Solidity- Programming Language

Data Types

Similar to other statically typed languages, Solidity data types specify the information stored in a variable, such as a specific value or a reference address of a memory location.

However, one of the key restrictions in Solidity is that the developer needs to specify the data type of every variable at compile time

Solidity has eight value types: signed/unsigned integers, booleans, fixed point numbers, addresses, byte arrays, literals, enums, and contract and function types.

1. **Signed/Unsigned integers** - Integer data types store whole numbers, with signed integers storing both positive and negative values and unsigned integers storing non-negative values. Mean Esse Variable hoo jaate hai ye jssmi negative value or positive value dno store kr paayee hum unko signed integer variable bolthe hai or unsiged variabe esse jo sirf positive value store kre
   Syntx - Signed Variable:- int variable_name
   Syntx - Unsigned Variable:- uint Variable_Name

2. **Booleans** - Boolean data type is declared with the bool keyword, and can hold only two possible constant values, true or false

3. **Fixed-point numbers** - Fixed point numbers represent decimal numbers in Solidity, although they aren't fully supported by the Ethereum virtual machine yet.

4. **Addresses** - The address type is used to store Ethereum wallet or smart contract addresses, typically around 20 bytes. An address type can be suffixed with the keyword "payable", which restricts it to store only wallet addresses and use the transfer and send crypto functions.
   Syntx - Addresses public owner = "0xABL...."

5. **Byte arrays** - Byte arrays, declared with the keyword "bytes", is a fixed-size array used to store a predefined number of bytes up to 32, usually declared along with the keyword (bytes1, bytes2).

Syntx - uint256 or uint8

6. **Literals** - Literals are immutable values such as addresses, rationals and integers, strings, unicode and hexadecimals, which can be stored in a variable.

7. **Enums** - Enums, short for Enumerable, are a user-defined data type, that restrict the value of the variable to a particular set of constants defined within the program.

8. **Contract & Function Types -** Similar to other object oriented languages, contract and function types are used to represent classes and their functions respectively. Contracts contain functions that can modify the contract's state variables.

Solidity reference types, such as, are variables that point to the memory address of stored data.** Unlike value types, reference types do not store any value but serve as a location tracker of the intended data. Reference types are primarily categorized into four different types, as follows:

- **Fixed arrays** - Fixed arrays are arrays with a pre-defined size at runtime, declared during initialization.
- **Dynamic arrays** - Dynamic arrays are used to allocate size dynamically at runtime depending on data the programs requires it to store.
- **Structs** - Structs in Solidity are a complex data type which allow you to create a custom type containing members of other types, usually used to group linked data.
- **Mappings** - Mappings store data in key-value pairs similar to dictionaries in other object-oriented languages, with the key being a value data type, and value being any type.

Mapping:-

Mapping in Solidity acts like a hash table or dictionary in any other language. These are used to store the data in the form of key-value pairs, a key can be any of the built-in data types but reference types are not allowed while the value can be of any type.

Code

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

contract Counter {
    // Mapping
    mapping(uint => string) names;
    mapping(uint => book) public books;

    struct book{
        string title;
        string author;
    }

    constructor() public {
        names[1] = "Adam";
        names[2] = "Bruce";
        names[3] = "Carl";
    }

    function addbook(
        uint _id,
        string memory _title,
        string memory _author
    ) public {
        books[_id] = book(_title,_author);

    }
}
```

Constructor

Constructor is a special function declared using **constructor** keyword. It is an optional funtion and is used to initialize state variables of a contract. Following are the key characteristics of a constructor.

- A contract can have only one constructor.
- A constructor code is executed once when a contract is created and it is used to initialize contract state.
- After a constructor code executed, the final code is deployed to blockchain. This code include public functions and code reachable through public functions. Constructor code or any internal method used only by constructor are not included in final code.
- A constructor can be either public or internal.
- A internal constructor marks the contract as abstract.

- In case, no constructor is defined, a default constructor is present in the contract.

Syntax:-

```
contract Smart {
    int data;
    constructor public {
          data = 10;
    }
}
```

**Other Method:-

```
pragma solidity ^0.5.0;

contract Base {
   uint data;
   constructor(uint  _data) public {
      data = _data;
   }
}
contract Derived is Base (5) {
   constructor() public {}
}
```

Enum

In Solidity, *enums* stand for `Enumerable`. **Enums** are user-defined data types that restrict the variable to have only one of the predefined values.

The predefined values present in the enumerated list are called enums. Internally, enums are treated as numbers. Solidity automatically converts the enums to unsigned integers.

An enum should have at least one value in the enumerated list. This value cannot be a number (positive or negative) or a boolean value (true or false).

```
pragma solidity ^0.5.0;

contract Example {

// creating an enum
enum Button { ON, OFF }

// declaring a variable of type enum

Button button;

// function to turn on the button

function buttonOn() public {

// set the value of button to ON

button = Button.ON;

}
// function to turn off the button

function buttonOff() public {

// set the value of button to OFF

button = Button.OFF;

}

// function to get the value of the button

function getbuttonState() public view returns(Button) {

// return the value of button

return button;

}

}
```

Explanation

- Line 3: We create a contract named `Example`.
- Line 5: We create an enum named `Button`.
- Line 8: We declare a variable `button` of the `enum` type.
- Line 11: We create a function `buttonOn()` to turn on the button.
- Line 16: We create another function, `buttonOff()` to turn off the button.
- Line 21: We create a third button, `getButtonState()` to get the value of the button.

Output

- When we call the `buttonOn()` function, the value of the button will be set to `Button.ON`.
- When we call the `buttonOff()` function, the value of the button will be set to `Button.OFF`.
- When we call the `getButtonState()` function, the value of the button will be returned.

Difference Between msg.sender or msg.transfer keyword or function?

In Solidity, `msg.sender` is a pre-defined global variable that represents the address of the account (or contract) that called the current function.

For example, if a contract A calls a function in contract B, then `msg.sender` inside that function in contract B will be the address of contract A. If a user directly calls a function in contract B, then `msg.sender` inside that function will be the address of the user's account.

`msg.transfer` is not a valid function or property in Solidity. However, there is a similar function called `address.transfer(uint256 amount)` which is used to send Ether from one address to another in Solidity.

The `address.transfer` function is a built-in Solidity function that is used to transfer Ether from one address to another. It takes an input parameter `amount` which specifies the amount of Ether to be transferred in Wei. It has a gas stipend of 2300 gas which is enough for a basic transfer to succeed.

```
address payable receiver = 0x1234567890123456789012345678901234567890; //
the address of the receiver
uint256 amount = 1 ether; // the amount of Ether to transfer

// Transfer Ether to the receiver address
receiver.transfer(amount);
```

**Payable

In Solidity, the `payable` modifier is used to indicate that a function can receive Ether (i.e., cryptocurrency) along with its function call. This modifier is used in the function declaration to specify that the function can accept Ether.

Functions marked as `payable` can receive Ether from two sources:

1. From an external account or contract calling the function and sending Ether along with it.
2. From another function within the same contract that transfers Ether to the `payable` function.

Here's an example of a function that is marked as `payable`:

```
function receiveFunds() payable public {
  // do something with the received Ether
}
```

In this example, the `receiveFunds` function is marked as `payable` and can receive Ether sent along with its function call. The `payable` modifier must come before the function's visibility modifier (i.e., `public`, `private`, `internal`, or `external`).

It's important to note that functions marked as `payable` must also have implementation logic to handle the received Ether, such as storing the received Ether in the contract or performing some computation based on the received Ether. If a `payable` function does not handle the received Ether properly, it may result in loss of funds or unexpected behavior.

Modifier

In Solidity, a modifier is a piece of code that can be reused to add additional functionality to a function. A modifier is defined using the `modifier` keyword and can be applied to a function using the `modifier` keyword before the function definition.

Here's an example of a modifier that checks whether the caller of a function is the owner of a contract:

```
contract MyContract {
  address public owner;

  modifier onlyOwner {
    require(msg.sender == owner, "Only the owner can call this function");
    _; // This is a placeholder for the function body
  }

  function myFunction() public onlyOwner {
    // This function can only be called by the contract owner
  }
}
```

In this example, the `onlyOwner` modifier checks whether the caller of the function is the owner of the contract. If the caller is not the owner, the function will revert with an error message. If the caller is the owner, the function will execute normally.

The `onlyOwner` modifier is applied to the `myFunction` function using the `modifier` keyword. This means that only the owner of the contract can call the `myFunction` function.

Modifiers are useful for adding functionality to multiple functions without duplicating code. For example, if you have several functions that can only be called by the owner of a contract, you can use the `onlyOwner` modifier to enforce this requirement in all of the functions.

Require Function

In Solidity, `require` is a built-in function that is used to validate a condition within a smart contract. It is typically used to ensure that certain conditions are met before allowing a function to be executed. If the condition specified in the `require` statement is not met, the function execution is immediately terminated, and all changes made to the contract's state are reverted.

Syntax :-

require(condition, error message);

Here, `condition` is the condition that must be met for the function to continue execution, and `error message` is the message that is returned if the condition is not met.

Here's an example of using `require` to validate that an input parameter is greater than zero:

```
function myFunction(uint256 num) public {
  require(num > 0, "num must be greater than zero");
  // function body
  }
```

In this example, if the `num` parameter is not greater than zero, the `require` statement will cause the function execution to stop, and an error message will be returned. This helps to prevent invalid inputs and protects the contract's state from being altered by invalid data.

The `require` function is also used to enforce various conditions, such as ensuring that a certain address has a certain balance, a certain condition is met before allowing a state change, or that an input parameter meets certain requirements. It is an important tool for writing secure and robust smart contracts in Solidity

-----------------------------------Basic End---