**Introduction to Smart Contracts**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#introduction-to-smart-contracts)

**A Simple Smart Contract**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#a-simple-smart-contract)

Let us begin with a basic example that sets the value of a variable and exposes it for other contracts to access. It is fine if you do not understand everything right now, we will go into more details later.

**Storage Example**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#storage-example)

*// SPDX-License-Identifier: GPL-3.0*

**pragma solidity** >=**0.4.16** <**0.9.0**;

**contract** **SimpleStorage** {

uint storedData;

function set(uint x) public {

storedData = x;

}

function get() public view returns (uint) {

return storedData;

}

}

The first line tells you that the source code is licensed under the GPL version 3.0. Machine-readable license specifiers are important in a setting where publishing the source code is the default.

The next line specifies that the source code is written for Solidity version 0.4.16, or a newer version of the language up to, but not including version 0.9.0. This is to ensure that the contract is not compilable with a new (breaking) compiler version, where it could behave differently. [Pragmas](https://docs.soliditylang.org/en/v0.8.19/layout-of-source-files.html#pragma) are common instructions for compilers about how to treat the source code (e.g. [pragma once](https://en.wikipedia.org/wiki/Pragma_once)).

A contract in the sense of Solidity is a collection of code (its *functions*) and data (its *state*) that resides at a specific address on the Ethereum blockchain. The line uint storedData; declares a state variable called storedData of type uint (*u*nsigned *int*eger of *256* bits). You can think of it as a single slot in a database that you can query and alter by calling functions of the code that manages the database. In this example, the contract defines the functions set and get that can be used to modify or retrieve the value of the variable.

To access a member (like a state variable) of the current contract, you do not typically add the this. prefix, you just access it directly via its name. Unlike in some other languages, omitting it is not just a matter of style, it results in a completely different way to access the member, but more on this later.

This contract does not do much yet apart from (due to the infrastructure built by Ethereum) allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number. Anyone could call set again with a different value and overwrite your number, but the number is still stored in the history of the blockchain. Later, you will see how you can impose access restrictions so that only you can alter the number.

**Warning**

Be careful with using Unicode text, as similar looking (or even identical) characters can have different code points and as such are encoded as a different byte array.

**Note**

All identifiers (contract names, function names and variable names) are restricted to the ASCII character set. It is possible to store UTF-8 encoded data in string variables.

**Subcurrency Example**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#subcurrency-example)

The following contract implements the simplest form of a cryptocurrency. The contract allows only its creator to create new coins (different issuance schemes are possible). Anyone can send coins to each other without a need for registering with a username and password, all you need is an Ethereum keypair.

*// SPDX-License-Identifier: GPL-3.0*

**pragma solidity** ^**0.8.4**;

**contract** **Coin** {

*// The keyword "public" makes variables*

*// accessible from other contracts*

address **public** minter;

mapping(address => uint) public balances;

*// Events allow clients to react to specific*

*// contract changes you declare*

event Sent(address from, address to, uint amount);

*// Constructor code is only run when the contract*

*// is created*

constructor() {

minter = **msg.sender**;

}

*// Sends an amount of newly created coins to an address*

*// Can only be called by the contract creator*

function mint(address receiver, uint amount) public {

require(**msg.sender** == minter);

balances[receiver] += amount;

}

*// Errors allow you to provide information about*

*// why an operation failed. They are returned*

*// to the caller of the function.*

error InsufficientBalance(uint requested, uint available);

*// Sends an amount of existing coins*

*// from any caller to an address*

function send(address receiver, uint amount) public {

if (amount > balances[**msg.sender**])

revert InsufficientBalance({

requested: amount,

available: balances[**msg.sender**]

});

balances[**msg.sender**] -= amount;

balances[receiver] += amount;

emit Sent(**msg.sender**, receiver, amount);

}

}

This contract introduces some new concepts, let us go through them one by one.

The line address public minter; declares a state variable of type [address](https://docs.soliditylang.org/en/v0.8.19/types.html#address). The address type is a 160-bit value that does not allow any arithmetic operations. It is suitable for storing addresses of contracts, or a hash of the public half of a keypair belonging to [external accounts](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#accounts).

The keyword public automatically generates a function that allows you to access the current value of the state variable from outside of the contract. Without this keyword, other contracts have no way to access the variable. The code of the function generated by the compiler is equivalent to the following (ignore external and view for now):

function minter() external view returns (address) { return minter; }

You could add a function like the above yourself, but you would have a function and state variable with the same name. You do not need to do this, the compiler figures it out for you.

The next line, mapping(address => uint) public balances; also creates a public state variable, but it is a more complex datatype. The [mapping](https://docs.soliditylang.org/en/v0.8.19/types.html#mapping-types) type maps addresses to [unsigned integers](https://docs.soliditylang.org/en/v0.8.19/types.html#integers).

Mappings can be seen as [hash tables](https://en.wikipedia.org/wiki/Hash_table) which are virtually initialised such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values. Record what you added to the mapping, or use it in a context where this is not needed. Or even better, keep a list, or use a more suitable data type.

The [getter function](https://docs.soliditylang.org/en/v0.8.19/contracts.html#getter-functions) created by the public keyword is more complex in the case of a mapping. It looks like the following:

function balances(address account) external view returns (uint) {

return balances[account];

}

You can use this function to query the balance of a single account.

The line event Sent(address from, address to, uint amount); declares an [“event”](https://docs.soliditylang.org/en/v0.8.19/contracts.html#events), which is emitted in the last line of the function send. Ethereum clients such as web applications can listen for these events emitted on the blockchain without much cost. As soon as it is emitted, the listener receives the arguments from, to and amount, which makes it possible to track transactions.

To listen for this event, you could use the following JavaScript code, which uses [web3.js](https://github.com/web3/web3.js/) to create the Coin contract object, and any user interface calls the automatically generated balances function from above:

Coin.Sent().watch({}, '', **function**(error, result) {

**if** (!error) {

console.log("Coin transfer: " + result.args.amount +

" coins were sent from " + result.args.**from** +

" to " + result.args.to + ".");

console.log("Balances now:\n" +

"Sender: " + Coin.balances.call(result.args.**from**) +

"Receiver: " + Coin.balances.call(result.args.to));

}

})

The [constructor](https://docs.soliditylang.org/en/v0.8.19/contracts.html#constructor) is a special function that is executed during the creation of the contract and cannot be called afterwards. In this case, it permanently stores the address of the person creating the contract. The msg variable (together with tx and block) is a [special global variable](https://docs.soliditylang.org/en/v0.8.19/units-and-global-variables.html#special-variables-functions) that contains properties which allow access to the blockchain. msg.sender is always the address where the current (external) function call came from.

The functions that make up the contract, and that users and contracts can call are mint and send.

The mint function sends an amount of newly created coins to another address. The [require](https://docs.soliditylang.org/en/v0.8.19/control-structures.html#assert-and-require) function call defines conditions that reverts all changes if not met. In this example, require(msg.sender == minter); ensures that only the creator of the contract can call mint. In general, the creator can mint as many tokens as they like, but at some point, this will lead to a phenomenon called “overflow”. Note that because of the default [Checked arithmetic](https://docs.soliditylang.org/en/v0.8.19/control-structures.html#unchecked), the transaction would revert if the expression balances[receiver] += amount; overflows, i.e., when balances[receiver] + amount in arbitrary precision arithmetic is larger than the maximum value of uint (2\*\*256 - 1). This is also true for the statement balances[receiver] += amount; in the function send.

[Errors](https://docs.soliditylang.org/en/v0.8.19/contracts.html#errors) allow you to provide more information to the caller about why a condition or operation failed. Errors are used together with the [revert statement](https://docs.soliditylang.org/en/v0.8.19/control-structures.html#revert-statement). The revert statement unconditionally aborts and reverts all changes similar to the require function, but it also allows you to provide the name of an error and additional data which will be supplied to the caller (and eventually to the front-end application or block explorer) so that a failure can more easily be debugged or reacted upon.

The send function can be used by anyone (who already has some of these coins) to send coins to anyone else. If the sender does not have enough coins to send, the if condition evaluates to true. As a result, the revert will cause the operation to fail while providing the sender with error details using the InsufficientBalance error.

**Note**

If you use this contract to send coins to an address, you will not see anything when you look at that address on a blockchain explorer, because the record that you sent coins and the changed balances are only stored in the data storage of this particular coin contract. By using events, you can create a “blockchain explorer” that tracks transactions and balances of your new coin, but you have to inspect the coin contract address and not the addresses of the coin owners.

**Blockchain Basics**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#blockchain-basics)

Blockchains as a concept are not too hard to understand for programmers. The reason is that most of the complications (mining, [hashing](https://en.wikipedia.org/wiki/Cryptographic_hash_function), [elliptic-curve cryptography](https://en.wikipedia.org/wiki/Elliptic_curve_cryptography), [peer-to-peer networks](https://en.wikipedia.org/wiki/Peer-to-peer), etc.) are just there to provide a certain set of features and promises for the platform. Once you accept these features as given, you do not have to worry about the underlying technology - or do you have to know how Amazon’s AWS works internally in order to use it?

**Transactions**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#transactions)

A blockchain is a globally shared, transactional database. This means that everyone can read entries in the database just by participating in the network. If you want to change something in the database, you have to create a so-called transaction which has to be accepted by all others. The word transaction implies that the change you want to make (assume you want to change two values at the same time) is either not done at all or completely applied. Furthermore, while your transaction is being applied to the database, no other transaction can alter it.

As an example, imagine a table that lists the balances of all accounts in an electronic currency. If a transfer from one account to another is requested, the transactional nature of the database ensures that if the amount is subtracted from one account, it is always added to the other account. If due to whatever reason, adding the amount to the target account is not possible, the source account is also not modified.

Furthermore, a transaction is always cryptographically signed by the sender (creator). This makes it straightforward to guard access to specific modifications of the database. In the example of the electronic currency, a simple check ensures that only the person holding the keys to the account can transfer money from it.

**Blocks**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#blocks)

One major obstacle to overcome is what (in Bitcoin terms) is called a “double-spend attack”: What happens if two transactions exist in the network that both want to empty an account? Only one of the transactions can be valid, typically the one that is accepted first. The problem is that “first” is not an objective term in a peer-to-peer network.

The abstract answer to this is that you do not have to care. A globally accepted order of the transactions will be selected for you, solving the conflict. The transactions will be bundled into what is called a “block” and then they will be executed and distributed among all participating nodes. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

These blocks form a linear sequence in time, and that is where the word “blockchain” derives from. Blocks are added to the chain at regular intervals, although these intervals may be subject to change in the future. For the most up-to-date information, it is recommended to monitor the network, for example, on [Etherscan](https://etherscan.io/chart/blocktime).

As part of the “order selection mechanism” (which is called “mining”) it may happen that blocks are reverted from time to time, but only at the “tip” of the chain. The more blocks are added on top of a particular block, the less likely this block will be reverted. So it might be that your transactions are reverted and even removed from the blockchain, but the longer you wait, the less likely it will be.

**Note**

Transactions are not guaranteed to be included in the next block or any specific future block, since it is not up to the submitter of a transaction, but up to the miners to determine in which block the transaction is included.

If you want to schedule future calls of your contract, you can use a smart contract automation tool or an oracle service.

**The Ethereum Virtual Machine**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#index-6)

**Overview**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#overview)

The Ethereum Virtual Machine or EVM is the runtime environment for smart contracts in Ethereum. It is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes. Smart contracts even have limited access to other smart contracts.

**Accounts**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#accounts)

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

The address of an external account is determined from the public key while the address of a contract is determined at the time the contract is created (it is derived from the creator address and the number of transactions sent from that address, the so-called “nonce”).

Regardless of whether or not the account stores code, the two types are treated equally by the EVM.

Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage**.

Furthermore, every account has a **balance** in Ether (in “Wei” to be exact, 1 ether is 10\*\*18 wei) which can be modified by sending transactions that include Ether.

**Transactions**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#index-8)

A transaction is a message that is sent from one account to another account (which might be the same or empty, see below). It can include binary data (which is called “payload”) and Ether.

If the target account contains code, that code is executed and the payload is provided as input data.

If the target account is not set (the transaction does not have a recipient or the recipient is set to null), the transaction creates a **new contract**. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the “nonce”). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

**Note**

While a contract is being created, its code is still empty. Because of that, you should not call back into the contract under construction until its constructor has finished executing.

**Gas**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#gas)

Upon creation, each transaction is charged with a certain amount of **gas** that has to be paid for by the originator of the transaction (tx.origin). While the EVM executes the transaction, the gas is gradually depleted according to specific rules. If the gas is used up at any point (i.e. it would be negative), an out-of-gas exception is triggered, which ends execution and reverts all modifications made to the state in the current call frame.

This mechanism incentivizes economical use of EVM execution time and also compensates EVM executors (i.e. miners / stakers) for their work. Since each block has a maximum amount of gas, it also limits the amount of work needed to validate a block.

The **gas price** is a value set by the originator of the transaction, who has to pay gas\_price \* gas up front to the EVM executor. If some gas is left after execution, it is refunded to the transaction originator. In case of an exception that reverts changes, already used up gas is not refunded.

Since EVM executors can choose to include a transaction or not, transaction senders cannot abuse the system by setting a low gas price.

**Storage, Memory and the Stack**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#storage-memory-and-the-stack)

The Ethereum Virtual Machine has three areas where it can store data: storage, memory and the stack.

Each account has a data area called **storage**, which is persistent between function calls and transactions. Storage is a key-value store that maps 256-bit words to 256-bit words. It is not possible to enumerate storage from within a contract, it is comparatively costly to read, and even more to initialise and modify storage. Because of this cost, you should minimize what you store in persistent storage to what the contract needs to run. Store data like derived calculations, caching, and aggregates outside of the contract. A contract can neither read nor write to any storage apart from its own.

The second data area is called **memory**, of which a contract obtains a freshly cleared instance for each message call. Memory is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide. Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word). At the time of expansion, the cost in gas must be paid. Memory is more costly the larger it grows (it scales quadratically).

The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the **stack**. It has a maximum size of 1024 elements and contains words of 256 bits. Access to the stack is limited to the top end in the following way: It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it. All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack. Of course it is possible to move stack elements to storage or memory in order to get deeper access to the stack, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

**Instruction Set**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#instruction-set)

The instruction set of the EVM is kept minimal in order to avoid incorrect or inconsistent implementations which could cause consensus problems. All instructions operate on the basic data type, 256-bit words or on slices of memory (or other byte arrays). The usual arithmetic, bit, logical and comparison operations are present. Conditional and unconditional jumps are possible. Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

For a complete list, please see the [list of opcodes](https://docs.soliditylang.org/en/v0.8.19/yul.html#opcodes) as part of the inline assembly documentation.

**Message Calls**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#message-calls)

Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn can create further message calls.

A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain. If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack. In this case, only the gas sent together with the call is used up. In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions “bubble up” the call stack.

As already said, the called contract (which can be the same as the caller) will receive a freshly cleared instance of memory and has access to the call payload - which will be provided in a separate area called the **calldata**. After it has finished execution, it can return data which will be stored at a location in the caller’s memory preallocated by the caller. All such calls are fully synchronous.

Calls are **limited** to a depth of 1024, which means that for more complex operations, loops should be preferred over recursive calls. Furthermore, only 63/64th of the gas can be forwarded in a message call, which causes a depth limit of a little less than 1000 in practice.

**Delegatecall and Libraries**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#delegatecall-and-libraries)

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context (i.e. at the address) of the calling contract and msg.sender and msg.value do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

This makes it possible to implement the “library” feature in Solidity: Reusable library code that can be applied to a contract’s storage, e.g. in order to implement a complex data structure.

**Logs**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#logs)

It is possible to store data in a specially indexed data structure that maps all the way up to the block level. This feature called **logs** is used by Solidity in order to implement [events](https://docs.soliditylang.org/en/v0.8.19/contracts.html#events). Contracts cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain. Since some part of the log data is stored in [bloom filters](https://en.wikipedia.org/wiki/Bloom_filter), it is possible to search for this data in an efficient and cryptographically secure way, so network peers that do not download the whole blockchain (so-called “light clients”) can still find these logs.

**Create**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#create)

Contracts can even create other contracts using a special opcode (i.e. they do not simply call the zero address as a transaction would). The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller / creator receives the address of the new contract on the stack.

**Deactivate and Self-destruct**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#deactivate-and-self-destruct)

The only way to remove code from the blockchain is when a contract at that address performs the selfdestruct operation. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

**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).

**Warning**

Even if a contract is removed by selfdestruct, it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using selfdestruct is not the same as deleting data from a hard disk.

**Note**

Even if a contract’s code does not contain a call to selfdestruct, it can still perform that operation using delegatecall or callcode.

If you want to deactivate your contracts, you should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

**Precompiled Contracts**[**ℑ**](https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html#precompiled-contracts)

There is a small set of contract addresses that are special: The address range between 1 and (including) 8 contains “precompiled contracts” that can be called as any other contract but their behaviour (and their gas consumption) is not defined by EVM code stored at that address (they do not contain code) but instead is implemented in the EVM execution environment itself.

Different EVM-compatible chains might use a different set of precompiled contracts. It might also be possible that new precompiled contracts are added to the Ethereum main chain in the future, but you can reasonably expect them to always be in the range between 1 and 0xffff (inclusive).