**Solidity**

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM).

Solidity is statically typed, supports inheritance, libraries and complex user-defined types programming language.

You can use Solidity to create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

**What is Smart Contract?**

A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible.

The concept of smart contracts was first proposed by Nick Szabo in 1994. Szabo is a legal scholar and cryptographer known for laying the groundwork for digital currency.

A Solidity source files can contain an any number of contract definitions, import directives and pragma directives.

pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {

uint storedData;

function set(uint x) public {

storedData = x;

}

function get() public view returns (uint) {

return storedData;

}

}

## Pragma

The first line is a pragma directive which tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality up to, but not including, version 0.6.0.

A pragma directive is always local to a source file and if you import another file, the pragma from that file will not automatically apply to the importing file.

So a pragma for a file which will not compile earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0 will be written as follows −

pragma solidity ^0.4.0;

Here the second condition is added by using ^.

## Contract

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

The line uint stored Data declares a state variable called stored Data of type uint and the functions set and get can be used to modify or retrieve the value of the variable.

## Importing Files

Though above example does not have an import statement but Solidity supports import statements that are very similar to those available in JavaScript.

The following statement imports all global symbols from "filename".

import "filename";

The following example creates a new global symbol symbolName whose members are all the global symbols from "filename".

import \* as symbolName from "filename";

To import a file x from the same directory as the current file, use import "./x" as x;. If you use import "x" as x; instead, a different file could be referenced in a global "include directory".

## Reserved Keywords

Following are the reserved keywords in Solidity −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | after | alias | apply | static |
| auto | case | catch | copyof | typedef |
| default | define | final | immutable | supports |
| implements | in | inline | let | typeof |
| macro | match | mutable | null | switch |
| of | override | partial | promise | unchecked |
| reference | relocatable | sealed | sizeof | try |

# **Solidity - Comments**

Solidity supports both C-style and C++-style comments, Thus −

* Any text between a // and the end of a line is treated as a comment and is ignored by Solidity Compiler.

## Any text between the characters /\* and \*/ is treated as a comment. This may span multiple lines.

## Value Types

Solidity offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

|  |  |  |
| --- | --- | --- |
| **Type** | **Keyword** | **Values** |
| Boolean | bool | true/false |
| Integer | int/uint | Signed and unsigned integers of varying sizes. |
| Integer | int8 to int256 | Signed int from 8 bits to 256 bits. int256 is the same as int. |
| Integer | uint8 to uint256 | Unsigned int from 8 bits to 256 bits. uint256 is the same as uint. |

Solidity supports three types of variables.

* **State Variables** − Variables whose values are permanently stored in a contract storage.
* **Local Variables** − Variables whose values are present till function is executing.
* **Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.

Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes.

* **Public** − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.
* **Internal** − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.
* **Private** − Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

Array:

Array is a data structure, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

* Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

## Declaring Arrays

To declare an array of fixed size in Solidity, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid Solidity data type. For example, to declare a 10-element array called balance of type uint, use this statement −

uint balance[10];

To declare an array of dynamic size in Solidity, the programmer specifies the type of the elements as follows −

type[] arrayName;

## Initializing Arrays

You can initialize Solidity array elements either one by one or using a single statement as follows −

uint balance[3] = [1, 2, 3];

The number of values between braces [ ] can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

uint balance[] = [1, 2, 3];

You will create exactly the same array as you did in the previous example.

balance[2] = 5;

The above statement assigns element number 3rd in the array a value of 5.

## Creating dynamic memory arrays

Dynamic memory arrays are created using new keyword.

uint size = 3;

uint balance[] = new uint[](size);

## Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

uint salary = balance[2];

The above statement will take 3rd element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

## Members

* **length** − length returns the size of the array. length can be used to change the size of dynamic array be setting it.
* **push** − push allows to append an element to a dynamic storage array at the end. It returns the new length of the array.

Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

## Defining a Struct

To define a Struct, you must use the **struct** keyword. The struct keyword defines a new data type, with more than one member. The format of the struct statement is as follows −

struct struct\_name {

type1 type\_name\_1;

type2 type\_name\_2;

type3 type\_name\_3;

}

### **Example**

struct Book {

string title;

string author;

uint book\_id;

}

## Accessing a Struct and its variable

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.

Mapping is a reference type as arrays and structs. Following is the syntax to declare a mapping type.

**Mapping:**

mapping(\_KeyType => \_ValueType)

Where

* **\_KeyType** − can be any built-in types plus bytes and string. No reference type or complex objects are allowed.
* **\_ValueType** − can be any type.

## Considerations

* Mapping can only have type of **storage** and are generally used for state variables.
* Mapping can be marked public. Solidity automatically create getter for it.

Function:

* A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Function Definition

* Before we use a function, we need to define it. The most common way to define a function in Solidity is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

### **Syntax**

The basic syntax is shown here.

function function-name(parameter-list) scope returns() {

//statements

}

Function Modifiers are used to modify the behaviour of a function. For example to add a prerequisite to a function.

First we create a modifier with or without parameter.

contract Owner {

modifier onlyOwner {

require(msg.sender == owner);

\_;

}

modifier costs(uint price) {

if (msg.value >= price) {

\_;

}

}

}

The function body is inserted where the special symbol "\_;" appears in the definition of a modifier. So if condition of modifier is satisfied while calling this function, the function is executed and otherwise, an exception is thrown.

Fallback function is a special function available to a contract. It has following features −

* It is called when a non-existent function is called on the contract.
* It is required to be marked external.
* It has no name.
* It has no arguments
* It can not return any thing.
* It can be defined one per contract.
* If not marked payable, it will throw exception if contract receives plain ether without data.