What Is Solidity?

- It is a high-level programming language designed for implementing smart contracts.
- It is a statically typed object-oriented(contract-oriented) language.
- This type of language is widely used in creating smart contracts features in blockchain platforms.
- It is highly influenced by Python, c++, and JavaScript which run on the Ethereum Virtual Machine(EVM).

Advantage

- Object-oriented language that classes and inheritance.
- High-level language, and human-readable code.
- Large community support.
- A lot of developer tools are available eg. Remix, Truffle Suite, Hardhat, Brownie, Etherlime, Solhint etc.
- Static type language, means every data contains a type before storing it.
- Everything is contracted oriented.

Disadvantage

- Once Contract is created, adding features to the existing contract is not possible.
- Solidity is the latest and young, Still, some bugs exist.
- It does not have a floating type.

Solidity Source Code File Extension

File extension tells the type of the file. Smart contracts created with a file name and .sol extension.

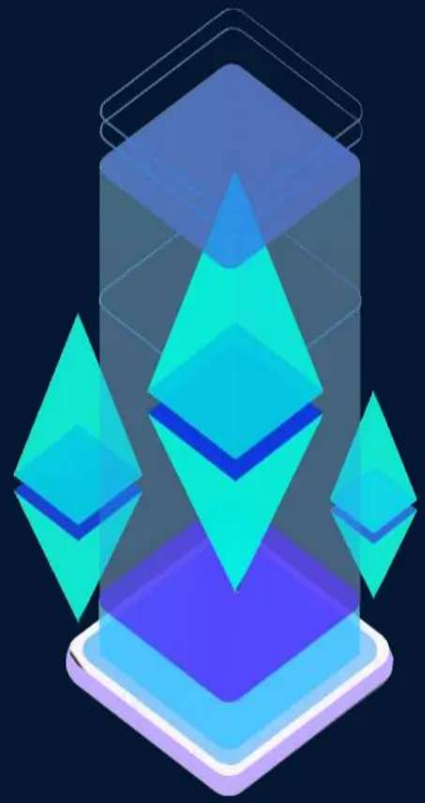
For example, BHU_Computer_Science.sol is a BHU Computer Science Smart Contract Example.

Concepts You Should Know to Understand Solidity

Ethereum

Ethereum is a decentralized, open-source blockchain platform that enables developers to build and deploy smart contracts and decentralized applications (dApps).

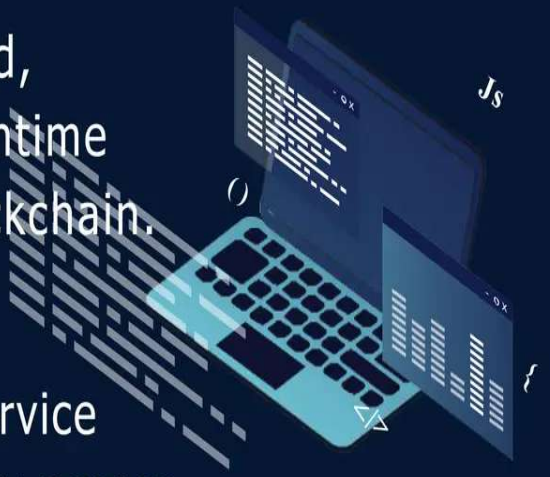
It was proposed by Vitalik Buterin in late 2013 and development began in early 2014, with the network going live on July 30, 2015.



Concepts You Should Know to Understand Solidity

Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a decentralized, Turing-complete virtual machine that serves as the runtime environment for smart contracts on the Ethereum blockchain.



It's also very effective in preventing DOS or Denial-of- Service attacks and confirms that the program does not have any access to each other's state, also ensures that the communication is established without any potential interference.

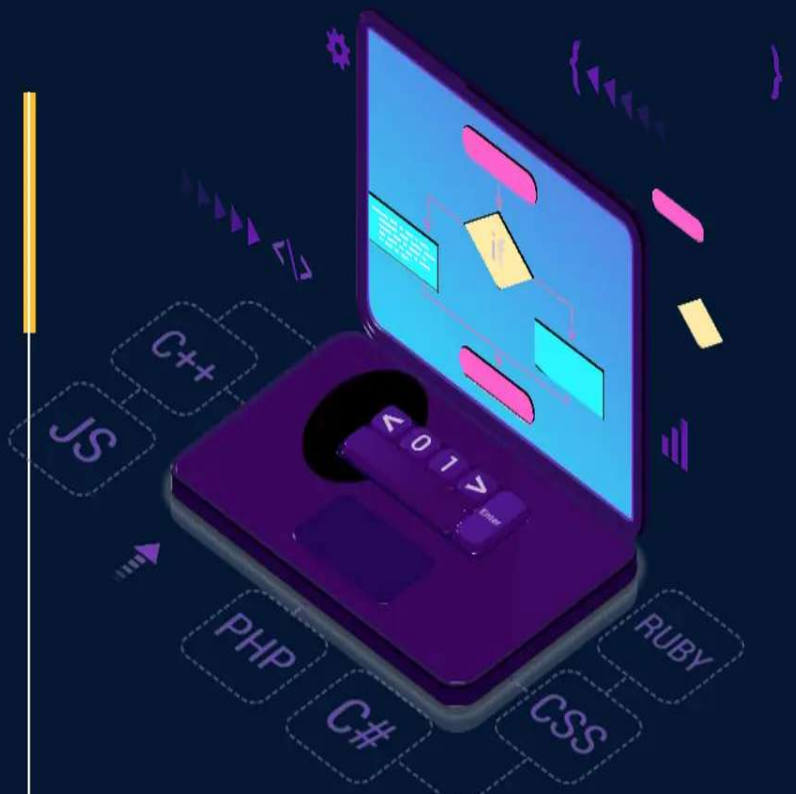
Concepts You Should Know to Understand Solidity

Smart Contracts

Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They run on blockchain networks and automatically execute and enforce the terms of the agreement when certain conditions are met.

Developers use the Solidity programming language to develop smart contracts. Using Solidity, you can program the contracts to do any type of task.

Different Methods of Setting Up Solidity Compiler Environment

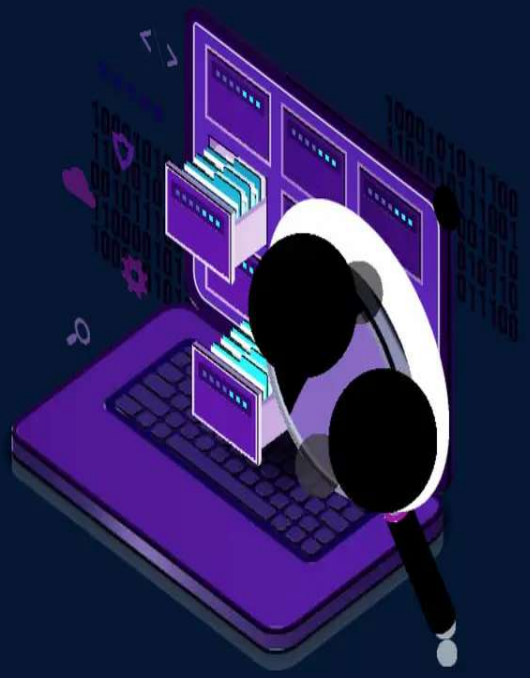


- **Remix:** Remix IDE is an application that provides plugins and a development environment for smart contracts. Users can use this application online without installing any software for the environment.
- Node.js
- Docker Image
- Binary Packages

Reserved Keywords

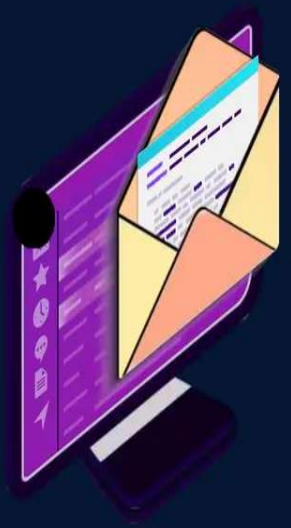
Some of the Reserved keywords are:

- Alias
- Auto
- Unchecked
- Sizeof
- Copyof
- Define
- Override
- Switch
- Etc.



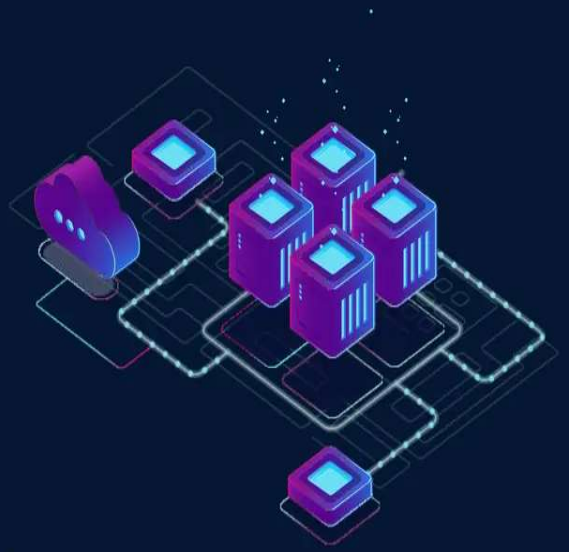
Data Types in Solidity

- **uint**: Unsigned integer types.
- **int**: Signed integer types.
- **bool**: Boolean values variable is true or false.
- **address**: It's used to store the addresses of users or other contracts.
- **enum**: User-defined types for creating a set of named constants.
- **bytes and bytes32**: Eventually all the other types are stored in the form of bytes.
- **strings**: A collection of one or more characters.
- **arrays**: A collection of data of the same type, like uint[] or bool[];
- **mappings**: Key-value stores where data is organized in a mapping from keys to values.
- **structs**: User-defined data structures that can contain a combination of different data types.



Operators in Solidity

- **Arithmetic Operators** e.g. +,-,*, /,%
- **Comparison Operators** e.g. ==, !=, <, >, >=, <=
- **Logical Operators** e.g. &&, ||, !
- **Assignment Operators** e.g. =, +=, -=, *=, &=
- **Bitwise Operators** e.g. &, |, ^, ~, <<, >>
- **Special Operators** e.g. Member Access (.), ++, --, ? : etc.



Solidity Variables: Visibility

Variables can be declared with different visibility modifiers, such as public, private, internal, which determine who can access the variable. Two types of visibility solidity support State variable visibility and Function visibility

- **Public: a**ccessible by everyone including the contract itself.
- **Private:** can only be accessed defined contracts.
- **Internal:** can only be accessed by the defined / derived contract
- **External**: can be called from other contracts and via transactions

*Note: **External** Visibility can used only in Functions*

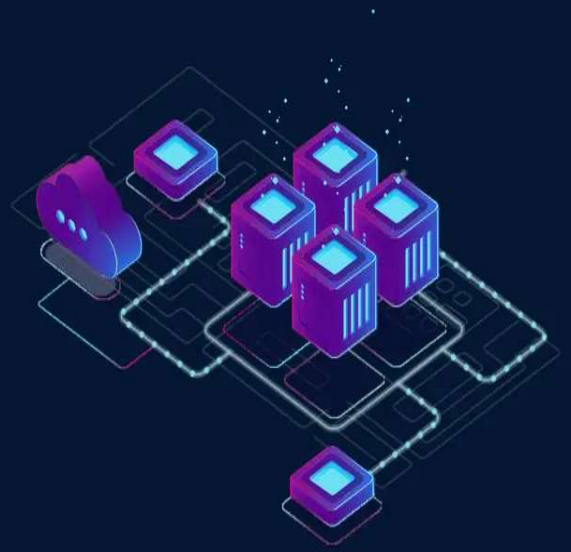
Understanding the Solidity Syntax

```
pragma solidity >=0.4.0 <0.6.0;

Contract Name{

        //variable declaration
        //function
}
```

Compiler Version

Solidity Contract

Codes / functions



REMIX

Understanding the Solidity Syntax

Function name

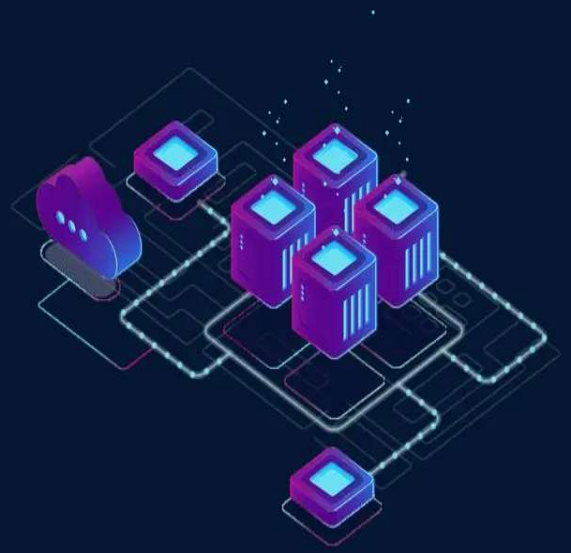
Parameters

Visibility

```
function getLatestPrice(address token) external override view
    returns (int, uint) {

    /*  TODO: implement your functions here
        ...
    */
}
```

Return types

Mutability



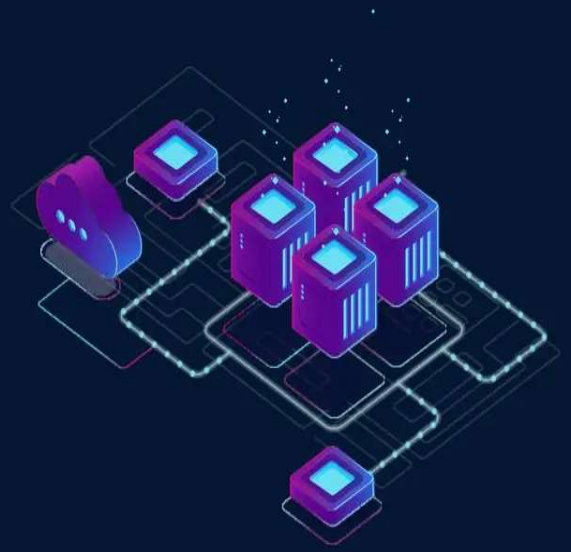
Understanding the Solidity Syntax

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Compiler version

Storage / state

Codes / functions



Understanding the Solidity Syntax

Q1. Write a solidity program for addition of two numbers.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version

contract sumoftwonumbers {
    function sum(uint a, uint b) public pure returns (uint256) {
        uint c = a + b; // Adding the two input numbers
        return c; // Returning the sum
    }
}
```

Understanding the Solidity Syntax

Q2. Write a solidity program to print the string.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;

contract PrintString {    // Function to print the string input provided by the user
   function printString(bytes memory str) public pure returns (string memory) {
      // Convert the bytes array back to a string and return it
      return string(str);
   }
}
```

Understanding the Solidity Syntax

Q3. Write a solidity program for to find the average of three numbers.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;

contract averageofthreenumbers {    // Calculate the average of three numbers
   function average(int256 a, int256 b, int256 c) public pure returns (int256) {
      // Calculate the sum of the three numbers and divide by 3 to get the average
      return (a + b + c) / 3;
   }
}
```

Understanding the Solidity Syntax

Q4. Write a solidity program for concatenation of two strings.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version

contract StringConcatenate {
   function concatenate(string memory s1, string memory s2) public
pure returns (string memory) {
      return string.concat(s1, " ", s2); // Concatenation of String
   }
}
```

Understanding the Solidity Syntax

Q4. Write a solidity program for concatenation of two strings.

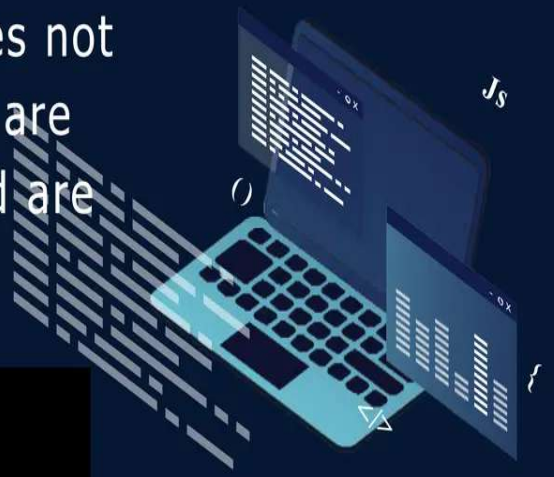
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version

contract StringConcatenate {
   function concatenate(string memory s1, string memory s2) public pure returns (string memory) {
      return string(abi.encodePacked(s1, " ", s2)); // Concatenation of String with space
   }
}
```

Note: abi.encodePacked is a function provided by the Application Binary Interface (ABI) encoder that packs the provided arguments tightly without padding. It converts both strings into bytes and then packs them. It's often used to concatenate strings, convert data types to bytes, and more.

State Mutability in Solidity

State mutability refers to the ability of a function to modify the state of a contract. The state of a contract refers to its variables, which can be read or modified by its functions.

Pure Functions: A pure function is a function that does not read or modify the state of a contract. These functions are typically used for mathematical or string operations and are executed locally.

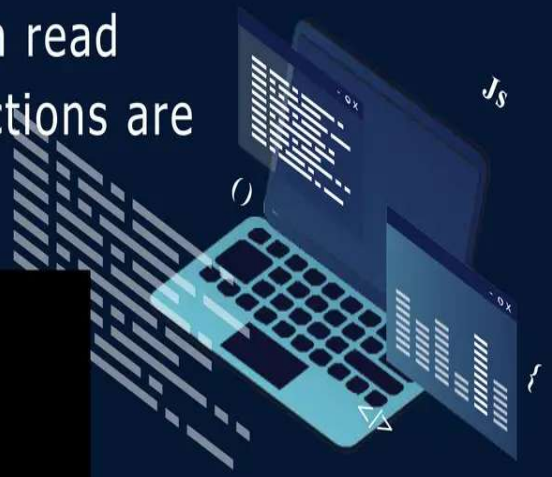



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Calculator {
        function addition(uint256 x, uint256 y) public pure returns (uint256)
{
        return x + y;
        }
}
```

State Mutability in Solidity

State mutability refers to the ability of a function to modify the state of a contract. The state of a contract refers to its variables, which can be read or modified by its functions.

View Functions: A view function is a function that can read the state of a contract but cannot modify it. These functions are typically used to retrieve data from a contract.

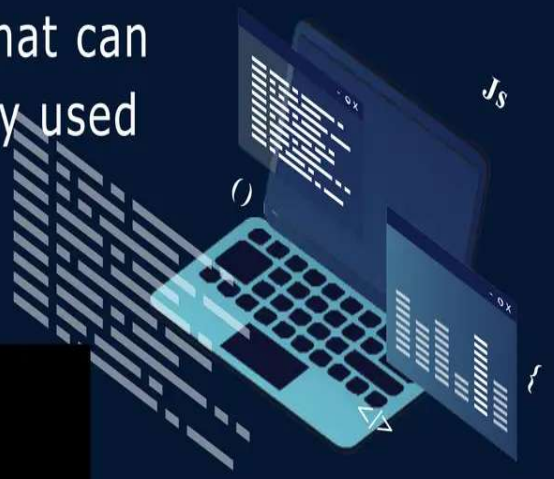



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Calculator {
        uint256 x= 10;
        uint256 y= 20;
        function addition() public view returns (uint256) {
        return x + y;
        }
}
```

State Mutability in Solidity

State mutability refers to the ability of a function to modify the state of a contract. The state of a contract refers to its variables, which can be read or modified by its functions.

Payable Functions: A payable function is a function that can receive Ether in a contract. These functions are typically used for financial transactions.

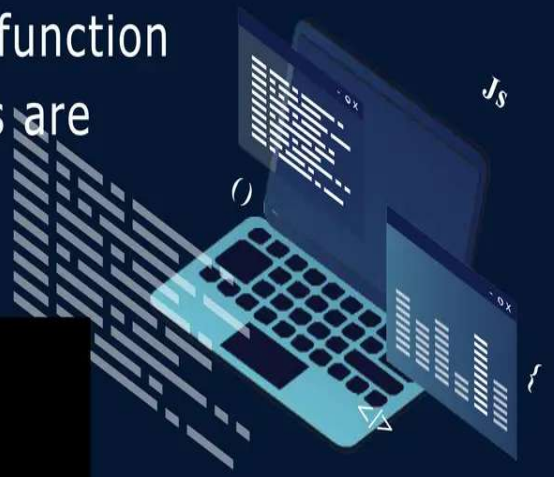



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        function ReceivedPayment () public payable{
                // Code something with the received Ether
```

State Mutability in Solidity

State mutability refers to the ability of a function to modify the state of a contract. The state of a contract refers to its variables, which can be read or modified by its functions.

Non-Payable Functions: A non-payable function is a function that cannot receive Ether in a contract. These functions are typically used to modify the state of a contract.




```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint256 Number= 10;
        function SetNumber (uint256 InputNumber) public {
                Number = InputNumber;
        }
}
```

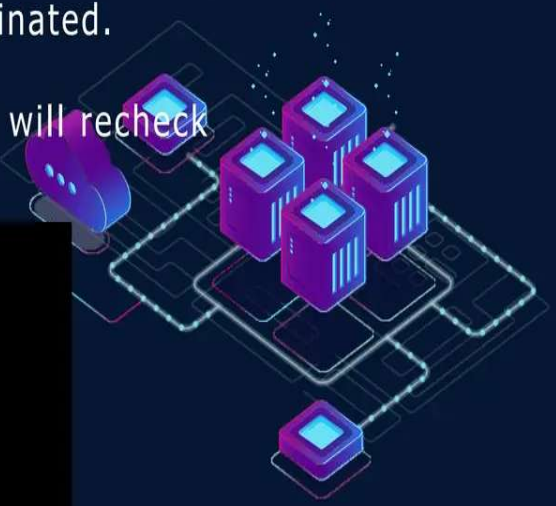
loops in Solidity Syntax

for(initialization; condition; iteration)

Initialization: This initializes the iterator. It is executed only once.

Condition: This checks whether or not the condition is true. If the condition is true, the body will be executed. If the condition is false, the loop will be terminated.

Iteration: This updates the iterator's value. After updating the value, it will recheck the loop condition.



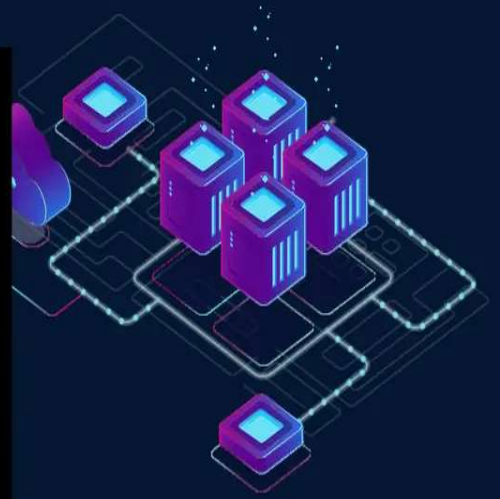
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint i = 0;
        function forloop () public returns (uint) {
                for (uint j = 0; j < 5;  j++) {
                        i++;
                }
          return i;
        }
}
```

loops in Solidity Syntax

while(condition)

Condition: The while loop executes a block of code repeatedly, as long as the specified condition is true. When the condition becomes false, the loop will terminate. If the condition is false at the start of the loop, it won't execute the code.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint i = 0;
        function whileloop () public returns (uint) {
                while (i < 5) {
                        i++;
                }
          return i;
        }
}
```

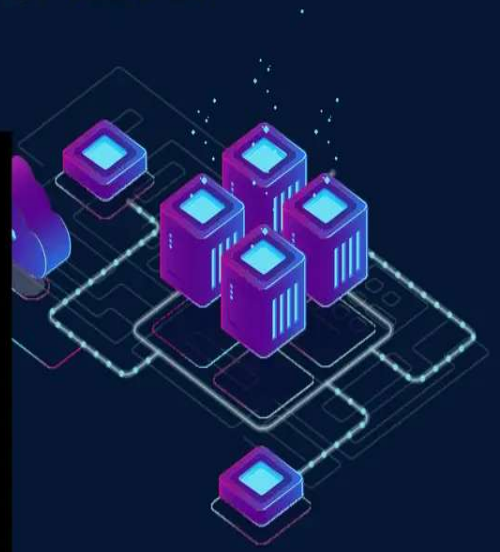


loops in Solidity Syntax

Do - while(condition)

The do-while loop is similar to the while loop with the difference that it checks the condition at the end of the loop. Therefore, it executes a block of code at least once, even if the condition is false.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint i = 0;
        function dowhileloop () public returns (uint) {
                do {
                        i++;
                } while (i < 5);
          return i;
        }
}
```

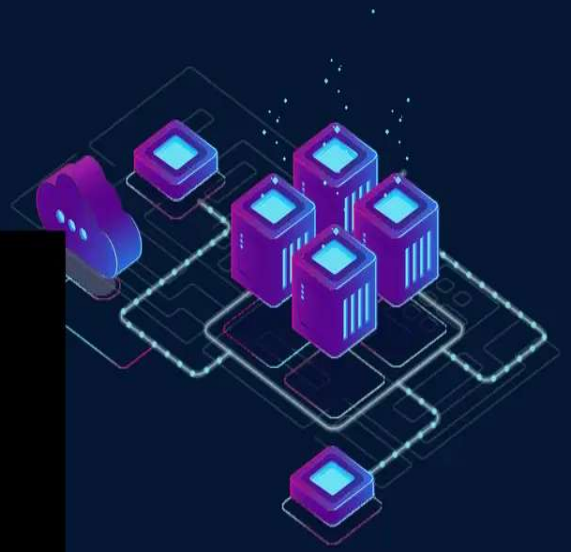


Solidity - Decision Making

if statement: The if statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally.

```
if (expression) {

    Statement(s) to be executed if expression is true

}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint i = 5;
        function ifStatement () public returns (bool) {
                if (i>5) {
                        return true;
        }
}
```

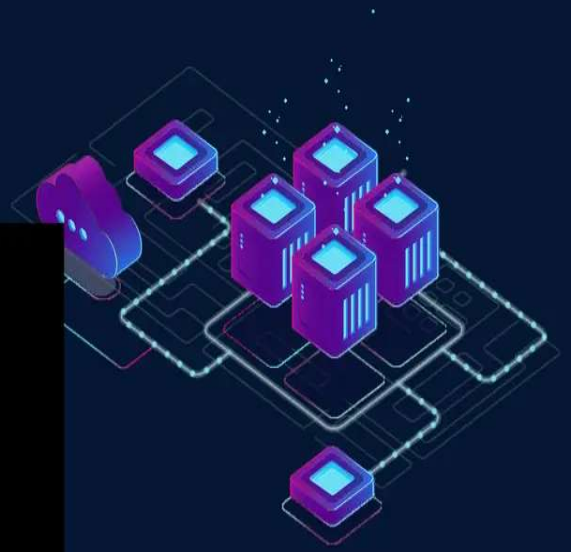


Solidity - Decision Making

if else statement: The 'if...else' statement is the next form of control statement that allows Solidity to execute statements in a more controlled way.

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

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint i = 5;
        function IfElseStatement () public returns (string memory) {
                if (I > 5) {
                        return "i is greater than 5";
                    } else {
                        return "i is less than 5";
                    }
        }
}
```



Solidity - Decision Making

if else if statement: The statement is an advanced form of if else that allows Solidity to make a correct decision out of several conditions.

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

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;  // Version
contract Mycontract {
        uint i = 5;
        function IfElseIfStatement () public returns (string memory) {
                if (i < 2) {
                        return "i is less than 2";
                    } else if (i >= 2 && i  < 5) {
                        return "i is between 2 and 5";
                    } else {
                        return "i is greater than 5";
                    }
        }
}
```