**Units and Globally Available Variables**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#units-and-globally-available-variables)

**Ether Units**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#ether-units)

A literal number can take a suffix of wei, gwei or ether to specify a subdenomination of Ether, where Ether numbers without a postfix are assumed to be Wei.

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=YXNzZXJ0KDEgd2VpID09IDEpOwphc3NlcnQoMSBnd2VpID09IDFlOSk7CmFzc2VydCgxIGV0aGVyID09IDFlMTgpOw==)

assert(1 wei == 1);

assert(1 gwei == 1e9);

assert(1 ether == 1e18);

The only effect of the subdenomination suffix is a multiplication by a power of ten.

**Note**

The denominations finney and szabo have been removed in version 0.7.0.

**Time Units**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#time-units)

Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit and units are considered naively in the following way:

* 1 == 1 seconds
* 1 minutes == 60 seconds
* 1 hours == 60 minutes
* 1 days == 24 hours
* 1 weeks == 7 days

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of [leap seconds](https://en.wikipedia.org/wiki/Leap_second). Due to the fact that leap seconds cannot be predicted, an exact calendar library has to be updated by an external oracle.

**Note**

The suffix years has been removed in version 0.5.0 due to the reasons above.

These suffixes cannot be applied to variables. For example, if you want to interpret a function parameter in days, you can in the following way:

[open in Remix](https://remix.ethereum.org/?#language=solidity&version=0.8.20&code=ZnVuY3Rpb24gZih1aW50IHN0YXJ0LCB1aW50IGRheXNBZnRlcikgcHVibGljIHsKICAgIGlmIChibG9jay50aW1lc3RhbXAgPj0gc3RhcnQgKyBkYXlzQWZ0ZXIgKiAxIGRheXMpIHsKICAgICAgICAvLyAuLi4KICAgIH0KfQ==)

function f(uint start, uint daysAfter) public {

if (**block.timestamp** >= start + daysAfter \* 1 days) {

*// ...*

}

}

**Special Variables and Functions**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#special-variables-and-functions)

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

**Block and Transaction Properties**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#block-and-transaction-properties)

* blockhash(uint blockNumber) returns (bytes32): hash of the given block when blocknumber is one of the 256 most recent blocks; otherwise returns zero
* block.basefee (uint): current block’s base fee ([EIP-3198](https://eips.ethereum.org/EIPS/eip-3198) and [EIP-1559](https://eips.ethereum.org/EIPS/eip-1559))
* block.chainid (uint): current chain id
* block.coinbase (address payable): current block miner’s address
* block.difficulty (uint): current block difficulty (EVM < Paris). For other EVM versions it behaves as a deprecated alias for block.prevrandao ([EIP-4399](https://eips.ethereum.org/EIPS/eip-4399) )
* block.gaslimit (uint): current block gaslimit
* block.number (uint): current block number
* block.prevrandao (uint): random number provided by the beacon chain (EVM >= Paris)
* block.timestamp (uint): current block timestamp as seconds since unix epoch
* gasleft() returns (uint256): remaining gas
* msg.data (bytes calldata): 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)

**Note**

The values of all members of msg, including msg.sender and msg.value can change for every **external** function call. This includes calls to library functions.

**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**

The function blockhash was previously known as block.blockhash, which was deprecated in version 0.4.22 and removed in version 0.5.0.

**Note**

The function gasleft was previously known as msg.gas, which was deprecated in version 0.4.21 and removed in version 0.5.0.

**Note**

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

**ABI Encoding and Decoding Functions**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#abi-encoding-and-decoding-functions)

* abi.decode(bytes memory encodedData, (...)) returns (...): ABI-decodes the given data, while 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](https://docs.soliditylang.org/en/v0.8.20/abi-spec.html#abi-packed-mode) of the given arguments. Note that packed 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.encodeWithSignature(string memory signature, ...) returns (bytes memory): Equivalent to abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)
* 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, (...))

**Note**

These encoding functions can be used to craft data for external function calls without actually calling an external function. Furthermore, keccak256(abi.encodePacked(a, b)) is a way to compute the hash of structured data (although be aware that it is possible to craft a “hash collision” using different function parameter types).

See the documentation about the [ABI](https://docs.soliditylang.org/en/v0.8.20/abi-spec.html#abi) and the [tightly packed encoding](https://docs.soliditylang.org/en/v0.8.20/abi-spec.html#abi-packed-mode) for details about the encoding.

**Members of bytes**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#members-of-bytes)

* bytes.concat(...) returns (bytes memory): [Concatenates variable number of bytes and bytes1, …, bytes32 arguments to one byte array](https://docs.soliditylang.org/en/v0.8.20/types.html#bytes-concat)

**Members of string**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#members-of-string)

* string.concat(...) returns (string memory): [Concatenates variable number of string arguments to one string array](https://docs.soliditylang.org/en/v0.8.20/types.html#string-concat)

**Error Handling**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#error-handling)

See the dedicated section on [assert and require](https://docs.soliditylang.org/en/v0.8.20/control-structures.html#assert-and-require) for more details on error handling and when to use which function.

**assert(bool condition)**

causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.

**require(bool condition)**

reverts if the condition is not met - to be used for errors in inputs or external components.

**require(bool condition, string memory message)**

reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.

**revert()**

abort execution and revert state changes

**revert(string memory reason)**

abort execution and revert state changes, providing an explanatory string

**Mathematical and Cryptographic Functions**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#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. 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.

**keccak256(bytes memory) returns (bytes32)**

compute the Keccak-256 hash of the input

**Note**

There used to be an alias for keccak256 called sha3, which was removed in version 0.5.0.

**sha256(bytes memory) returns (bytes32)**

compute the SHA-256 hash of the input

**ripemd160(bytes memory) returns (bytes20)**

compute RIPEMD-160 hash of the input

**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. The function parameters correspond to ECDSA values of the signature:

* r = first 32 bytes of signature
* s = second 32 bytes of signature
* v = final 1 byte of signature

ecrecover returns an address, and not an address payable. See [address payable](https://docs.soliditylang.org/en/v0.8.20/types.html#address) for conversion, in case you need to transfer funds to the recovered address.

For further details, read [example usage](https://ethereum.stackexchange.com/questions/1777/workflow-on-signing-a-string-with-private-key-followed-by-signature-verificatio).

**Warning**

If you use ecrecover, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. In the Homestead hard fork, this issue was fixed for \_transaction\_ signatures (see [EIP-2](https://eips.ethereum.org/EIPS/eip-2#specification)), but the ecrecover function remained unchanged.

This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin have a [ECDSA helper library](https://docs.openzeppelin.com/contracts/4.x/api/utils#ECDSA) that you can use as a wrapper for ecrecover without this issue.

**Note**

When running sha256, ripemd160 or ecrecover on a *private blockchain*, you might encounter Out-of-Gas. This is because these functions are implemented as “precompiled contracts” and only really exist after they receive the first message (although their contract code is hardcoded). Messages to non-existing contracts are more expensive and thus the execution might run into an Out-of-Gas error. A workaround for this problem is to first send Wei (1 for example) to each of the contracts before you use them in your actual contracts. This is not an issue on the main or test net.

**Members of Address Types**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#members-of-address-types)

**<address>.balance (uint256)**

balance of the [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address) in Wei

**<address>.code (bytes memory)**

code at the [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address) (can be empty)

**<address>.codehash (bytes32)**

the codehash of the [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address)

**<address payable>.transfer(uint256 amount)**

send given amount of Wei to [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address), reverts on failure, forwards 2300 gas stipend, not adjustable

**<address payable>.send(uint256 amount) returns (bool)**

send given amount of Wei to [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address), returns false on failure, forwards 2300 gas stipend, not adjustable

**<address>.call(bytes memory) returns (bool, bytes memory)**

issue low-level CALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

**<address>.delegatecall(bytes memory) returns (bool, bytes memory)**

issue low-level DELEGATECALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

**<address>.staticcall(bytes memory) returns (bool, bytes memory)**

issue low-level STATICCALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

For more information, see the section on [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address).

**Warning**

You should avoid using .call() whenever possible when executing another contract function as it bypasses type checking, function existence check, and argument packing.

**Warning**

There are some dangers in using send: The transfer fails if the call stack depth is at 1024 (this can always be forced by the caller) and it also fails if the recipient runs out of gas. So in order to make safe Ether transfers, always check the return value of send, use transfer or even better: Use a pattern where the recipient withdraws the money.

**Warning**

Due to the fact that the EVM considers a call to a non-existing contract to always succeed, Solidity includes an extra check using the extcodesize opcode when performing external calls. This ensures that the contract that is about to be called either actually exists (it contains code) or an exception is raised.

The low-level calls which operate on addresses rather than contract instances (i.e. .call(), .delegatecall(), .staticcall(), .send() and .transfer()) **do not** include this check, which makes them cheaper in terms of gas but also less safe.

**Note**

Prior to version 0.5.0, Solidity allowed address members to be accessed by a contract instance, for example this.balance. This is now forbidden and an explicit conversion to address must be done: address(this).balance.

**Note**

If state variables are accessed via a low-level delegatecall, the storage layout of the two contracts must align in order for the called contract to correctly access the storage variables of the calling contract by name. This is of course not the case if storage pointers are passed as function arguments as in the case for the high-level libraries.

**Note**

Prior to version 0.5.0, .call, .delegatecall and .staticcall only returned the success condition and not the return data.

**Note**

Prior to version 0.5.0, there was a member called callcode with similar but slightly different semantics than delegatecall.

**Contract-related**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#contract-related)

**this (current contract’s type)**

The current contract, explicitly convertible to [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address)

**super**

A contract one level higher in the inheritance hierarchy

**selfdestruct(address payable recipient)**

Destroy the current contract, sending its funds to the given [Address](https://docs.soliditylang.org/en/v0.8.20/types.html#address) and end execution. Note that selfdestruct has some peculiarities inherited from the EVM:

* the receiving contract’s receive function is not executed.
* the contract is only really destroyed at the end of the transaction and revert s might “undo” the destruction.

Furthermore, all functions of the current contract are callable directly including the current function.

**Warning**

From version 0.8.18 and up, the use of selfdestruct in both Solidity and Yul will trigger a deprecation warning, since the SELFDESTRUCT opcode will eventually undergo breaking changes in behaviour as stated in [EIP-6049](https://eips.ethereum.org/EIPS/eip-6049).

**Note**

Prior to version 0.5.0, there was a function called suicide with the same semantics as selfdestruct.

**Type Information**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#type-information)

The expression type(X) can be used to retrieve information about the type X. Currently, there is limited support for this feature (X can be either a contract or an integer type) but it might be expanded in the future.

The following properties are available for a contract type C:

**type(C).name**

The name of the contract.

**type(C).creationCode**

Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the create2 opcode. This property can **not** be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.

**type(C).runtimeCode**

Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of C. If C has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with .creationCode also apply for this property.

In addition to the properties above, the following properties are available for an interface type I:

**type(I).interfaceId:**

A bytes4 value containing the [EIP-165](https://eips.ethereum.org/EIPS/eip-165) interface identifier of the given interface I. This identifier is defined as the XOR of all function selectors defined within the interface itself - excluding all inherited functions.

The following properties are available for an integer type T:

**type(T).min**

The smallest value representable by type T.

**type(T).max**

The largest value representable by type T.

**Reserved Keywords**[**ℑ**](https://docs.soliditylang.org/en/v0.8.20/units-and-global-variables.html#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.