

An introduction to Solidity and smart contracts

This document is not

- a textbook on Solidity, so it does not aim for completeness. The official [Solidity documentation](#) is the most complete source of information;
- a guide to Object Oriented Programming, whose concepts are prerequisites.

This document is

- a summary resulting from combining different online tutorials with a few handbooks and the latest documentation;
- a simple introduction to the syntax of Solidity;
- aiming to make you skilled enough to read and write simple smart contracts for the Ethereum blockchain.

1 Solidity

Solidity is a high-level programming language whose syntax is similar to JavaScript, Python and other object-oriented languages. It is designed to compile code that runs on the **Ethereum Virtual Machine**, which is hosted on **Ethereum nodes** (the computers connected peer-to-peer which constitute the Ethereum network) that are maintaining the blockchain.

Before going into the analysis of a Solidity source code, we need the following tools:

- **Remix**. EVMs do not understand Solidity, but they can read instructions written in **bytecode**. Therefore we need the Solidity compiler to convert Solidity code into bytecode. Throughout this short guide we will make use of [Remix](#), an open source IDE (available both [online](#) and [offline](#)) to test, debug and deploy smart contracts written in Solidity for Ethereum's blockchain.
- **Metamask**. If we own coins or bills, we need a wallet to store them. Metamask is a cryptocurrency **wallet** in form of browser extension, [available](#) for Chrome, Firefox, Opera and Brave. It allows users to register and store keys for Ether (and ERC20 tokens) and to access various Ethereum networks (such as the Main one or the testnet Goerli) directly from the browser. The use of Metamask does not require the download of the entire blockchain: it stores it centrally and helps users connect to their store using the browser. Remember that MetaMask requires no login and does not store your private keys, but the browser does!

- **Ethereum Faucet.** A faucet is a reward system, typically a website or an app, that dispenses prizes in the form of very small fractions of a cryptocurrency. For Ethereum, most of the faucets prizes are around 1000 gwei, 1ETH corresponds to 10^9 gwei, and a gwei corresponds to 10^9 wei. For our purposes, you can use the following [faucet for the Sepolia testnet](#) or some other things you find easy or interesting. You will have to enter your Metamask address and you will be provided with test ETH to use on the testnet.

1.1 The header of a Solidity Source file

In this section we will see how to specify the compiler version to be used, how to import source files and how to comment the code.

pragma. The pragma directive is a method for providing additional information to the compiler, beyond what is conveyed in the language itself. Source files should always begin with a version pragma to avoid incompatibility issues to arise when trying to compile the code with a later compiler versions. The version pragma (at August 2024 the latest version in 0.8.26) consists of a major build number and a minor build number (respectively 8 and 21 in the latest version) and is declared as follows:

```
1 pragma solidity ^0.8.0;
```

A source file with this pragma version does not compile with a compiler earlier than version 0.8.0., and does not work if the compiler version has a larger major build number (condition added using the symbol `^`). As we will see later in an example, it is also possible to specify an interval in which the compiler version must lie in order for the code to compile correctly.

Importing files. In Solidity it is possible to import other source files in a file. The procedure for the offline compilers can be found in the [Solidity documentation](#). Remix provides an automatic remapping for GitHub and automatically retrieves the file over the network. As an example, you can import the ERC721 standard from OpenZeppelin as follows:

```
1 import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
```

Commenting. We have three different ways to comment the code:

1. *single-line*, introduced with a double slash `//` at the beginning of a line;
2. *multi-line*, where all the content between `/*` and `*/` is considered as a comment;
3. *natspec*, that should be only used to document functions and provide a confirmation text which is shown to users when they attempt to invoke a function. They are like multi-line comments with a double asterisk at the beginning of the block:

```
/** ... */
```

Example 1. A very simple example of Solidity code is the following: define a contract (think of it as a class in OOP) that enables the user to set the value of a variable and makes it accessible by other contracts. The contract below allows anyone to store an integer that is accessible by anyone else, without a (feasible) way to prevent any other address from modifying this number. Anyone could call again the function set with a different value and overwrite the current number, with the previous one is still stored in the history of the blockchain. Later on we will see how to impose access restrictions.

```

1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract SimpleStorage {
4      // A contract to store the value of a variable that can be
5      // rewritten and read
6      uint storedData;
7      function set(uint x) public {
8          // the function to overwrite the value of the variable
9          storedData = x;
10     }
11     function get() public view returns (uint) {
12         // the function to retrieve the value of the variable
13         return storedData;
14     }

```

The first line tells us that the source code is written for a Solidity version from 0.5.0 up to, but not including, version 0.7.0. This is to ensure that the contract is not compilable with a new (possibly breaking) compiler version, where it could behave differently. The line `uint storedData;` declares a state variable called storedData of type uint (unsigned integer of 256 bits). The contract defines the functions set and get that can be used to modify or retrieve the value of the variable.

Example 2. A contract which contains a function to compute the properties of a rectangle, i.e. its area and its perimeter.

```

1  pragma solidity >=0.5.0;
2
3  contract ShapeCalculator {
4      /** @dev Calculates a rectangle's surface and perimeter.
5       * @param w Width of the rectangle.
6       * @param h Height of the rectangle.
7       * @return s The calculated surface.
8       * @return p The calculated perimeter.*/
9      function rectangle(uint w, uint h) public pure returns (uint s,
10         uint p) {
11         s = w * h;
12         p = 2 * (w + h);
13     }

```

1.2 Types

Solidity is a statically typed language: for each variable (state and local) we need to specify its type. The concept of “undefined” or “null” values does not exist in Solidity, but newly declared variables always have a default value dependent on its type.

The different types that we can assign to in Solidity are:

- Booleans
- Integers
- Address
- Contract types
- Fixed-size byte arrays
- Dynamically-sized byte arrays
- Address Literals
- Rational and Integer Literals
- String literals and Types
- Hexadecimal Literals
- Enums
- Function Types

What follows is an overview of the types which are peculiar to Solidity. Information about the others (common to almost all typed languages) will be briefly listed in the last subsection and can be found in the [Type section of the documentation](#).

1.2.1 Address

Since version 0.5.0 we have two different nuances of the address type, both holding a 20 byte value (the size of an Ethereum address): `address` and `address payable`, the latter of which has the additional members (also known as *methods* in other OOP languages) `transfer` and `send`. In practice, the only distinction between the two is that `address payable` can receive Ether, `address` can not. An `address payable` can be implicitly converted into an `address`, while the opposite is possible only via an intermediate conversion into a uint160 (an unsigned integer represented with 160 bits).

Three of the most used members of the address types are

- `<address>.balance (uint256)`, to get the balance of the Address in Wei;
- `<address payable>.transfer(uint256 amount)`, to send `amount` Wei to an address. It reverts on failure and forwards 2300 gas stipend, which is not adjustable;
- `<address payable>.send(uint256 amount) returns (bool)`, to send an amount of Wei to an address. It returns `false` on failure (but does not revert) and it forwards 2300 gas stipend, which is not adjustable. The use of `Transfer` is preferable.

See the [Address section](#) for further details and other members of the address type.

1.2.2 Function

A function is used to return some output(s) after performing some operations. The most general function has the following shape:

```
1   function name_of_the_function(<parameter types>) {internal |  
  external | private | public} {pure | view | payable} returns (<return  
  types>) {body_of_the_function}
```

(the curly brackets are only for this document's readability and should not be typed in Solidity). Let us analyse the possible attributes (**modifiers**) defining the function type:

- **parameter types** : it can be empty, otherwise it forces the inputs to the function to be of the specified type;
- **internal** | **external** : internal functions can only be called inside the current contract, while external functions, consisting of an address and a function signature, can be passed via and returned from external function calls. If omitted, the function is regarded as internal by default;
- **private** | **public** : private functions are internal functions with a more restricted access, i.e. they can only be used in contracts declaring them, not even within derived contracts. On the other hand, public functions are accessible to anyone and are part of the contract, with the possibility to call them both internally and externally.
- **view** : it stands for the pledge that the function will not modify the state. In other words, the function will not write to state variables, emit events, create other contracts, self-destruct, send Ethers via calls or call any view or pure function;
- **pure** : it is a condition even stricter than view, since in this case the pledge is not to read from or modify the state. We expand the above list of restrictions by including reading from state variables and accessing the balance of an address, calling any non-pure;
- **payable** | **non-payable** : a non-payable function will reject Ether sent to it (so it cannot be converted to a payable function), while a payable one will accept (and can be regarded as a non-payable thanks to the possibility to send 0 Ether);
- **returns** (<return types>) : the return types must be specified, while the names of the returned values can be omitted; if the function should not return anything, the whole part has to be omitted.

Two of the most diffuse members of this type are **.gas(uint)** and **.value(uint)**, used to respectively send a specific amount of gas and a specific amount of gwei to the target function.

In order to uniquely identify a function we can use the method **.selector** : it returns the first four bytes of the call data for a function call specify the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak-256 hash of the signature of the function.

Example 3. In the following example we have a small contract with two functions. The first function returns the selector of the current function. Here we specify that the function is public and that it must return a 4 bytes value. The second function sends to the current function an amount of 800 gwei, burning 10 units of gas.

```

1  pragma solidity >=0.5.0;
2
3  contract Example {
4      function identifier() public payable returns (bytes4) {
5          return this.f.selector;
6      }
7
8      function send800Gwei() public {
9          this.f.gas(10).value(800)();
10     }
11 }
```

1.2.3 Enums

Enums are the solution provided by Solidity to users who need to create new types. They cannot be implicitly converted, while can be explicitly converted into all integer types.

```

1  pragma solidity >=0.4.16 <0.7.0;
2
3  contract test {
4      enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
5      ActionChoices choice;
6      ActionChoices constant defaultChoice = ActionChoices.GoStraight;
7
8      function setGoStraight() public {
9          choice = ActionChoices.GoStraight;
10     }
11     /* Since enum types are not part of the ABI, the signature of "getChoice"
12        will automatically be changed to "getChoice() returns (uint8)" for all matters external to Solidity.
13        The integer type used is just large enough to hold all enum values,
14        i.e. if you have more than 256 values, 'uint16' will be used and
15        so on.*/
16     function getChoice() public view returns (ActionChoices) {
17         return choice;
18     }
19     function getDefaultChoice() public pure returns (uint) {
20         return uint(defaultChoice);
21     }
```

1.2.4 Other value types

Let us briefly mention the other types of objects you may end up using in your smart contract, together with some of their operators and members.

The **boolean** type `bool` comes in the usual alternative values `true|false`. Negation is expressed via `!`, “AND” via `&&`, “OR” via `||`, equality via `==` and inequality via `!=, <, >, >=, <=`.

The **integer** types `int|uint`, respectively signed and unsigned, come in all variants (in 8-bit steps) from 8-bit `int8|uint8` up to 256-bit `int256|uint256` (which is the one assigned by default with `int|uint` if the size in unspecified).

Comparisons:	<code><=, <, ==, !=, >, >=</code> (evaluate to bool)
Bit operators:	<code>&, , ^</code> (bitwise XOR), <code>~</code> (bitwise negation)
Shift operators:	<code><<</code> (left shift), <code>>></code> (right shift)
Arithmetic operators:	<code>+, -, unary -, *, /, %</code> (modulo), <code>**</code> (exponentiation)

Fixed point numbers are not fully supported by Solidity yet: they can be declared but cannot be assigned to or from. The most general definition is `fixedMxN|ufixedMxN` for the signed/unsigned versions: M (a multiple of 8) is the number of bits used in total to represent the type and N (between 0 and 80) states how many decimal digits are available.

The **fixed-size byte arrays** `bytes1, bytes2 ..., bytes32` hold a sequence of bytes, from one up to 32 respectively. They share the operators with the integer type, but the arithmetic operators are substituted by the index access: if `x` is of type `bytesI`, then `x[k]` for $0 \leq k < I$ returns the k -th byte. The member `.length` returns the length of the array.

Integer literals are formed from a sequence of numbers in the range 0-9 and are interpreted as decimals. **Decimal fraction literals** are formed by a `.` with at least one number on one side, e.g. `1.`, `.1` and `1.3`. Scientific notation is also supported, where the base can have fractions, while the exponent cannot. Examples include `2e10`, `-2e10`, `2e-10`, `2.5e1`. Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal `123_000`, hexadecimal `0x2eff_abde`, scientific decimal notation `1_2e345_678` are all valid.

String literals are written with either double or single-quotes. They support the following escape characters: `\<newline>` (escapes an actual newline), `\\\` (backslash), `\'` (single quote), `\"` (double quote), `\b` (backspace), `\f` (form feed), `\n` (newline), `\r` (carriage return), `\t` (tab), `\v` (vertical tab), `\xNN` (takes a hex value and inserts the appropriate byte), `\uNNNN` (takes a Unicode codepoint and inserts an UTF-8 sequence).

1.2.5 Arrays

In this section we will briefly discuss arrays, although they are a data structure and not a data type. Arrays always refer to groups of values of the same type and make life significantly easier when it comes to perform operations on sets of data. Given `T` the type of data, an array can be initialized with fixed size `k` as `T[k]` or with dynamic size as `T[]`. It is possible to combine the two notions, e.g. to create an array of `k` dynamic arrays of integer type as `int[][][k]`. The notation is from-outside-to-inside: the outermost index refers to the array, the innermost one refers to the element, so that `X[4][2]` stands for the 5th element in the 3rd array of `X` (indices are 0-based).

Some of the main array members are:

- `.length`, which contains the number of elements in the array. This member can be assigned to dynamically-sized arrays to modify their length. If reducing an array performs an implicit delete on each of the removed elements, extending it adds new zero-initialised elements at the tail.
- `.push`, which appends an element (initialised to zero) at the end of a dynamically-sized array.
- `.pop`, which removes an element (initialised to zero) from the end of a dynamically-sized array.

2 Smart contracts

As we said in the first chapter, smart contracts are the analogue of classes in object-oriented languages. In each contract we can find 6 types of elements:

1. **State variables**, which are variables whose values are permanently stored. In ?? we have seen different types to which they can be initialised, together with the crucial aspect of visibility. If a variable is declared as `constant`, then it has to be assigned from an expression which is a constant at compile time.
2. **Enum types**, the way in which a user can define custom types. Go back to ?? for a quick example;
3. **Struct types**, custom defined types that can collect several variables. Note that they differ from Enums since they cannot contain functions but variables only;
4. **Functions**, executable code units that can be called both internally and externally, according to the visibility they have. Go back to ?? for details;
5. **function Modifiers**, that can be used to easily change the behaviour of functions (e.g. check conditions on values, ownerships, ...). See the contract below for an example, and the [Function Modifiers documentation](#) for more information;
6. **Events**, which are nothing more than their analogues in OOP languages. We will not discuss them in these short notes, but you can get an idea visiting the [Events documentation](#).

We conclude this short introduction to Solidity with the analysis (in the form of coded comments) of a smart contract for Safe Remote Purchasing. Try to read it and use the comments inside the code to understand the reasons behind the choice of any parameter, modifier and function construction. You can find more examples in the documentation, under [Solidity by Example](#).

```

1  pragma solidity ^0.5.1;
2
3  contract Purchase {
4      uint public value;
5      address payable public seller;
6      address payable public buyer;
7      enum State { Created, Locked, Inactive }
8      // We build an enum State which consists of 3 possible values for
9      // the state of the contract. Every state variable has a default
10     // value of the first member, 'State.created'
11     State public state;
12
13     /*We require that the variable "value" in msg is an even number.
14     Division will truncate if it is an odd number.*/
15     constructor() public payable {
16         seller = msg.sender;
17         value = msg.value / 2;
18         require((2 * value) == msg.value, "Value has to be even.");
19     }
20
21     modifier condition(bool _condition) {
22         require(_condition);
23         _;
24     }
25     // Define a modifier for a function that only the buyer can call
26     modifier onlyBuyer() {
27         require( msg.sender == buyer, "Only buyer can call this.");
28         _;
29     }
30     // Define a modifier for a function that only the seller can call
31     modifier onlySeller() {
32         require( msg.sender == seller, "Only seller can call this.");
33         _;
34     }
35
36     modifier inState(State _state) {
37         require( state == _state, "Invalid state.");
38         _;
39     }
40
41     event Aborted();
42     event PurchaseConfirmed();
43     event ItemReceived();

```

```
41  /* Define a function to abort the purchase and reclaim the ether.
42   This function can only be called by the seller before the
43   contract is locked.*/
44  function abort() public onlySeller inState(State.Created)
45  {
46    emit Aborted();
47    state = State.Inactive;
48    seller.transfer(address(this).balance);
49  }
50
51  /* Define a function that allows the buyer to confirm the
52   purchase.
53   Transaction has to include '2 * value' ether.
54   The amount of ether will be locked until the function
55   confirmReceived is called.*/
56  function confirmPurchase() public inState(State.Created)
57  condition(msg.value == (2 * value)) payable
58  {
59    emit PurchaseConfirmed();
60    buyer = msg.sender;
61    state = State.Locked;
62  }
63
64  /* Define a function that allows the buyer to confirm that he
65   received the item.
66   This will release the locked amount of ether. */
67  function confirmReceived() public onlyBuyer inState(State.
68  Locked)
69  {
70    emit ItemReceived();
71    /* It is essential to change the state first because otherwise,
72     the contracts called using 'send' below can call in again here.*/
73    state = State.Inactive;
74    /* NOTE: This actually allows both the buyer and the seller to
75     block the refund - the withdraw pattern should be used. */
76    buyer.transfer(value);
77    seller.transfer(address(this).balance);
78  }
79
80 }
```