
Solidity
Programming
Parsa Noori - Fall 2023

SimpleStorage.sol Contract

SPDX-License Identifier

SPDX-License-Identifier is a standardized identifier used in source code files to specify the licensing terms under which the code is released. SPDX stands for "Software Package Data Exchange," and it is a widely accepted standard for documenting and communicating the licenses associated with software and open-source projects.

```
// SPDX-License-Identifier: <SPDX-License>
// SPDX-License-Identifier: MIT
// SPDX-License-Identifier: Apache-2.0
// SPDX-License-Identifier: GPL-3.0-or-later
```

Pragma Solidity

It is a crucial directive in Solidity used to specify the compiler version for a smart contract, ensuring that the contract is compiled and executed correctly, and it helps maintain code compatibility and consistency across different compiler versions.

```
pragma solidity ^0.8.0;
pragma solidity 0.8.0;
pragma solidity ~0.8.0;
pragma solidity 0.7.6 || 0.8.0;
pragma solidity >= 0.7.0 <0.8.9;
```

What's the tradeoff?
Nodes support vs lower gas fess.

The syntax

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MyContract{
    // the contract code goes here
}

interface MyInterface{
    // the interface code goes here
}
```

It’s near to the concepts of OOP in languages such as Java, Python and C++.
Classes → Contracts
Interfaces → Interfaces

Value types

1. Boolean
2. int/uint
3. int8 to int256 (incremented by 8)
4. uint8 to uint256 (incremented by 8)
5. fixed/unfixed
6. fixedMxN
7. address

Variable Types: 1 State Variables

State Variables:
State variables are stored on the Ethereum blockchain and represent the state of a contract.
They are declared outside of functions and have global scope within the contract.
Changes to state variables persist across function calls and transactions.
State variables are typically used to store contract-specific data.
Example: uint256 public totalSupply;

Variable Types: 2 Local Variable

Local Variables:
Local variables are declared within a function and have function-level scope. They are used for temporary storage during the execution of a function and are discarded when the function exits.
Local variables are stored in the function's memory by default.
Example: uint256 balance = 100;

Variable Types: 3 Special Variables

Solidity provides some special variables like `msg.sender`, `msg.value`, `block.number`, and others that provide information about the current transaction and the Ethereum environment.
These special variables are not user-declared but are available for use within contract functions.

Variable scopes

1. Public:
   Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.
2. Internal:
   Internal state variables can be accessed only internally from the current contract or contract inheriting from it.
3. Private:
   Private state variables can be accessed only internally from the current contract they are defined not in the inherited contract from it.

The view modifier

It is is used when a function doesn't mutate or do any event or dangerous thing.
It disallows:

- Modifying state variables.
- Emitting events.
- Creating other contracts.
- Using selfdestruct.
- Sending Ether via calls.
- Calling any function which is not marked view or pure.
- Using low-level calls.
- Using inline assembly containing certain opcodes.

The functions is Solidity

Functions in solidity have the following syntax:

```
function function_name() public view returns (string
memory) {
    …
}

function function_name() internal returns (uint) {
    ….
}
```

The code of the SimpleStorage.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    string public myMessage;

    function set(string memory message) public {
        myMessage = message;
    }

    function get() public view returns (string memory)
{
        return myMessage;
```

Voting.sol

Structs

Definition:

```
struct Book {
    string title;
    string author;
    uint book_id;
}
```

Usage:

```
contract test {
    struct Book {
        string title;
        string author;
        uint book_id;
    }
    Book book;

    function setBook() public {
        book = Book('Learn Java', 'TP', 1);
    }
    function getBookId() public view returns (uint)
        return book.book_id;
    }
}
```

Mappings

They are like `unordered_map` in C++ and `HashMaps` in Java.
They can only get the Storage type.

```
pragma solidity ^0.5.0;

contract LedgerBalance {
    mapping(address => uint) public balances;

    function updateBalance(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}
```

Events

In Solidity, events are a tool for logging and notifying external entities, like user interfaces or other contracts, about specific contract actions or changes. They are defined using the `event` keyword and can emit data when triggered. Events generate log entries on the blockchain, making it possible to track contract activities off-chain and enhancing transparency in decentralized applications.

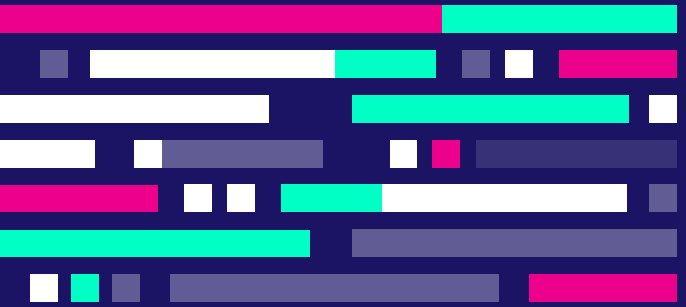
```
event Ajab(string dalilesh);

function ajab(string memory dalil){
    emit Ajab(dalil);
}
```

require

In Solidity, `require` is a control structure used to validate conditions within smart contracts. It checks a given condition, and if it evaluates to `false`, it reverts the transaction, undoing all changes and preventing further execution, with an optional custom error message explaining the reason for the failure. `require` is commonly used for input validation and ensuring that certain preconditions are met before allowing a function to proceed.

```
require(amount > 0, "Amount must be greater than zero");
```



Let's write a Voting contract!

For the sake of completeness.

Importing files

Local import

```
import "./MyLibrary.sol"; // Import a file from the current directory
import "./utils/Helper.sol"; // Import a file from a subdirectory
```

Global import

```
import "openzeppelin/contracts/token/ERC20/ERC20.sol"; // Import a file from a package
```

From a URL

```
import "https://github.com/organization/repo/contracts/MyContract.sol"; // Import from a URL
```

Importing contracts

Import a specific contract

```
import { MyContract1, MyContract2 } from "./MyContracts.sol"; // Import
specific contracts
```

Aliasing Imported Names:

```
import { MyContract as AliasContract } from "./MyContracts.sol"; // Alias
imported contract
```

pure keyword

Pure functions ensure that they not read or modify the state. A function can be declared as pure. The following statements if present in the function are considered reading the state and compiler will throw warning in such cases.

- Reading state variables.
- Accessing address(this).balance or <address>.balance.
- Accessing any of the special variable of block, tx, msg (msg.sig and msg.data can be read).
- Calling any function not marked pure.
- Using inline assembly that contains certain opcodes.
-

Arrays

Like all other languages solidity has got arrays with the following syntax:

```
uint maghadir[10];
uint maghadir[3] = [1, 4, 5];
uint maghadir[] = [3, 2, 5];

// dynamic size:
uint size = 3;
uint maghadir[] = new uint[](size);

// access
uint meghdar = meghdar[2];
```

Enums

Enums in Solidity are data types used to define a fixed set of named values or options, typically representing distinct and predefined states or choices within a smart contract.

```
enum State { Pending, Active, Inactive }

State public currentState; // Declare a state variable

function setState(State newState) public {
    currentState = newState;
}

// Function to get the current state value
function getState() public view returns (uint256) {
    return currentState;
}
```

Arrays

Like all other languages solidity has got arrays with the following syntax:

```
uint maghadir[10];
uint maghadir[3] = [1, 4, 5];
uint maghadir[] = [3, 2, 5];

// dynamic size:
uint size = 3;
uint maghadir[] = new uint[](size);

// access
uint meghdar = meghdar[2];
```

While loop

```
while (expression) {
    Statement(s) to be executed if expression is true
}

// example
uint i = 0;
while (i < 10) {
    // do stuff
    i++;
}
```

For loop

```
for (initialization; test condition; iteration
statement) {
   Statement(s) to be executed if test condition is
true
}

// example
uint i;
for (i=0; i < 10; i++) {
    // do stuff
}
```

continue and break

Continue and break are always there as always.

if and if-else

```
if (expression 1) {
   Statement(s) to be executed if expression 1 is true
} else if (expression 2) {
   Statement(s) to be executed if expression 2 is true
} else if (expression 3) {
   Statement(s) to be executed if expression 3 is true
} else {
   Statement(s) to be executed if no expression is
true
}
```