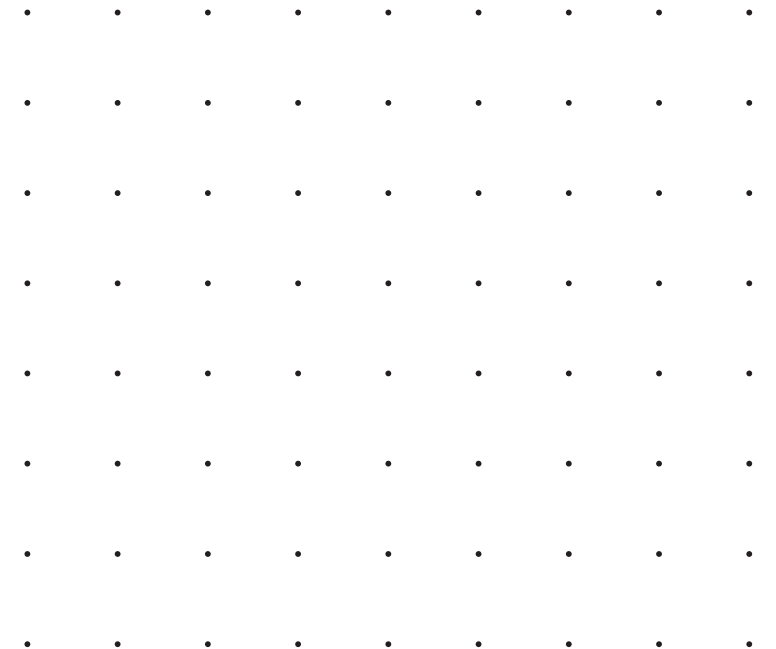


Week 5 – Introduction to Solidity

Dr. Minfeng Qi

28 Aug 2023



• • • • •
• • • • •

Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne’s Australian campuses are located in Melbourne’s east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne’s Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.

• •
• •

• • • • • • • • • • • • • •
• • • • • • • • • • • • • •



What is Solidity?

Solidity is a coding language specifically designed for crafting smart contracts on the Ethereum Virtual Machine (EVM)

```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6  * @title Storage
7  * @dev Store & retrieve value in a variable
8  * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
9  */
10 contract Storage {
11
12     uint256 public number;
13
14     /**
15      * @dev Store value in variable
16      * @param num value to store
17      */
18     function store(uint256 num) public {
19         number = num;
20     }
21
22     /**
23      * @dev Return value
24      * @return value of 'number'
25      */
26     function retrieve() public view returns (uint256){
27         return number;
28     }
29
30     function ifElseTest(uint256 _number) public pure returns(bool){
31         if(_number == 0){
32             return(true);
33         }else{
34             return(false);
35         }
36     }
37 }
```

The **first line** is a comment, which denotes the software license (license identifier) used by the program. We are using the MIT license. If you do not indicate the license used, the program can compile successfully but will report a warning during compilation. Solidity's comments are denoted with "//", followed by the content of the comment (which will not be run by the program).

Solidity - Syntax

```
3  pragma solidity >=0.7.0 <0.9.0;
```

The second line declares the Solidity version used by the source file, because the syntax of different versions is different. This line of code means that the source file will not allow compilation by compilers version lower than v0.7.0 and not higher than v0.9.0.

```
10  contract Storage {  
11  
12      uint256 public number;  
13
```

Lines 10 and 12 are the main body of the smart contract. Line 10 creates a contract with the name Storage. Line 12 is the content of the contract. Here, we created a uint variable called number which has a default value of 0.

• • • • • • • • • •
• • • • • • • • • •

Value Type - Boolean

Value Type: This include boolean, integer, address, fixed-size byte arrays, and enumeration. These variables directly pass values when assigned.

Boolean is a binary variable, and its values are true or false.

// Boolean

```
bool public _bool = true;
```

Operators for Boolean type include:

- ! (logical NOT)
- && (logical AND)
- || (logical OR)
- == (equality)
- != (inequality)



Value Type - Integers

Integers types in Solidity includes signed integer int and unsigned integer uint. It can store up to a 256-bit integers or data units.

// Integer

int public _int = -1; *// integers including negative numbers*

uint public _uint = 1; *// non-negative numbers*

uint256 public _number = 20220330; *// 256-bit positive integers*


Commonly used integer operators include:

- Inequality operator (which returns a Boolean) : <=, <, ==, !=, >=, >
- Arithmetic operator : +, -, *, /, % (modulo), ** (exponent)



Value Type - Addresses

Definition: Addresses in Solidity represent Ethereum addresses. They are a fundamental type that allows you to perform specific operations.

From: [0x333333f332a06ECB5D20D35da44ba07986D6E203](#) (MEV Builder: 0x333...203) 

To: [0xcF6715Cbd7a900BBbde15E8d1dD6779815258e60](#) 

Key Properties:

- 20 Bytes Long: Addresses are essentially 20-byte long values that denote a specific account on the Ethereum network.
- Hold Ether: An address can hold ether, the native cryptocurrency of the Ethereum platform.
- Smart Contracts and Externally Owned: Addresses can represent either a smart contract or an externally owned account (EOA).

Examples:

- address public ownerAddress;
- address payable public beneficiaryAddress;



Value Type - Fixed-size byte arrays

Byte arrays in Solidity come in two types:

- Fixed-length byte arrays: belong to value types, including byte, bytes8, bytes32, etc, depending on the size of each element (maximum 32 bytes). The length of the array can not be modified after declaration.
- Variable-length byte arrays: belong to reference type, including bytes, etc. The length of the array can be modified after declaration. We will learn more detail in later chapters

// Fixed-size byte arrays

```
bytes32 public _byte32 = "Solidity";
```

```
bytes1 public _byte = _byte32[0];
```



Value Type - Enumeration

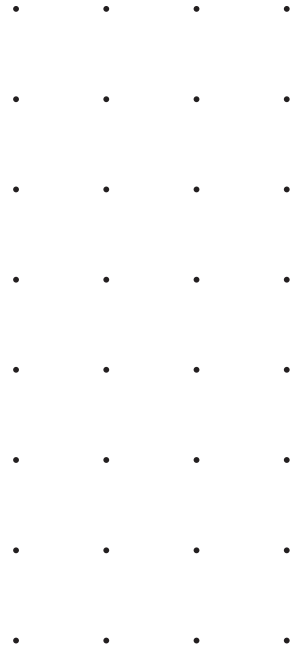
Enumeration (enum) is a user-defined data type within Solidity. It is mainly used to assign names to uint, which keeps the program easy to read.

```
// Let uint 0, 1, 2 represent Buy, Hold, Sell  
enum ActionSet { Buy, Hold, Sell }  
// Create an enum variable called action  
ActionSet action = ActionSet.Buy;
```

It can be converted to uint easily:

```
// Enum can be converted into uint  
function enumToUint() external view returns(uint){  
    return uint(action);  
}
```

NOTE: enum is a less popular type in Solidity.



Reference Type - Array

An array is a variable type commonly used in Solidity to store a set of data (integers, bytes, addresses, etc.). There are two types of arrays: fixed-sized and dynamically-sized arrays:

- fixed-sized arrays: The length of the array is specified at the time of declaration. An array is declared in the format `T[k]`, where `T` is the element type and `k` is the length.

// fixed-length array

```
uint[8] array1;  
byte[5] array2;  
address[100] array3;
```

- Dynamically-sized array(dynamic array) : Length of the array is not specified during declaration. It uses the format of `T[]`, where `T` is the element type.

// variable-length array

```
uint[] array4;  
byte[] array5;  
address[] array6;  
bytes array7;
```

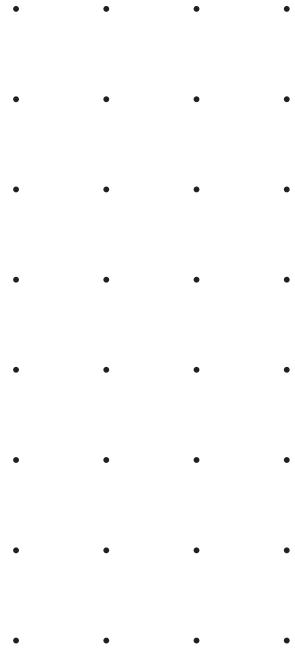


Reference Type - Struct

You can define new types in the form of struct in Solidity. Elements of struct can be primitive types or reference types. And struct can be the element for array or mapping.

```
// struct  
struct Student{  
    uint256 id;  
    uint256 score;  
}
```

```
Student student; // Initially a student structure
```



Mapping Type - Mapping

With mapping type, people can query the corresponding Value by using a Key. For example, a person's wallet address can be queried by their id.

The format of declaring the mapping is `mapping(_KeyType => _ValueType)`, where `_KeyType` and `_ValueType` are the variable types of Key and Value respectively. For example:

```
mapping(uint => address) public idToAddress; // id maps to address
mapping(address => address) public swapPair; // mapping of token pairs, from
address to address
```



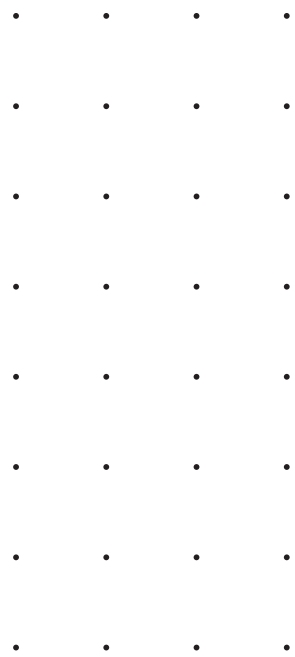
Data Storage

There are three types of data storage locations in solidity: storage, memory and calldata. Gas costs are different for different storage locations.

The data of a storage variable is stored on-chain, similar to the hard disk of a computer, and consumes a lot of gas; while the data of memory and calldata variables are temporarily stored in memory, consumes less gas.

General usage:

- 1) storage: The state variables are storage by default, which are stored on-chain.
- 2) memory: The parameters and temporary variables in the function generally use memory label, which is stored in memory and not on-chain.
- 3) calldata: Similar to memory, stored in memory, not on-chain. The difference from memory is that calldata variables cannot be modified, and is generally used for function parameters



Control Flow – if-else / for loop

1.if-else

```
function ifElseTest(uint256 _number) public pure returns(bool){  
    if(_number == 0){  
        return(true);  
    }else{  
        return(false);  
    }  
}
```

2.for loop

```
function forLoopTest() public pure returns(uint256){  
    uint sum = 0;  
    for(uint i = 0; i < 10; i++){  
        sum += i;  
    }  
    return(sum);  
}
```



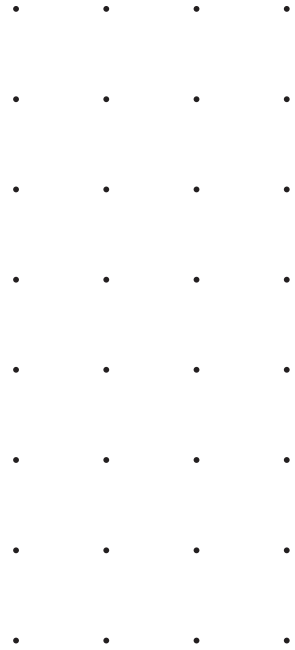
Control Flow – while loop / do-while loop

3.while loop

```
function whileTest() public pure returns(uint256){
    uint sum = 0;
    uint i = 0;
    while(i < 10){
        sum += i;
        i++;
    }
    return(sum);
}
```

4.do-while loop

```
function doWhileTest() public pure returns(uint256){
    uint sum = 0;
    uint i = 0;
    do{
        sum += i;
        i++;
    }while(i < 10);
    return(sum);
}
```



Function

Here's the format of a function in Solidity:

```
function <function name>(<parameter types>) [internal|external] [pure|view|payable]  
[returns (<return types>)]
```

- 1) **function**: To write a function, you need to start with the keyword function.
- 2) **<function name>**: The name of the function.
- 3) **<parameter types>**: The input parameter types and names.
- 4) **[internal|external|public|private]**: Function visibility specifiers. There are 4 kinds of them public is the default visibility if left empty:
 - **public**: Visible to all.
 - **private**: Can only be accessed within this contract, derived contracts cannot use it.
 - **external**: Can only be called from other contracts. But can also be called by `this.f()` inside the contract, where `f` is the function name.
 - **internal**: Can only be accessed internal and by contracts deriving from it.
- 5) **[pure|view|payable]**: Keywords that dictate a Solidity functions behavior.
- 6) **[returns (<return types>)]**: Return variable types and names.

Function Output

There are two keywords related to function output: return and returns:

- returns is added after the function name to declare variable type and variable name;
- return is used in the function body and returns desired variables.

// returning multiple variables

```
function returnMultiple() public pure returns(uint256, bool, uint256[3] memory){  
    return(1, true, [uint256(1),2,5]);  
}
```

In the above code, the returnMultiple() function has multiple outputs: returns (uint256, bool, uint256[3] memory) , and then we specify the return variables/values in the function body with return (1, true, [uint256 (1), 2,5]) .



Modify Contract State

The contract state variables are stored on block chain, and gas fee is very expensive. If you don't rewrite these variables, you don't need to pay gas. You don't need to pay gas for calling *pure* and *view* functions.

The following statements are considered modifying the state:

- 1) Writing to state variables.
 - 2) Emitting events.
 - 3) Creating other contracts.
 - 4) Using selfdestruct.
 - 5) Sending Ether via calls.
 - 6) Calling any function not marked view or pure.
 - 7) Using low-level calls.
 - 8) Using inline assembly that contains certain opcodes.
- *pure* : Functions containing pure keyword cannot read nor write state variables on-chain.
 - *view* : Functions containing view keyword can read but cannot write on-chain state variables.



