LESSON INTRO 5 – Variables

University of West Attica
Department of Electrical and Electronics Engineering
Ioannis Christidis
Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish?

We will see the *variable types of solidity*.

Understand how variables work in SCs.

Learn the *basics of operations available in each type*.

Although not required it is a good practice to write the code yourself to test it as we progress.

Integers (`int`)

`int` in Solidity is a *signed integer type*. It can store *both positive and negative whole numbers*.

You can define the *size of the integer* by specifying it after the `int` (e.g., `int64`).

`int` is by default `256 bits` wide. This means that `int = int256`.

There are also smaller sizes available: `int8, int16, int32, int64, int128, int256`.

As a size comparison, the minimum and maximum values from `int8` and `int256` are:

`-128 <= int8 <= 127, -2^255 <= int256 <= 2^255 - 1`

Its default value is zero (`0`).

```
int number; // will be 0
int8 negativeNumber = -3;
int32 negativeNumber = 18;
```

Integers (2)

You generally want to avoid using integers and stick unsigned integers (`uint`) because of the following reasons:

- *<u>Overflow and Underflow</u>*: Before `Solidity 0.8.0`, `int` values could silently wrap around if they exceeded their range. After `version 0.8.0`, *<u>overflow and underflow checks are automatically enforced</u>*, and the SC will revert if the value exceeds the allowable range.

- *<u>Gas Costs</u>*: `int` operations are more expensive (not much) than `uint` due to the additional logic for handling the sign.

When to use `int`?

- When your <u>application logic requires negative numbers</u>. (e.g., temperature measuring)

Why use smaller `int`?

- When stored in an *`array` or `struct`*, *<u>smaller types can save space by packing multiple smaller variables together</u>*. (we will learn `arrays` and `structs` later)

Unsigned Integers (`uint`)

`uint` stands for *unsigned integer*, meaning it can *only store non-negative whole numbers* (0 and above).

You can define the *size of the unsigned integer* by specifying it after the `uint` (e.g., `uint64`).

`uint` is by default `256 bits` wide. This means that `uint = uint256`.

There are also smaller sizes available (e.g., `uint8`, `uint16`, …, `uint256`).

As a size comparison, the minimum and maximum values from `uint8` and `uint256` are:

`0 <= uint8 <= 255, 0 <= uint256 <= 2^256 - 1`

Its *default value is zero (`0`)*.

```
uint number; // will be 0
uint32 negativeNumber = 18;
```

Unsigned Integers (2)

You generally prefer to use unsigned integers and avoid signed integers (`int`) because of the following reasons:

- *Larger Range* (e.g. `uint8` can be up to `255` while an `int8` only goes up to `127`)
- Using `uint` *avoids the complexity of handling negative numbers in SC logic.*
- *Gas Efficiency*: Since `uint` doesn't require handling a sign, operations are cheaper in terms of gas compared to `int`.

Why use smaller `uint`?

- When stored in an `array` *or* `struct`, *smaller types can save space by packing multiple smaller variables together.* (we will learn `arrays` and `structs` later).

Be aware of overflows and underflows since they can break your code's functionality. (e.g. when you have a function that increments a `uint8` and it reaches the maximum value, the function will be unusable since the transaction will revert every time you try to call it.)

Operations of `int` and `uint`

Arithmetic Operations:

- *Addition* (+): Adds two integers or unsigned integers.
- *Subtraction* (`-`): Subtracts one integer unsigned integer from another.
- *Multiplication* (`*`): Multiplies two integers or unsigned integers.
- *Division* (`/`): Divides two integers or unsigned integers. Note: Division truncates the decimal part for integers or unsigned integers.
- *Modulo* (%): Returns the remainder of a division.

Comparison Operations:

- *Equality* (==): Checks if two integers or unsigned integers are equal.
- *Inequality* (`!=`): Checks if two integers or unsigned integers are not equal.
- *Greater than* (`>, >=`): Checks if one integer or unsigned integer is greater than another.
- *Less than* (`<, <=`): Checks if one integer or unsigned integer is less than another.

Booleans (`bool`)

`bool` is the *Boolean data type in Solidity*.

It can either be `true` or `false`.

Boolean variables are used for conditions, flags, and decision-making in SCs.

The default value of a `bool` variable is `false`.

Solidity allocates 1 byte (8 bits) to store a single boolean variable in storage.

Boolean variables *can be very gas-efficient when they are packed in a `struct` or `array` as in that case, they only store a single bit of data*.

```
bool isFalse; // will be false
bool isTrue = true;
```

Operations of bool

Logical Operations:

- *AND* (&&): Returns true if both operands are true.
- *OR* (||): Returns true if at least one operand is true.
- *NOT* (!): Reverses the boolean value (true becomes false, and vice versa).

Comparison Operations:

- *Equality* (==): Checks if two boolean values are equal.
- *Inequality* (!=): Checks if two boolean values are not equal.

Strings (`string`)

A `string` in Solidity is a *dynamically sized UTF-8 encoded sequence of characters*.

`Strings` *are dynamically sized, meaning their length can vary.* This differentiates them from fixed-size types like `uint`.

`strings` are stored as a *sequence of bytes* in Solidity.

The default value of an uninitialized string is an empty string ("").

Avoid using strings:

- Strings *consume significant gas because they are dynamically sized and require more storage space compared to fixed-size types* (**!!1 BYTE EACH CHARACTER!!**).
- String *operations like concatenation, comparison, or manipulation are gas-intensive*. It's often better to handle strings off-chain and store or use the result on-chain.
- Consider using the `bytes` type instead (We will talk about it after the strings)

Strings (2)

To use a `string` as parameter or for a local variable in a function you will need to store it in `memory`. We will talk about `memory` in the future but for now here is an example of how to use a `string` in `memory`.

```
string public storedWord; //Empty string ""

function setWord(string memory _storedWord) public {
  storedWord = _storedWord;
}

function setWordWithoutParameter() public {
  string memory wordToBeStored = "word";
  storedWord = wordToBeStored;
}
```

Bytes bytes

Bytes is a dynamic array of `bytes` used to store raw binary data.

Solidity also provides a *fixed-size version:* `bytes1` *to* `bytes32`, where the size is fixed and cannot be changed after declaration.

The *default value for both bytes and fixed-size bytes is an array filled with zeros*.

For `bytes32: 0x00..0` and for `bytes`: "" (*empty array*)

```
//dynamic size bytes
bytes public data;
function setData(bytes memory _data) public {
   data = _data;
}
function getData() public view returns (bytes memory) {
  return data;
}

//fixed size bytes
bytes4 public fixedData;
function setFixedData(bytes4 _data) public {
  fixedData = _data; // Store exactly 4 bytes
}
```

Operations of `string` and `bytes`

Comparison: *Fixed-size bytes can be compared directly using the standard comparison operators* (`==`, `!=`, etc.) because Solidity supports direct equality checks for these types. *Comparison of strings and dynamic size bytes cannot be directly compared using the == operator because the language does not support native string comparison like some other programming languages.* Instead, *string comparison is done using hashing*. Bellow you see an example of comparing strings or dynamic size bytes a and b. (Do not worry about the meaning of *keccak256(abi.encodePacked())* for now, we will not need it for this course. Just know that with it, we can hash anything.)

```
bool isEqual = keccak256(abi.encodePacked(a)) == keccak256(abi.encodePacked(b));
```

Concatenation: As of version 0.8.12 of solidity `concat` support has been added for strings.

```
function concatenateStings(string memory a, string memory b) public pure returns (string memory) {
  return string.concat(a,b);
}
```

Concatenation of bytes must be done similarly to their comparison. This means that we must use `abi.encodePacked`.

```
function concatenateMultiple(bytes memory a, bytes memory b, bytes memory c) public pure returns (bytes memory) {
 return abi.encodePacked(a, b, c);
}
```

Addresses `address`

An `address` in Solidity is a `20-byte (160-bit)` value that *represents an Ethereum address.*

Ethereum addresses can be externally owned accounts (EOAs) (controlled by private keys) or SC addresses (deployed SCs).

Solidity has *two* `address` *types*:

- *The standard* `address` *type* that provides basic methods for interaction, like balance checks or sending ETH.
- `Payable address`: A specialized address type that *can receive and send ETH*. Use `payable address` when working with ETH transfers.

The default value of an `address` is `address(0) = 0x00...0`.

Along with `uint`, `address` is the most used types when writing SCs.

```
// Address
address public myAddress = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
// Payable address
address payable public myAddress = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
```

Outro

Great, now you know the basics of solidity variable.

In the next lesson we will start building our project.

The project will be IoT based!

Test: Write a SC like `SimpleStorage` for each type.

We already did it for uint!

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.26;
//  SCs are mainly built with the getter and setter functionality
SC SimpleStorage {
    // This is unsigned integer. Since its value is not defined it is considered 0
    uint256 favouriteNumber;
    // This is the setter function that sets the number to a specific value
    function setFavouriteNumber(uint256 _favouriteNumber) public { // writes on the blockchain (costs gas)
        favouriteNumber = _favouriteNumber;
    }
    // this is the getter function that retrieves the value of the number
    function getFavouriteNumber() public view returns (uint256) { // reads from the blockchain
        return  favouriteNumber;
    }
}
```