Passing arguments

```
function c(int num) public anotherModifier(num) { ... }
```

Events

```
event name([argument, ...]) [anonymous];
emit name([argument, ...]);
```

```
event FundTransfer(address indexed to, uint value);
emit FundTransfer(someAddress, 100);
```

indexed arguments can be used for filtering later. **anonymous** event can't be filtered for by it's name.

Interfaces

```
interface AbstractZombie {
  function feedOn(address target) public onlyOwner;
  function getLevel() public view returns(uint level) onlyOwner;
}
```

Interfaces are like abstract contracts which other contracts can inherit. Inheriting contract has to implement all interface methods.

Interfaces can't contain anything else than function signatures.

Libraries

```
library MyLibrary {
  function add(int1 num, int num2) view returns(int result) {
    return num1 + num2;
  }
}

contract Number {
  int num;

  using MyLibrary for int;

  function addTwo() public {
    num = num.add(42);
  }
}
```

In the code above functions from MyLibrary can be called directly on all `int`s. The variable itself is then passed as a first argument to the library func.

Library functions can also be called statically as `MyLibrary.add(40, 2);`. It's not needed to use **`using`** clause then.

Code in library **is executed in caller's context** and libraries thus see and can manipulate local state.

Found a bug? Open up an issue or pull request here.