Cheatsheet

Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

Precedence	Description	Operator
1	Postfix increment and decrement	`++` , `--`
	New expression	`new <typename>`
	Array subscripting	`<array>[<index>]`
	Member access	`<object>.<member>`
	Function-like call	`<func>(<args...>)`
	Parentheses	`(<statement>)`
2	Prefix increment and decrement	`++` , `--`
	Unary minus	`-`
	Unary operations	`delete`
	Logical NOT	`!`
	Bitwise NOT	`~`
3	Exponentiation	`**`
4	Multiplication, division and modulo	`*` , `/` , `%`
5	Addition and subtraction	`+` , `-`
6	Bitwise shift operators	`<<` , `>>`
7	Bitwise AND	`&`
8	Bitwise XOR	`^`
9	Bitwise OR	`|`
10	Inequality operators	`<` , `>` , `<=` , `>=`
11	Equality operators	`==` , `!=`
12	Logical AND	`&&`
13	Logical OR	`||`
14	Ternary operator	`<conditional> ? <if-true> : <if-false>`
	Assignment operators	`=` , `|=` , `^=` , `&=` , `<<=` , `>>=` , `+=` , `-=` , `*=` , `/=` , `%=`
15	Comma operator	`,`

Global Variables

- `abi.decode(bytes memory encodedData, (...)) returns (...)` : ABI-decodes the provided data. The types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[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)), ...)`
- `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
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): 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 in seconds since Unix epoch
- `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` (`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)
- `assert(bool condition)`: abort execution and revert state changes if condition is `false` (use for internal error)
- `require(bool condition)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
- `require(bool condition, string memory message)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
- `revert()`: abort execution and revert state changes
- `revert(string memory message)`: abort execution and revert state changes providing an explanatory string
- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `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(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`. Assert that `k != 0` starting from version 0.5.0.
- `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`. Assert that `k != 0` starting from version 0.5.0.
- `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
- `<address>.balance` (`uint256`): balance of the Address in Wei
- `<address>.code` (`bytes memory`): code at the Address (can be empty)
- `<address>.codehash` (`bytes32`): the codehash of the Address
- `<address payable>.send(uint256 amount) returns (bool)`: send given amount of Wei to Address, returns `false` on failure
- `<address payable>.transfer(uint256 amount)`: send given amount of Wei to Address, throws on failure

- `type(C).name` (`string`): the name of the contract
- `type(C).creationCode` (`bytes memory`): creation bytecode of the given contract, see Type Information.
- `type(C).runtimeCode` (`bytes memory`): runtime bytecode of the given contract, see Type Information.
- `type(I).interfaceId` (`bytes4`): value containing the EIP-165 interface identifier of the given interface, see Type Information.
- `type(T).min` (`T`): the minimum value representable by the integer type `T`, see Type Information.
- `type(T).max` (`T`): the maximum value representable by the integer type `T`, see Type Information.

Note

When contracts are evaluated off-chain rather than in context of a transaction included in a block, you should not assume that `block.*` and `tx.*` refer to values from any specific block or transaction. These values are provided by the EVM implementation that executes the contract and can be arbitrary.

Note

Do not rely on `block.timestamp` or `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

Note

The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

Note

In version 0.5.0, the following aliases were removed: `suicide` as alias for `selfdestruct`, `msg.gas` as alias for `gasleft`, `block.blockhash` as alias for `blockhash` and `sha3` as alias for `keccak256`.

Note

In version 0.7.0, the alias `now` (for `block.timestamp`) was removed.

Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externally and internally (creates a getter function for storage/state variables)
- `private`: only visible in the current contract
- `external`: only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- `internal`: only visible internally

Modifiers

- `pure` for functions: Disallows modification or access of state.
- `view` for functions: Disallows modification of state.
- `payable` for functions: Allows them to receive Ether together with a call.
- `constant` for state variables: Disallows assignment (except initialisation), does not occupy storage slot.
- `immutable` for state variables: Allows exactly one assignment at construction time and is constant afterwards. Is stored in code.
- `anonymous` for events: Does not store event signature as topic.
- `indexed` for event parameters: Stores the parameter as topic.
- `virtual` for functions and modifiers: Allows the function's or modifier's behaviour to be changed in derived contracts.
- `override` : States that this function, modifier or public state variable changes the behaviour of a function or modifier in a base contract.

Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

`after` , `alias` , `apply` , `auto` , `byte` , `case` , `copyof` , `default` , `define` , `final` , `implements` , `in` , `inline` , `let` , `macro` , `match` , `mutable` , `null` , `of` , `partial` , `promise` , `reference` , `relocatable` , `sealed` , `sizeof` , `static` , `supports` , `switch` , `typedef` , `typeof` , `var` .