# Solidity Cheat Sheet

We created this Solidity Cheat Sheet initially for students of our Solidity, Blockchain and Ethereum Developer Bootcamp. But we're now sharing it with any and all Developers that want to learn and remember some of the key functions and concepts of Solidity, and have a quick reference guide to the basics of Solidity development.<

If you're just starting to learn solidity, you've made a great choice! As the core programming language used to implement smart contracts on Ethereum, it's a great language to learn if you're interested in becoming a Blockchain Developer.

However, if you're stuck in an endless cycle of YouTube tutorials and want to start building real world projects, become a professional developer, have fun and actually get hired, then come join the Zero To Mastery Academy.

You'll learn Solidity from an actual industry professional alongside thousands of students in our private Discord community.

You'll not only learn to become a top 10% Solidity Developer by learning advanced topics most courses don't cover. But you'll also build awesome projects, including your own ERC-20 token, a stablecoin, and a decentralized casino that you can add to your portfolio and wow employers with!

**Just want the cheatsheet?** No problem! Please enjoy and if you'd like to submit any suggestions, feel free to email us at support@zerotomastery.io

## Contents

## Structure of a Smart Contract

SPDX-License-Identifier: MIT

Specify that the source code is for a version of Solidity of exactly 0.8.17: `pragma solidity 0.8.17;`

Specify any imports: `import "./MyOtherContract.sol";`

A contract is a collection of functions and data (its state) that resides at a specific address on the blockchain.

```
contract HelloWorld {

// The keyword "public" makes variables accessible from ou

string public message;

// The keyword "private" makes variables only accessible f

address private owner;

event MessageUpdated(string indexed newMessage);
error NotOwner(address owner, address sender);

// any struct and enum types

modifier onlyOwner {
        if (msg.sender != owner) {
            revert NotOwner(owner, msg.sender);
        }
        _;
}

// A special function only run during the creation of the

constructor(string memory initMessage) {

        // Takes a string value and stores the value in th

        message = initMessage;

        // setting owner as contract creator

        owner = msg.sender;
}

// An externally accessible function that takes a string a

function update(string memory newMessage) external onlyOwn
        message = newMessage;
        emit MessageUpdated(newMessage);
    }
}
```

## Variable Types

State variables can be declared private or public. Public will generate a public view function for the type. In addition they can be declared constant or immutable. Immutable variables can only be assigned in the constructor. Constant variables can only be assigned upon declaration.

### Simple Data Types

| | |
|---|---|
| `bool` | true or false |
| `uint (uint256)` | unsigned integer with 256 bits (also available are uint8...256 in steps of 8) |
| `int (int256)` | signed integer with 256 bits (also available are int8...256 in steps of 8) |
| `bytes32` | 32 raw bytes (also available are bytes1...32 in steps of 1) |

### Address

address: 0xba57bF26549F2Be7F69EC91E6a9db6Ce1e375390

`myAddr.balance`

Payable address also has `myAddr.transfer` which transfers Ether but reverts if receiver uses up more than 2300 gas. It's generally better to use .call and handle reentrancy issues separately:

```
(bool success,) = myAddr.call{value: 1 ether}("");
require(success, "Transfer failed");
```

Low-level call sends a transaction with any data to an address: myAddr.call{value: 1 ether, gas: 15000}(abi.encodeWithSelector(bytes4(keccak256("update(string)")), "myNewString"))

Like call, but will revert if the called function modifies the state in any way: `myAddr.staticcall`

Like call, but keeps all the context (like state) from current contract. Useful for external libraries and upgradable contracts with a proxy: `myAddr.delegatecall`

## Mapping

A hash table where every possible key exists and initially maps to a type's default value, e.g. 0 or "".

```
mapping(KeyType => ValueType) public myMappingName;
mapping(address => uint256) public balances;
mapping(address => mapping(address => uint256)) private _a

Set value: balances[myAddr] = 42;
Read value: balances[myAddr];
```

## Struct

```
struct Deposit {
  address depositor;
  uint256 amount;
}

Deposit memory deposit;
Deposit public deposit;
deposit = Deposit({ depositor: msg.sender, amount: msg.val
deposit2 = Deposit(0xa193…, 200);
```

```
Read value: deposit.depositor;
Set value: deposit.amount = 23;
```

## Enums

```
enum Colors { Red, Blue, Green };
Color color = Colors.Red;
```

## Arrays

```
uint8[] public myStateArray;
uint8[] public myStateArray = [1, 2, 3];
uint8[3] public myStateArray  = [1, 2, 3];
uint8[] memory myMemoryArray = new uint8[](3);
uint8[3] memory myMemoryArray = [1, 2, 3];

myStateArray.length;
```

Only dynamic state arrays:

```
myStateArray.push(3);
myStateArray.pop();
```

**Special Array bytes**: bytes memory/public data. More space-efficient form of bytes1[ ].

**Special Array string**: string memory/public name. Like bytes but no length or index access.

## Control Structures

- if (boolean) { ... } else { ... }
- while (boolean) { ... }
- do { ... } while (boolean)
- for (uint256 i; i < 10; i++) { ... }

- break;
- continue;
- return
- boolean ? ... : ...;

## Functions

```
function functionName([arg1, arg2...]) [public|external|in
function setBalance(uint256 newBalance) external { ... }
function getBalance() view external returns(uint256 balanc
function _helperFunction() private returns(uint256 myNumbe
```

- Function call for function in current contract: `_helperFunction();`
- Function call for function in external contract: `myContract.setBalance{value: 123, gas: 456 }(newBalance);`
- View functions don't modify state. They can be called to read data without sending a transaction.
- Pure functions are special view functions that don't even read data.
- Payable functions can receive Ether.

### Function Modifiers

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

function changeOwner(address newOwner) external onlyOwner
  owner = newOwner;
}
```

### Fallback Functions

```
contract MyContract {
    // executed when called with empty data, must be exter
    receive() external payable {}

    // executed when no other function matches, must be ex
    fallback() external {}
}
```

## Contracts

```
contract MyContract {
    uint256 public balance;
    constructor(uint256 initialBalance) { balance = initia
    function setBalance(uint256 newBalance) external { bal
}
```

- MyContract myContract = new MyContract(100);
- MyContract myContract2 = MyContract(0xa41ab…);
- this: current contract
- address(this): current contract's address

### Inheritance

```
contract MyAncestorContract2 {
    function myFunction() external virtual { … }
}

contract MyAncestorContract1 is MyAncestorContract2 {
    function myFunction() external virtual override { … }
}

contract MyContract is MyAncestorContract1 {
    function myFunction() external override(MyAncestorCont
}
```

- Call first ancestor function: `super.myFunction()`
- Call specific ancestor function: `MyAncestorContract2.myFunction()`

## Abstract Contracts

Abstract contracts cannot be instantiated. You can only use them by inheriting from them and implementing any non implemented functions.

```
abstract contract MyAbstractContract {
 function myImplementedFunction() external { … }
 function myNonImplementedFunction() external virtual; //
}
```

## Interfaces

Interfaces are like abstract contracts, but can only have non-implemented functions. Useful for interacting with standardized foreign contracts like ERC20.

```
interface MyInterface {
 function myNonImplementedFunction() external; // always v
}
```

## Libraries

```
library Math {
    function min(uint256 a, uint256 b) internal pure retur
        if (a > b) { return b; }
        return a;
    }

  function max(uint256 a, uint256 b) internal pure returns
        if (a &lt; b) { return b; }
        return a;
    }
}

contract MyContract {
    function min(uint256 a, uint256) public pure returns (
        return Math.min(a,b);
    }
    function max(uint256 x) public pure returns (uint256)
        return Math.max(a,b);
    }
}

// Using LibraryName for type:

library Math {
    function ceilDiv(uint256 a, uint256 b) internal pure r
        return a / b + (a % b == 0 ? 0 : 1);
    }
}

contract MyContract {
    using Math for uint256;
    function ceilDiv(uint256 a, uint256) public pure retur
        return x.ceilDiv(y);
    }
}
```

## Events

Events allow for efficient look up in the blockchain for finding deposit() transactions. Up to three attributes can be declared as indexed which allows filtering for it.

```
contract MyContract {
    event Deposit(
        address indexed depositor,
        uint256 amount
    );

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

## Checked or Unchecked Arithmetic

```
contract CheckedUncheckedTests {
    function checkedAdd() pure public returns (uint256) {
        return type(uint256).max + 1; // reverts
    }

    function checkedSub() pure public returns (uint256) {
        return type(uint256).min - 1; // reverts
    }

    function uncheckedAdd() pure public returns (uint256)
        // doesn't revert, but overflows and returns 0
        unchecked { return type(uint256).max + 1; }
    }

    function uncheckedSub() pure public returns (uint256)
        // doesn't revert, but underflows and returns 2^25
        unchecked { return type(uint256).min - 1; }
    }
}
```

## Custom Types: Example with Fixed Point

```
type FixedPoint is uint256

library FixedPointMath {
    uint256 constant MULTIPLIER = 10**18;

    function add(FixedPoint a, FixedPoint b) internal pure
        return FixedPoint.wrap(FixedPoint.unwrap(a) + Fixe
    }

    function mul(FixedPoint a, uint256 b) internal pure re
        return FixedPoint.wrap(FixedPoint.unwrap(a) * b);
    }

   function mulFixedPoint(uint256 number, FixedPoint fixed
        return (number * FixedPoint.unwrap(fixedPoint)) /
    }

    function divFixedPoint(uint256 number, FixedPoint fixe
        return (number * MULTIPLIER) / Wad.unwrap(fixedPoi
    }

    function fromFraction(uint256 numerator, uint256 denom
      if (numerator == 0) {
        return FixedPoint.wrap(0);
      }

      return FixedPoint.wrap((numerator * MULTIPLIER) / de
        }
    }
```

## Error Handling

```
error InsufficientBalance(uint256 available, uint256 requi

function transfer(address to, uint256 amount) public {
    if (amount > balance[msg.sender]) {
        revert InsufficientBalance({
            available: balance[msg.sender],
            required: amount
        });
    }

    balance[msg.sender] -= amount;
    balance[to] += amount;
}
```

Alternatively revert with a string:

- revert(“insufficient balance”);
- require(amount <= balance, “insufficient balance”);
- assert(amount <= balance) // reverts with Panic(0x01)

Other built-in panic errors:

| | |
|---|---|
| 0x00 | Used for generic compiler inserted panics. |
| 0x01 | If you call assert with an argument that evaluates to false. |
| 0x11 | If an arithmetic operation results in underflow or overflow outside of an unchecked { ... } block. |
| 0x12 | If you divide or modulo by zero (e.g. 5 / 0 or 23 % 0). |
| 0x21 | If you convert a value that is too big or negative into an enum type. |
| 0x22 | If you access a storage byte array that is incorrectly encoded. |
| 0x31 | If you call .pop() on an empty array. |
| 0x32 | If you access an array, bytesN or an array slice at an out-of-bounds or negative index (i.e. x[i] where i >= x.length or i < 0). |
| 0x41 | If you allocate too much memory or create an array that is too large. |
| 0x51 | If you call a zero-initialized variable of internal function type. |

# Global Variables

## Block

| | |
|---|---|
| `block.basefee (uint256)` | Current block's base fee (EIP-3198 and EIP-1559) |
| `block.chainid (uint256)` | Current chain id |
| `block.coinbase (address payable)` | Current block miner's address |
| `block.difficulty (uint256)` | Outdated old block difficulty, but since the Ethereum mainnet upgrade called Paris as part of 'The Merge' in September 2022 it is now deprecated and represents `prevrandao` : a value from the randomness generation process called Randao (see EIP-4399 for details) |
| `block.gaslimit (uint256)` | Current block gaslimit |
| `block.number (uint256)` | Current block number |
| `block.timestamp (uint256)` | Current block timestamp in seconds since Unix epoch |
| `blockhash(uint256 blockNumber) returns (bytes32)` | Hash of the given block - only works for 256 most recent blocks |

## Transaction

| | |
|---|---|
| `gasleft() returns (uint256)` | Remaining gas |
| `msg.data (bytes)` | Complete calldata |
| `msg.sender (address)` | Sender of the message (current call) |
| `msg.sig (bytes4)` | First four bytes of the calldata (i.e. function identifier) |
| `msg.value (uint256)` | Number of wei sent with the message |
| `tx.gasprice (uint256)` | Gas price of the transaction |
| `tx.origin (address)` | Sender of the transaction (full call chain) |

## ABI

| | |
|---|---|
| `abi.decode(bytes memory encodedData, (...)) returns (...)` | ABI-decodes the provided data. The types are given in parentheses as second argument. Example: (uint256 a, uint256[2] memory b, bytes memory c) = abi.decode(data, (uint256, uint256[2], bytes)) |
| `abi.encode(...) returns (bytes memory)` | ABI-encodes the given arguments |
| `abi.encodePacked(...) returns (bytes memory)` | Performs packed encoding of the given arguments. Note that this encoding can be ambiguous! |
| `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` | ABI-encodes the given arguments starting from the second and prepends the given four-byte selector |
| `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)` | ABI-encodes a call to functionPointer with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals abi.encodeWithSelector(functionPointer.selector, (...)) |
| `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` | Equivalent to abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...) |

## Type

| | |
|---|---|
| `type(C).name (string)` | The name of the contract |
| `type(C).creationCode (bytes memory)` | Creation bytecode of the given contract. |
| `type(C).runtimeCode (bytes memory)` | Runtime bytecode of the given contract. |
| `type(I).interfaceId (bytes4)` | Value containing the EIP-165 interface identifier of the given interface. |
| `type(T).min (T)` | The minimum value representable by the integer type T. |
| `type(T).max (T)` | The maximum value representable by the integer type T. |

## Cryptography

| | |
|---|---|
| `keccak256(bytes memory) returns (bytes32)` | Compute the Keccak-256 hash of the input |
| `sha256(bytes memory) returns (bytes32)` | Compute the SHA-256 hash of the input |
| `ripemd160(bytes memory) returns (bytes20)` | Compute the RIPEMD-160 hash of the input |

| | |
|---|---|
| `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)` | Recover address associated with the public key from elliptic curve signature, return zero on error |
| `addmod(uint256 x, uint256 y, uint256 k) returns (uint256)` | Compute (x + y) % k where the addition is performed with arbitrary precision and does not wrap around at 2··256. Assert that k != 0. |
| `mulmod(uint256 x, uint256 y, uint256 k) returns (uint256)` | Compute (x * y) % k where the multiplication is performed with arbitrary precision and does not wrap around at 2··256. Assert that k != 0. |

## Misc

| | |
|---|---|
| `bytes.concat(...) returns (bytes memory)` | Concatenates variable number of arguments to one byte array |
| `string.concat(...) returns (string memory)` | Concatenates variable number of arguments to one string array |
| `this (current contract's type)` | The current contract, explicitly convertible to address or address payable |
| `super` | The contract one level higher in the inheritance hierarchy |
| `selfdestruct(address payable recipient)` | Destroy the current contract, sending its funds to the given address. Does not give gas refunds anymore since LONDON hardfork. |

Back To Top