

# solidity-cheatsheet

Cheat sheet and best practices for solidity. Write smart contracts for Ethereum.

[View on GitHub](#)

## Solidity Cheatsheet and Best practices

### Motivation

This document is a cheatsheet for **Solidity** that you can use to write **Smart Contracts** for **Ethereum** based blockchain.

This guide is not intended to teach you Solidity from the ground up, but to help developers with basic knowledge who may struggle to get familiar with Smart Contracts and Blockchain because of the Solidity concepts used.

**Note:** If you have basic knowledge in JavaScript, it's easier to learn Solidity.

### Table of contents

- [Solidity Cheatsheet and Best practices](#)
  - [Motivation](#)
  - [Table of contents](#)
  - [Version pragma](#)
  - [Import files](#)
  - [Types](#)
    - [Boolean](#)
    - [Integer](#)
    - [Address](#)
      - [balance](#)
      - [transfer and send](#)
      - [call](#)
      - [delegatecall](#)
      - [callcode](#)
    - [Array](#)
    - [Fixed byte arrays](#)
    - [Dynamic byte arrays](#)
    - [Enum](#)

- Struct
- Mapping
- Control Structures
- Functions
  - Structure
  - Access modifiers
  - Parameters
    - Input parameters
    - Output parameters
  - Constructor
  - Function Calls
    - Internal Function Calls
    - External Function Calls
    - Named Calls
    - Unnamed function parameters
  - Function type
  - Function Modifier
  - View or Constant Functions
  - Pure Functions
  - Payable Functions
  - Fallback Function
- Contracts
  - Creating contracts using `new`
  - Contract Inheritance
    - Multiple inheritance
    - Constructor of base class
  - Abstract Contracts
- Interface
- Events
- Library
- Using - For
- Error Handling
- Global variables
  - Block variables
  - Transaction variables
  - Mathematical and Cryptographic Functions
  - Contract Related

## Version pragma

`pragma solidity ^0.5.2;` will compile with a compiler version  $\geq 0.5.2$  and  $< 0.6.0$ .

## Import files

```
import "filename";
```

```
import * as symbolName from "filename"; or import "filename" as symbolName;
```

```
import {symbol1 as alias, symbol2} from "filename";
```

## Types

### Boolean

`bool` : true or false

Operators:

- Logical : `!` (logical negation), `&&` (AND), `||` (OR)
- Comparisons : `==` (equality), `!=` (inequality)

### Integer

Unsigned : `uint8` | `uint16` | `uint32` | `uint64` | `uint128` | `uint256(uint)`

Signed : `int8` | `int16` | `int32` | `int64` | `int128` | `int256(int)`

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=` and `>`
- Bit operators: `&`, `|`, `^` (bitwise exclusive or) and `~` (bitwise negation)
- Arithmetic operators: `+`, `-`, unary `-`, unary `+`, `*`, `/`, `%`, `**` (exponentiation), `<<` (left shift) and `>>` (right shift)

### Address

`address` : Holds an Ethereum address (20 byte value). `address payable` : Same as address, but includes additional methods `transfer` and `send`

Operators:

- Comparisons: `<=`, `<`, `==`, `!=`, `>=` and `>`

Methods:

### balance

- `<address>.balance (uint256) : balance of the Address in Wei`

## transfer and send

- `<address>.transfer(uint256 amount) : send given amount of Wei to Address, throws on failure`
- `<address>.send(uint256 amount) returns (bool) : send given amount of Wei to Address, returns false on failure`

## call

- `<address>.call(...) returns (bool) : issue low-level CALL, returns false on failure`

## delegatecall

- `<address>.delegatecall(...) returns (bool) : issue low-level DELEGATECALL, returns false on failure`

Delegatecall uses the code of the target address, taking all other aspects (storage, balance, ...) from the calling contract. The purpose of delegatecall is to use library code which is stored in another contract. The user has to ensure that the layout of storage in both contracts is suitable for delegatecall to be used.

```
contract A {
    uint value;
    address public sender;
    address a = address(0); // address of contract B
    function makeDelegateCall(uint _value) public {
        a.delegatecall(
            abi.encodePacked(bytes4(keccak256("setValue(uint)")), _value)
        ); // Value of A is modified
    }
}

contract B {
    uint value;
    address public sender;
    function setValue(uint _value) public {
        value = _value;
        sender = msg.sender; // msg.sender is preserved in delegatecall. It was not av
    }
}
```

gas() option is available for call, callcode and delegatecall. value() option is not supported for delegatecall.

## calldata

- `<address>.calldata(...)` returns `(bool)` : issue low-level CALLCODE, returns false on failure

Prior to homestead, only a limited variant called `calldata` was available that did not provide access to the original `msg.sender` and `msg.value` values.

## Array

Arrays can be dynamic or have a fixed size.

```
uint[] dynamicSizeArray;

uint[7] fixedSizeArray;
```

## Fixed byte arrays

`bytes1(byte)`, `bytes2`, `bytes3`, ..., `bytes32`.

Operators:

Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to bool) Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation), `<<` (left shift), `>>` (right shift) Index access: If `x` is of type `bytesl`, then `x[k]` for `0 <= k < l` returns the `k` th byte (read-only).

Members

- `.length` : read-only

## Dynamic byte arrays

`bytes` : Dynamically-sized `byte` array. It is similar to `byte[]`, but it is packed tightly in calldata. Not a value-type!

`string` : Dynamically-sized UTF-8-encoded string. It is equal to `bytes` but does not allow length or index access. Not a value-type!

## Enum

Enum works just like in every other language.

```
enum ActionChoices {
    GoLeft,
```

```
    GoRight,  
    GoStraight,  
    SitStill  
}  
  
ActionChoices choice = ActionChoices.GoStraight;
```

## Struct

New types can be declared using struct.

```
struct Funder {  
    address addr;  
    uint amount;  
}  
  
Funder funders;
```

## Mapping

Declared as `mapping(_KeyType => _ValueType)`

Mappings can be seen as **hash tables** which are virtually initialized such that every possible key exists and is mapped to a value.

**key** can be almost any type except for a mapping, a dynamically sized array, a contract, an enum, or a struct. **value** can actually be any type, including mappings.

## Control Structures

Most of the control structures from JavaScript are available in Solidity except for `switch` and `goto`.

- `if else`
- `while`
- `do`
- `for`
- `break`
- `continue`
- `return`
- `? :`

# Functions

## Structure

```
function (<parameter types>) {internal|external|public|private}  
[pure|constant|view|payable] [returns (<return types>)]
```

## Access modifiers

- `public` - Accessible from this contract, inherited contracts and externally
- `private` - Accessible only from this contract
- `internal` - Accessible only from this contract and contracts inheriting from it
- `external` - Cannot be accessed internally, only externally. Recommended to reduce gas. Access internally with `this.f`.

## Parameters

### Input parameters

Parameters are declared just like variables and are `memory` variables.

```
function f(uint _a, uint _b) {}
```

### Output parameters

Output parameters are declared after the `returns` keyword

```
function f(uint _a, uint _b) returns (uint _sum) {  
    _sum = _a + _b;  
}
```

Output can also be specified using `return` statement. In that case, we can omit parameter name `returns (uint)`.

Multiple return types are possible with `return (v0, v1, ..., vn)`.

## Constructor

Function that is executed during contract deployment. Defined using the `constructor` keyword.

```
contract C {  
    address owner;  
    uint status;  
    constructor(uint _status) {  
        owner = msg.sender;  
        status = _status;  
    }  
}
```

## Function Calls

### Internal Function Calls

Functions of the current contract can be called directly (internally - via jumps) and also recursively

```
contract C {  
    function funA() returns (uint) {  
        return 5;  
    }  
  
    function FunB(uint _a) returns (uint ret) {  
        return funA() + _a;  
    }  
}
```

### External Function Calls

`this.g(8);` and `c.g(2);` (where `c` is a contract instance) are also valid function calls, but, the function will be called “externally”, via a message call.

`.gas()` and `.value()` can also be used with external function calls.

### Named Calls

Function call arguments can also be given by name in any order as below.

```
function f(uint a, uint b) { }  
  
function g() {  
    f({b: 1, a: 2});  
}
```



## Unnamed function parameters

Parameters will be present on the stack, but are not accessible.

```
function f(uint a, uint) returns (uint) {  
    return a;  
}
```

## Function type

Pass function as a parameter to another function. Similar to `callbacks` and `delegates`

```
pragma solidity ^0.4.18;  
  
contract Oracle {  
    struct Request {  
        bytes data;  
        function(bytes memory) external callback;  
    }  
    Request[] requests;  
    event NewRequest(uint);  
    function query(bytes data, function(bytes memory) external callback) {  
        requests.push(Request(data, callback));  
        NewRequest(requests.length - 1);  
    }  
    function reply(uint requestID, bytes response) {  
        // Here goes the check that the reply comes from a trusted source  
        requests[requestID].callback(response);  
    }  
}  
  
contract OracleUser {  
    Oracle constant oracle = Oracle(0x1234567); // known contract  
    function buySomething() {  
        oracle.query("USD", this.oracleResponse);  
    }  
    function oracleResponse(bytes response) {  
        require(msg.sender == address(oracle));  
    }  
}
```

## Function Modifier

Modifiers can automatically check a condition prior to executing the function.

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

## View or Constant Functions

Functions can be declared `view` or `constant` in which case they promise not to modify the state, but can read from them.

```
function f(uint a) view returns (uint) {  
    return a * b; // where b is a storage variable  
}
```

The compiler does not enforce yet that a `view` method is not modifying state.

## Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state.

```
function f(uint a) pure returns (uint) {  
    return a * 42;  
}
```

## Payable Functions

Functions that receive `Ether` are marked as `payable` function.

## Fallback Function

A contract can have exactly one **unnamed function**. This function cannot have arguments and cannot return anything. It is executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all).

```
function() {  
    // Do something  
}
```

```
}
```

## Contracts

### Creating contracts using `new`

Contracts can be created from another contract using `new` keyword. The source of the contract has to be known in advance.

```
contract A {
    function add(uint _a, uint _b) returns (uint) {
        return _a + _b;
    }
}

contract C {
    address a;
    function f(uint _a) {
        a = new A();
    }
}
```

### Contract Inheritance

Solidity supports multiple inheritance and polymorphism.

```
contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract final is mortal {
    function kill() {
        super.kill(); // Calls kill() of mortal.
    }
}
```

## Multiple inheritance

```
contract A {}  
contract B {}  
contract C is A, B {}
```

## Constructor of base class

```
contract A {  
    uint a;  
    constructor(uint _a) { a = _a; }  
}  
  
contract B is A(1) {  
    constructor(uint _b) A(_b) {  
    }  
}
```

## Abstract Contracts

Contracts that contain implemented and non-implemented functions. Such contracts cannot be compiled, but they can be used as base contracts.

```
pragma solidity ^0.4.0;  
  
contract A {  
    function C() returns (bytes32);  
}  
  
contract B is A {  
    function C() returns (bytes32) { return "c"; }  
}
```

## Interface

Interfaces are similar to abstract contracts, but they have restrictions:

- Cannot have any functions implemented.
- Cannot inherit other contracts or interfaces.
- Cannot define constructor.
- Cannot define variables.
- Cannot define structs.
- Cannot define enums.

```
pragma solidity ^0.4.11;

interface Token {
    function transfer(address recipient, uint amount);
}
```

## Events

Events allow the convenient usage of the EVM logging facilities, which in turn can be used to “call” JavaScript callbacks in the user interface of a dapp, which listen for these events.

Up to three parameters can receive the attribute indexed, which will cause the respective arguments to be searched for.

All non-indexed arguments will be stored in the data part of the log.

```
pragma solidity ^0.4.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) payable {
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

## Library

Libraries are similar to contracts, but they are deployed only once at a specific address, and their code is used with `delegatecall (callcode)`.

```
library arithmetic {
    function add(uint _a, uint _b) returns (uint) {
        return _a + _b;
    }
}

contract C {
    uint sum;
```

```
function f() {  
    sum = arithmetic.add(2, 3);  
}  
}
```

## Using - For

using A for B; can be used to attach library functions to any type.

```
library arithmetic {  
    function add(uint _a, uint _b) returns (uint) {  
        return _a + _b;  
    }  
}  
  
contract C {  
    using arithmetic for uint;  
  
    uint sum;  
    function f(uint _a) {  
        sum = _a.add(3);  
    }  
}
```

## Error Handling

- `assert(bool condition)` : throws if the condition is not met - to be used for internal errors.
- `require(bool condition)` : throws if the condition is not met - to be used for errors in inputs or external components.
- `revert()` : abort execution and revert state changes

```
function sendHalf(address addr) payable returns (uint balance) {  
    require(msg.value % 2 == 0); // Only allow even numbers  
    uint balanceBeforeTransfer = this.balance;  
    addr.transfer(msg.value / 2);  
    assert(this.balance == balanceBeforeTransfer - msg.value / 2);  
    return this.balance;  
}
```

Catching exceptions is not yet possible.

# Global variables

## Block variables

- `block.blockhash(uint blockNumber)` returns `(bytes32)` : hash of the given block - only works for the 256 most recent blocks excluding current
- `block.coinbase (address)` : current block miner's address
- `block.difficulty (uint)` : current block difficulty
- `block.gaslimit (uint)` : current block gaslimit
- `block.number (uint)` : current block number
- `block.timestamp (uint)` : current block timestamp as seconds since unix epoch
- `now (uint)` : current block timestamp (alias for `block.timestamp`)

## Transaction variables

- `msg.data (bytes)` : complete calldata
- `msg.gas (uint)` : remaining gas
- `msg.sender (address)` : sender of the message (current call)
- `msg.sig (bytes4)` : first four bytes of the calldata (i.e. function identifier)
- `msg.value (uint)` : number of wei sent with the message
- `tx.gasprice (uint)` : gas price of the transaction
- `tx.origin (address)` : sender of the transaction (full call chain)

## Mathematical and Cryptographic Functions

- `addmod(uint x, uint y, uint k)` returns `(uint)` : compute  $(x + y) \% k$  where the addition is performed with arbitrary precision and does not wrap around at  $2^{256}$ .
- `mulmod(uint x, uint y, uint k)` returns `(uint)` : compute  $(x * y) \% k$  where the multiplication is performed with arbitrary precision and does not wrap around at  $2^{256}$ .
- `keccak256(...)` returns `(bytes32)` : compute the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments
- `sha256(...)` returns `(bytes32)` : compute the SHA-256 hash of the (tightly packed) arguments
- `sha3(...)` returns `(bytes32)` : alias to keccak256
- `ripemd160(...)` returns `(bytes20)` : compute RIPEMD-160 hash of the (tightly packed) arguments
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s)` returns `(address)` : recover the address associated with the public key from elliptic curve signature or return zero on error (example usage)

## Contract Related

- `this` (current contract's type) : the current contract, explicitly convertible to Address
  - `selfdestruct(address recipient)` : destroy the current contract, sending its funds to the given Address
  - `suicide(address recipient)` : alias to `selfdestruct`. Soon to be deprecated.
- 

**solidity-cheatsheet** is maintained by **manojpramesh**.

This page was generated by [GitHub Pages](#).