**Solidity Value Types**

**Table of Contents**

* Solidity Data Types
* Booleans
* Integers
* Address
* Contract Types
* Byte Type
* String Literals and Types
* Enums in Solidity
* Functions

**Solidity Data Types**

Solidity is a statically typed language, and each variable's (state and local) type must be declared. Complex kinds may be created by combining basic solidity types.

The concept of “undefined” or “null” values does not exist in Solidity, but newly declared variables always have a [default value](https://docs.soliditylang.org/en/v0.8.13/control-structures.html#default-value) dependent on its type.

The following are the various types in Solidity:

* Value Types
* Reference Types
* Mapping Types
* Operators

**Value Types**

The following types are also called value types because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments.

Solidity value types consist of the following types:

* **Booleans**
* **Integers**
* **Addresses**
* **Contract types**
* **Bytes type**
* **String type**
* **Enums**

**Booleans**

bool: The possible values are constants true and false.

Boolean Operators:

* ! (logical negation)
* && (logical conjunction, “and”)
* || (logical disjunction, “or”)
* == (equality)
* != (inequality)

**The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.**

**Examples:**

| **bool isPaid = true;** |
| --- |

**Boolean values can be modified within contracts and can be used in both incoming and outgoing parameters and the return value, as shown in the following example:**

| **// SPDX-License-Identifier: MIT  pragma solidity ^0.8.1;  contract BooleanExample {  bool public myBool;   function setMyBool(bool \_myBool) public {  myBool = \_myBool;  }  }** |
| --- |

**Integers**

Integer types in Solidity can be either unsigned or signed.

Unsigned integers are referenced by the uint keyword.

Unsigned integers can be assigned in steps of 8 bits upto 256 bits.

The following are the different types of unsigned integers:

| uint8 = 5;  uint16 = 100;  uint24 = 1000;  uint32 = 104833;  uint40 = 18348;  uint56 = 324234;  uint64 = 134243;  uint128 = 11334;  uint136 = 24;  uint256 = 100000; |
| --- |

The default value for uint type is uint256 and it can store 2^256 values, and because it is unsigned , the maximum value it can store is 2^256-1(zero requires one space).

Signed integers are referenced by the int keyword.

Signed integers can be assigned in steps of 8 bits upto 256 bits.

The following are the different types of signed integers:

| int8 = 5;  int16 = -100;  int24 = -1000;  int32 = 104833;  int40 = -18348;  int56 = -324234;  int64 = 134243;  int128 = -11334;  int136 = 24;  int256 = -100000; |
| --- |

The default value for uint type is int256 and it can store 2^256 values, and because it is signed , it contains positive as well as negative values centered around zero.

The minimum and maximum values for int256 are -(2^256-1)/2 and (2^256-1)/2.

Integer Operators:

* Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
* Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
* Shift operators: << (left shift), >> (right shift)
* Arithmetic operators: +, -, unary - (only for signed integers), \*, /, % (modulo), \*\* (exponentiation)

For an integer type X, you can use type(X).min and type(X).max to access the minimum and maximum value representable by the type.

#### Comparisons:

The value of a comparison is the one obtained by comparing the integer value.

This means, for example ~int256(0) == int256(-1).

**Address**

The address type comes in two categories:

Address: A 20 byte value (size of an Ethereum address).

Address payable: The same as the address but with the extra members transfer and send

This difference is made because a simple address is not designed to receive Ether, for example because it is a smart contract that was not built to accept Ether.

Type conversion:

Conversions from address payable to address are allowed implicitly, but must be done explicitly through payable(<address>).

uint160, integer literals, bytes20, and contract types support explicit address conversions.

Only address and contract-type expressions may be explicitly changed to address using the explicit conversion payable (...). This conversion is only permitted if the contract can receive Ether, i.e The contract has a receive or payable fallback method. Payable(0) is an exception to this rule.

Important: If you need a variable of type address and plan to send Ether to it, then declare its type as address payable to make this requirement visible. Also, try to make this distinction or conversion as early as possible.

Address Operators:

* <=, <, ==, !=, >= and >

Address Member Functions:

balance and transfer

Balance:

The property balance may be used to check an address's balance and the transfer function to send Ether (in wei) to a payable address:

| address payable x = payable(0x123); address myAddress = address(this);  if (x.balance 10 && myAddress.balance >= 10)  {  x.transfer(10);  } |
| --- |

if (x.balance 10 && myAddress.balance >= 10) x.transfer(10);

The transfer function fails if the current contract's balance is insufficient or the recipient account rejects the Ether transfer. If the transfer fails, it reverts.

Important:

It will be run together with the transfer call if x is a contract address (more precisely, its Receive Ether Function or its Fallback Function) (this is a feature of the EVM and cannot be prevented). That execution will fail if it runs out of gas or else fails, and the Ether transfer will be reversed.

Send:

Send is the opposite of transfer. If the execution fails, the current contract persists, but send returns false.

Warning:

Using send might be risky. It fails if the recipient runs out of gas. Always verify the return value of a send, utilize a transfer or even better: set up a pattern in which the receiver withdraws their money.

call, delegatecall and staticcall

If you need to work with contracts that don't follow the ABI, or if you want to get more control over how they're written, the functions call, delegatecall, and staticcall are there. It's the same thing with each one. They all take one byte of memory and return the success condition (a bool) and the data that was returned (bytes memory). Abi's encode, encodePacked, encodeWithSelector, and encodeWithSignature functions can be used to encode data that is organized in a way that can be read.

It is possible to adjust the supplied gas with the gas modifier:

| address(address\_name).call  {gas:1000000}  (abi.encodeWithSignature("register(string)", "MyName")); |
| --- |

Similarly, the supplied Ether value can be controlled too:

| address(address\_name).call  {value: 1 ether}  (abi.encodeWithSignature("register(string)", "MyName")); |
| --- |

Lastly, these modifiers can be combined. Their order does not matter:

| address(address\_name).call{gas: 1000000,  value: 1 ether}  (abi.encodeWithSignature("register(string)", "MyName")); |
| --- |

Using the function delegatecall is similar; the difference is that just the code of the provided address is utilized, and all other characteristics (storage, balance, and so on) are obtained from the current contract, rather than the previous contract. In order to make use of library code that is contained in another contract, the delegatecall function must be called. The user must check that the arrangement of

storage in both contracts is acceptable for the usage of delegatecall before proceeding with the implementation.

Warning:

All of these functions are low-level functions, and they should only be used with extreme caution when necessary. To be more specific, any unknown contract may be malicious, and if you call it, you pass over power to that contract, which may in turn call back into your contract, so be prepared for changes to your state variables when the call is completed. Contact with other contracts is often accomplished by calling a function on the contract object (e.g., f() on contract).

**code and codehash**

You may look for any smart contract in the code that has been deployed. Use the .code command to get the EVM bytecode as a bytes memory, which may or may not include any data. Codehash is a command that returns the Keccak-256 hash of a code (as a bytes32). It should be noted that using addr.codehash is less expensive than using keccak256 (addr.code).

**Address Literals:**

Hexadecimal literals that pass the address checksum test, for example “0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF” are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. You can prepend (for integer types) or append (for bytesNN types) zeros to remove the error.

**Contract Types**

Every contract has its own type. Contracts may be implicitly converted into the contracts from which they derive. Contracts may be explicitly converted to and from the address type by using the address type conversion function.

Using the contract type as a local variable, you may invoke functions that are associated with the contracted type. Make certain that it is assigned from a location that has the same contract type.

Contracts may also be formed and instantiated (which means they are newly created). More information may be found in the section under "Contracts through new."

* The data representation of a contract is the same as the data representation of the address type, and this type is also utilized in the Address Based Identification (ABI).
* Contracts do not provide any assistance to operators.
* The members of contract types are the external functions of the contract, as well as any state variables that have been declared as publicly accessible.
* When dealing with a contract C, you may use type(C) to get information about the contract's type.

**Byte Type**

The byte types in Solidity are comprised of fixed sized and dynamic sized byte arrays.

Fixed size byte arrays:

The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32.

Operators:

Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)

Shift operators: << (left shift), >> (right shift)

Index access: If x is of type bytesn, then x[k] for 0 <= k < n returns the k th byte (read-only).

Members:

The .length function yields the fixed length of the byte array (read-only).

**Dynamically-sized byte array:**

The bytes data type in Solidity is a byte[] array with a dynamically changing size. Because it is dynamically sized, the length of this type can increase and shrink as necessary. As opposed to the bytes1, bytes2,..., bytes31, and bytes32 types, the bytes type stores densely packed data, whereas the bytesN type does not

In the case of the bytes type variables, an empty string is used as their first value. Using the Remix IDE, you can see that it returns 0x, which means that it is a byte[] array with a length of 0.

Although there is no direct operator support for bytes, you may find yourself needing to compare two bytes variables from time to time.

**String Type and String Literals**

Solidity supports both double quotation (") and single quote (') when referencing a string literal. It makes it possible to declare a variable of type String by providing the data type string.

| pragma solidity ^0.5.0;  contract SolidityTest {  string data = "test"; } |
| --- |

In the above example, the string literal "test" is represented by the string variable data. Because string operations use more gas than byte operations, it is preferable to employ byte types rather than String operations wherever possible instead of String operations. Solidity includes built-in conversion between bytes and strings, as well as the other way around. In Solidity, we can quickly and efficiently assign a String literal to a byte32 type variable. Solidity treats it as if it were a byte32 literal.

| pragma solidity ^0.5.0;  contract SolidityTest {  bytes32 data = "test"; } |
| --- |

Additionally, string literals also support the following escape characters:\

* \<newline> (escapes an actual newline)
* \\ (backslash)
* \' (single quote)
* \" (double quote)
* \n (newline)
* \r (carriage return)
* \t (tab)
* \xNN (hex escape, see below)
* \uNNNN (unicode escape, see below)

**Enums in Solidity**

Enums limit the number of possible values for a variable to a small number of predefined options. The values contained within this enumerated list are referred to as enums.

It is possible to limit the number of bugs in your code by making use of enums in your code.

As an example, if we were to investigate an application for a fresh juice shop, it would be possible to limit the glass size to three options: small, medium, and big. This would ensure that no one would be able to order a size other than small, medium, or large because of the restriction.

Enum examples:

| pragma solidity ^0.5.0;  contract test {  enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }  FreshJuiceSize choice;  FreshJuiceSize constant defaultChoice = FreshJuiceSize.MEDIUM;   function setLarge() public {  choice = FreshJuiceSize.LARGE;  }  function getChoice() public view returns (FreshJuiceSize) {  return choice;  }  function getDefaultChoice() public pure returns (uint) {  return uint(defaultChoice);  } } |
| --- |

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.10;  contract Enum {  // Enum representing shipping status  enum Status {  Pending,  Shipped,  Accepted,  Rejected,  Canceled  }   // Default value is the first element listed in  // definition of the type, in this case "Pending"  Status public status;   // Returns uint  // Pending - 0  // Shipped - 1  // Accepted - 2  // Rejected - 3  // Canceled - 4  function get() public view returns (Status) {  return status;  }   // Update status by passing uint into input  function set(Status \_status) public {  status = \_status;  }   // You can update to a specific enum like this  function cancel() public {  status = Status.Canceled;  }   // delete resets the enum to its first value, 0  function reset() public {  delete status;  } } |
| --- |