Pakistan Blockchain Institute

PAKISTAN BLOCKCHAIN INSTITUTE

MODULE-2
BLOCKCHAIN AND SMART CONTRACT BASICS

Class-4

Raja Rizwan Saleem
Lead Blockchain Trainer

Edversity.

Introduction to Remix IDE

To create smart contracts on the Ethereum blockchain, developers require an environment that simplifies the development process, provides useful tools and features, and streamlines the testing and debugging of smart contracts. Remix IDE fulfills these requirements and is widely used by developers for Solidity Smart Contract Development.

Introduction to Remix IDE

Remix IDE is a powerful and popular Integrated Development Environment (IDE) for Solidity Smart Contract Development.

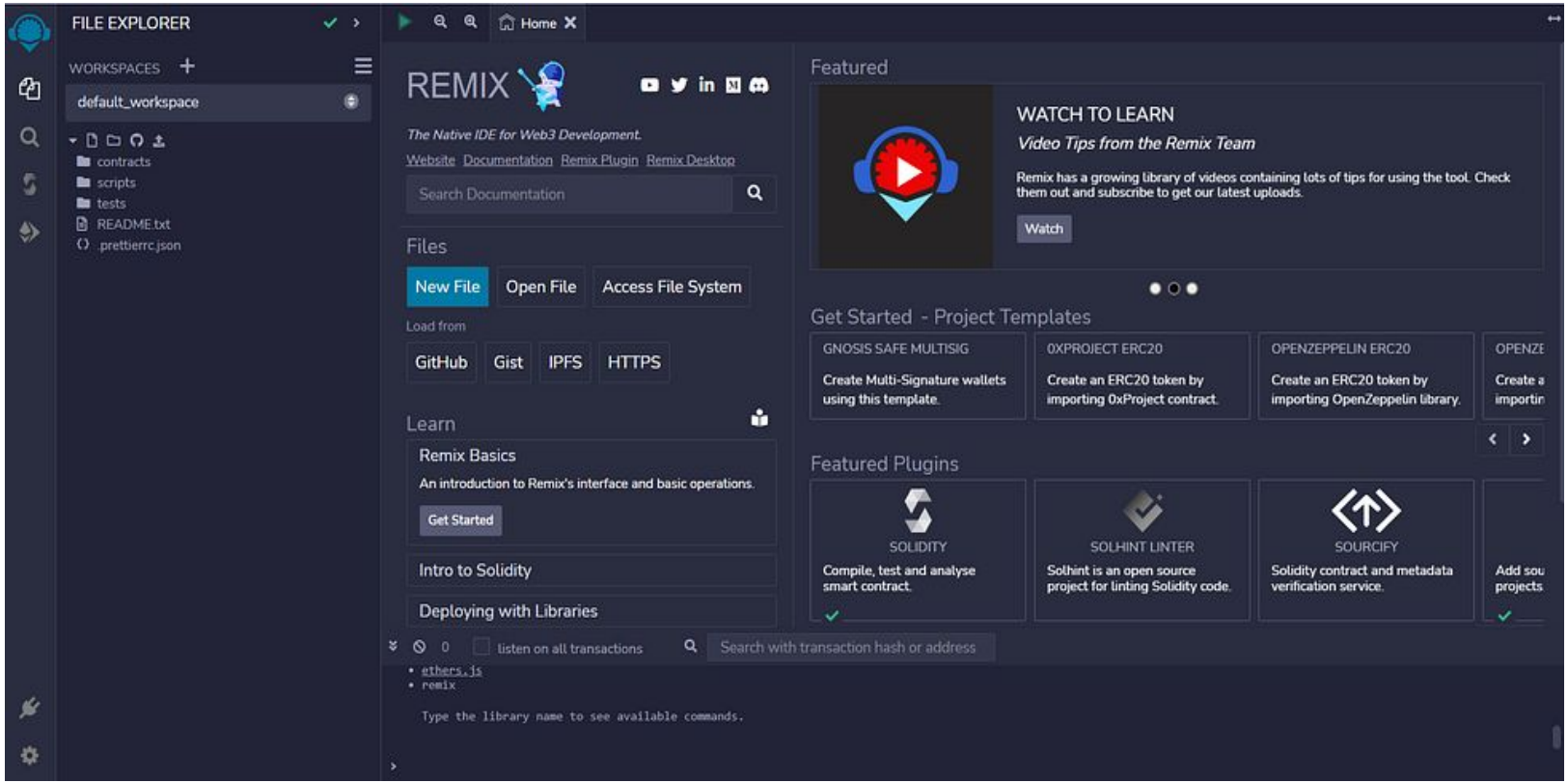
It is a web-based IDE that allows developers to write, test, debug, and deploy smart contracts on the Ethereum blockchain. Remix IDE provides a user-friendly interface that simplifies the Solidity development process, making it accessible to developers of all skill levels.

Access Remix IDE

Step 1: Access Remix IDE

Go to the Remix IDE website at remix.ethereum.org. Remix IDE runs in your web browser, so there's no need to install anything.

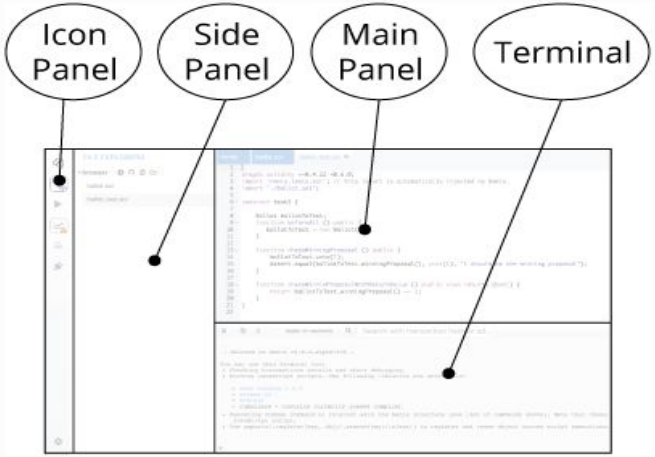
Access Remix IDE



Access Remix IDE

Remix-IDE Layout

The new structure



1. Icon Panel - click to change which plugin appears in the Side Panel
2. Side Panel - Most but not all plugins will have their GUI here.
3. Main Panel - In the old layout this was just for editing files. In the tabs can be plugins or files for the IDE to compile.
4. Terminal - where you will see the results of your interactions with the GUI's. Also you can run scripts here.

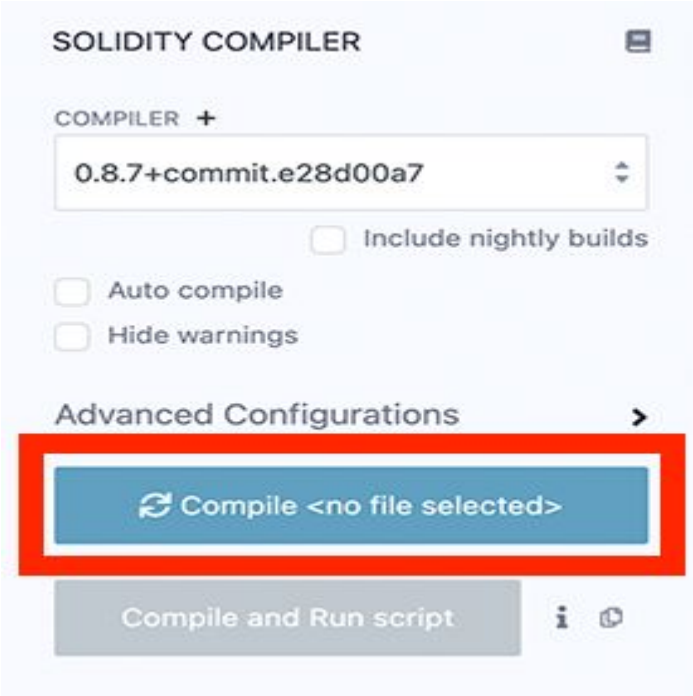
Access Remix IDE

- **File Explorer**: This section displays the files and folders in your project. To create a new file, click on the “**File Explorer**” panel on the left-hand side of the screen. Then, click on the “+” icon to create a new file. You can name the file with the “.sol” extension, which is the file extension used for Solidity smart contracts.
- **The Code Editor pane:** This section displays your new file. The Code Editor includes features like syntax highlighting, auto-completion, and error highlighting to help you write Solidity code more easily.

Access Remix IDE

- **Compiler**: This section displays the Solidity compiler and allows you to compile your code. To compile your code, click on the "**Solidity Compiler**" panel on the left-hand side of the screen. Then, select the version of Solidity you want to use and click on the "Compile" button. If your code contains any errors, the compiler will display them in the "Compilation Details" section.

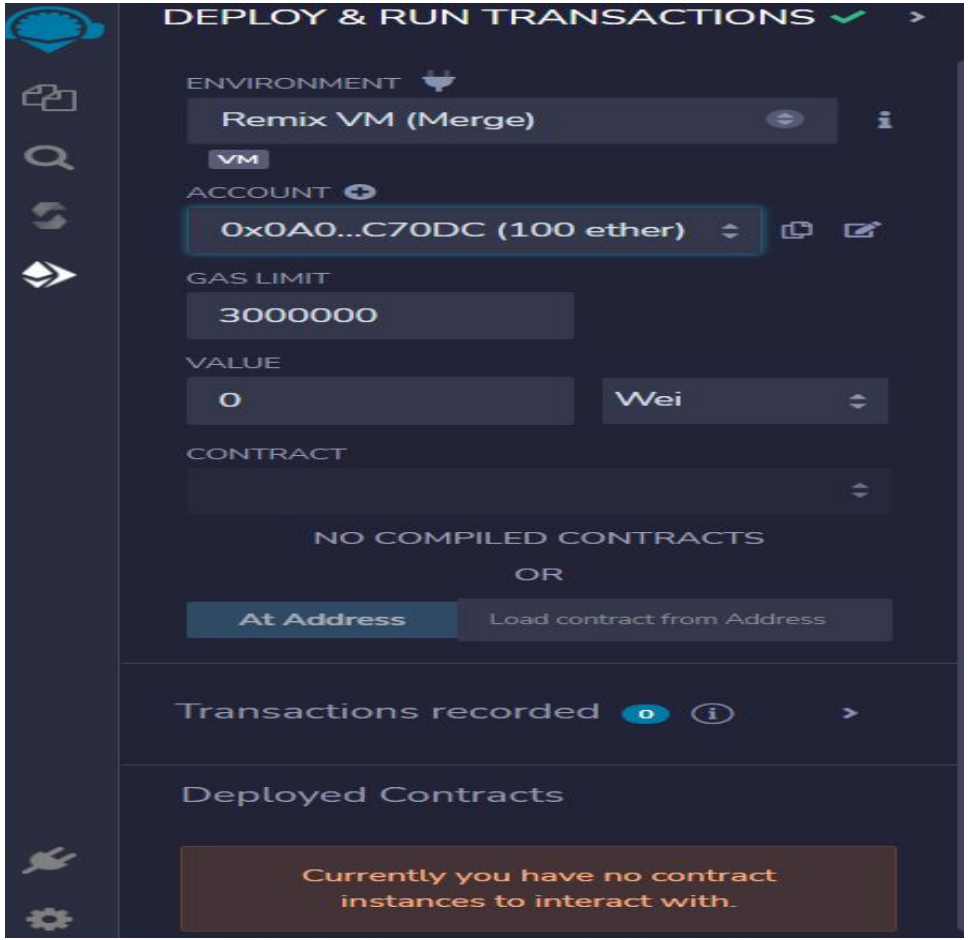
Access Remix IDE



Access Remix IDE

Deploy & Run Transactions: This section allows you to deploy your contract and interact with it on the Ethereum network. To deploy your smart contract, click on the "**Deploy & Run Transactions**" panel on the left-hand side of the screen. Make sure the correct contract is selected, and then click on the "Deploy" button. You'll need to have an Ethereum wallet connected to Remix IDE to deploy your contract.

Access Remix IDE



Access Remix IDE

Step 2: Create a New File

In Remix IDE, click on the “+” icon in the file explorer panel to create a new file. Name the file `SimpleStorage.sol`.

Step 3: Write Your Smart Contract

Copy and paste the following Solidity code into the `SimpleStorage.sol` file:

Access Remix IDE

Step 4: Compile Your Smart Contract

Click on the “Solidity” tab in Remix IDE. Select `SimpleStorage.sol` in the file explorer panel, then click on the "Compile SimpleStorage.sol" button to compile your smart contract.

Step 5: Deploy Your Smart Contract

Switch to the “Deploy & Run Transactions” tab in Remix IDE. Select “Injected Web3” as the environment (assuming you have MetaMask installed and connected to your local Ethereum network).

Access Remix IDE

Step 6: Deploy Your Smart Contract

Click on the “Deploy” button to deploy your smart contract. Confirm the transaction in MetaMask.

Step 7: Interact with Your Smart Contract

Once deployed, you can interact with your smart contract using the provided interface in Remix IDE. Use the “set” function to set a new value and the “get” function to retrieve the stored value.

Simple Smart Contract

```
pragma solidity ^0.8.0;

contract MyContract {
 uint myNumber;

 function setNumber(uint _number) public {
   myNumber = _number;
 }

 function getNumber() public view returns (uint) {
   return myNumber;
 }
}
```

Simple Smart Contract

Remix IDE provides several tools for code editing and management, including:

- Auto-completion: The Code Editor pane includes auto-completion functionality that suggests code completions as you type.
- Syntax highlighting: The Code Editor pane highlights keywords, variables, and other elements of Solidity syntax.
- Error highlighting: The Code Editor pane highlights errors in your code and provides information on how to fix them.
- Code formatting: The Code Editor pane includes options for formatting your code to make it more readable.
- Find and replace: The Code Editor pane includes a find and replace tool that allows you to search for specific code elements and replace them.
- Code snippets: Remix IDE includes a library of Solidity code snippets that you can use as a starting point for your own code.

Solidity compiler

Solidity compiler compiles the source code and generate bytecode and ABI which can be deployed on blockchain

What is ABI

- Application Binary Interface (ABI) as an interface consisting of all external and public function declarations along with their parameters and return types.
- The ABI defines the contract and any caller wanting to invoke any contract function can use the ABI to do so.
- The bytecode is what represents the contract and it is deployed in the Ethereum ecosystem.

What is ABI

- The bytecode is required during deployment and ABI is needed for invoking functions in a contract.
- A new instance of a contract is created using the ABI definition.
- Deploying a contract itself is a transaction. A transaction is created for deploying the contract on Ethereum. The bytecode and ABI are necessary inputs for deploying a contract.

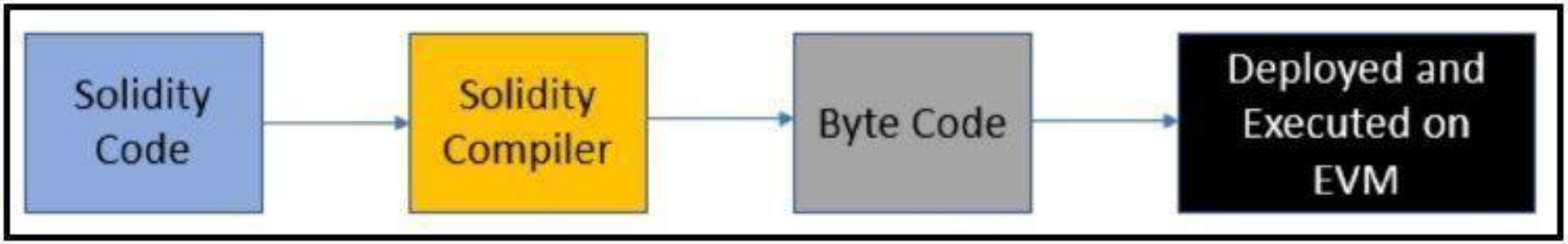
Solidity compiler

1. The code written using Solidity is compiled using a Solidity compiler, which outputs byte code and other artifacts needed for deployment of smart contracts.
2. The Solidity compiler also known as solc can be installed using npm:
    a. npm install -g solc

Ethereum Virtual Machine

1. EVM executes code that is part of smart contracts
2. Smart contracts are written in Solidity
3. EVM does not understand Solidity
4. EVM understands bytecode.
5. Solidity comes with a compiler ‘solc’ which convert Solidity code into bytecode

Ethereum Virtual Machine

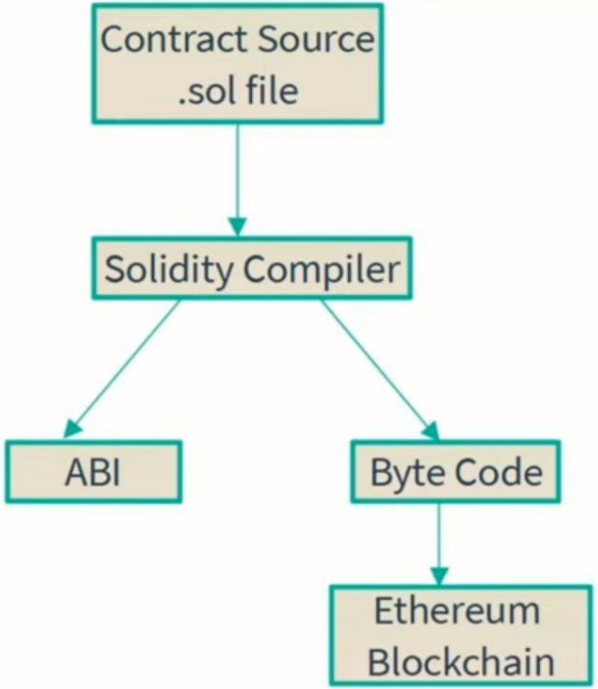


Solidity and Solidity files

1. Solidity is a programming language that is very close to JavaScript
2. Solidity is a statically-typed, case-sensitive, and object-oriented programming (OOP) language
3. The statement terminator in Solidity is the semicolon: ;
4. Solidity code is written in Solidity files that have the extension .sol and it is human readable text file

Smart Contract Compilation

Smart Contract Compilation



Contract Deployment

Contract Deployment

JavaScript Virtual Machine

- Transaction will be executed in a sandbox.
- Own memory blockchain.
- Ideal for testing.

Contract Deployment

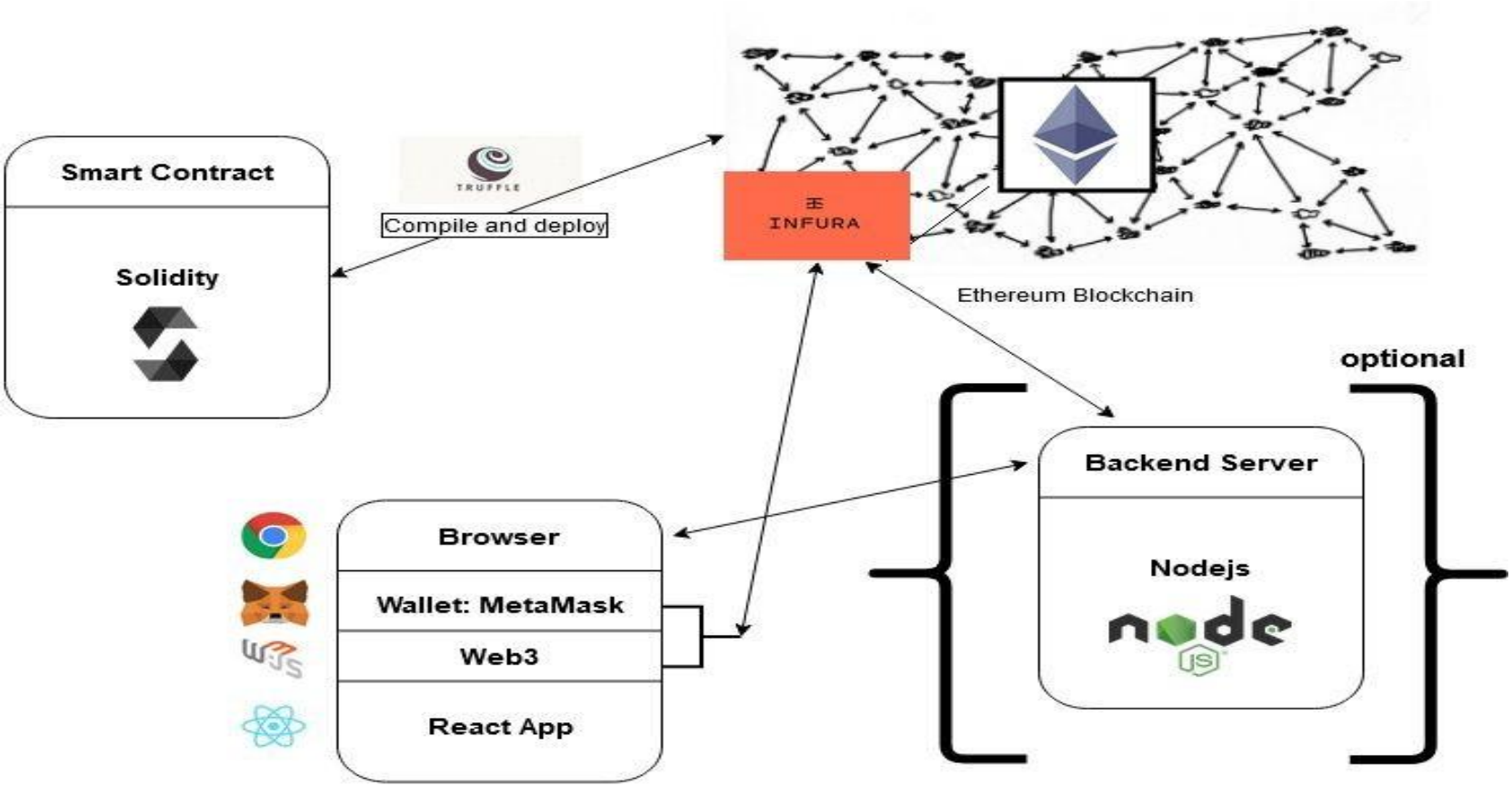
Injected Web3

- Deploy a contract or run a transaction on Ethereum main or test net.

Contract Deployment

Injected Web3

- Deploy a contract or run a transaction on Ethereum main or test net.



Structure of a typical application for Ethereum

SOLIDITY

A Solidity file is composed of four high-level constructs

1. Pragma
2. Comments
3. Import
4. Contracts/library/ interface

What is SPDX license?

All software products have a software license. Think of it this way, most published books have a copyright symbol that either gives you permission to reproduce or not reproduce a book. The same logic appears here. To build trust in a software product, most developers make their source code available on the web. However, the type of license used will determine whether or not the source code can be redistributed, or republished so as to avoid legal issues such as copyright infringement.

Pragma

1. pragma is generally the first line of code within any Solidity file.
2. *pragma* is a directive that specifies the compiler version to be used for current Solidity file.
3. Although it is not mandatory, it is a good practice to declare the pragma directive as the first statement in a Solidity file.

Pragma

1. The syntax for the pragma directive is as follows: `pragma solidity <<version number>>;` `pragma solidity ^0.5.14;`
2. The version number comprises of two numbers, a **major build** and a **minor build** number. 0.**5**.**14**
3. With the help of the pragma directive, you can choose The compiler version and target your code accordingly

pragma

In Solidity, "pragma" is a keyword used to indicate the version of the Solidity compiler that should be used to compile the contract. It sets the compiler version for the contract and helps ensure that the code is compiled using a specific version of the Solidity compiler.

Here's an example of how pragma is used in Solidity:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract MyContract {

    // Contract code goes here

}
```

Contract Syntax

```
Contract name {

// write down code logic here

}
```

Pragma Rules

The ^ character, also known as caret, is optional but plays a significant role in deciding the version number based on 3 rules

Rule 1

The ^ character refers to the latest version within a major version. So, ^0.5.0 refers to the latest version within build number 5, which currently would be 0.5.14.

Rule 2

The ^ character will not target any other major build apart from the one that is provided.

So ^0.5.0 will work for any minor version 0.5.1 or

0.5.11 but will not work for 0.6.0 or 0.4.0

Rule 3

The Solidity file will compile only with a compiler with 5 as the major build. It will not compile with any other major build.

So if compiler is major build 6 it will not work for 5

Pragma

As a good practice, it is better to compile Solidity code with an exact compiler version rather than using ^.

Comments

1. Single-line comments

```
// This is a single-line comment in Solidity
//int public a = 5;
```

2. Multiline comments

```
/* This is a multiline comment
In Solidity. Use this when multiple consecutive
lines Should be commented as a whole */
```

Comments

3. Ethereum Natural Specification (Natspec)
4. Natspec has two formats:
    a. / / for single-line and a
    b. combination of /** for beginning and */ for end of multiline comments.

Comments

```
pragma solidity ^0.5.6;

/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
contract Tree {
    /// @author Mary A. Botanist
    /// @notice Calculate tree age in years, rounded up, for live trees
    /// @dev The Alexandr N. Tetearing algorithm could increase precision
    /// @param rings The number of rings from dendrochronological sample
    /// @return age in years, rounded up for partial years
    function age(uint256 rings) external pure returns (uint256) {
        return rings + 1;
    }
}
```

Solidity file Components


Smart
contract

What are Smart Contracts?

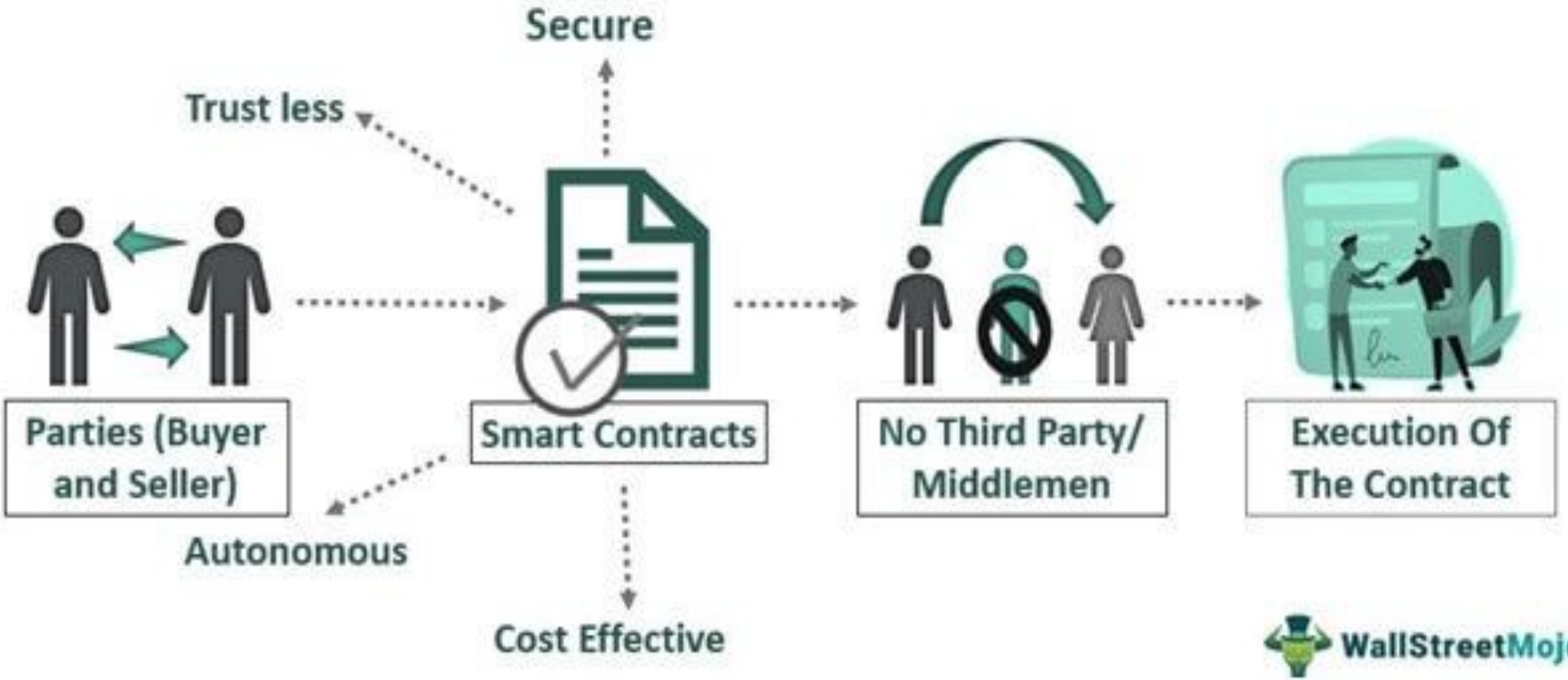
Smart Contracts

['smärt 'kän-ˌtrakts]

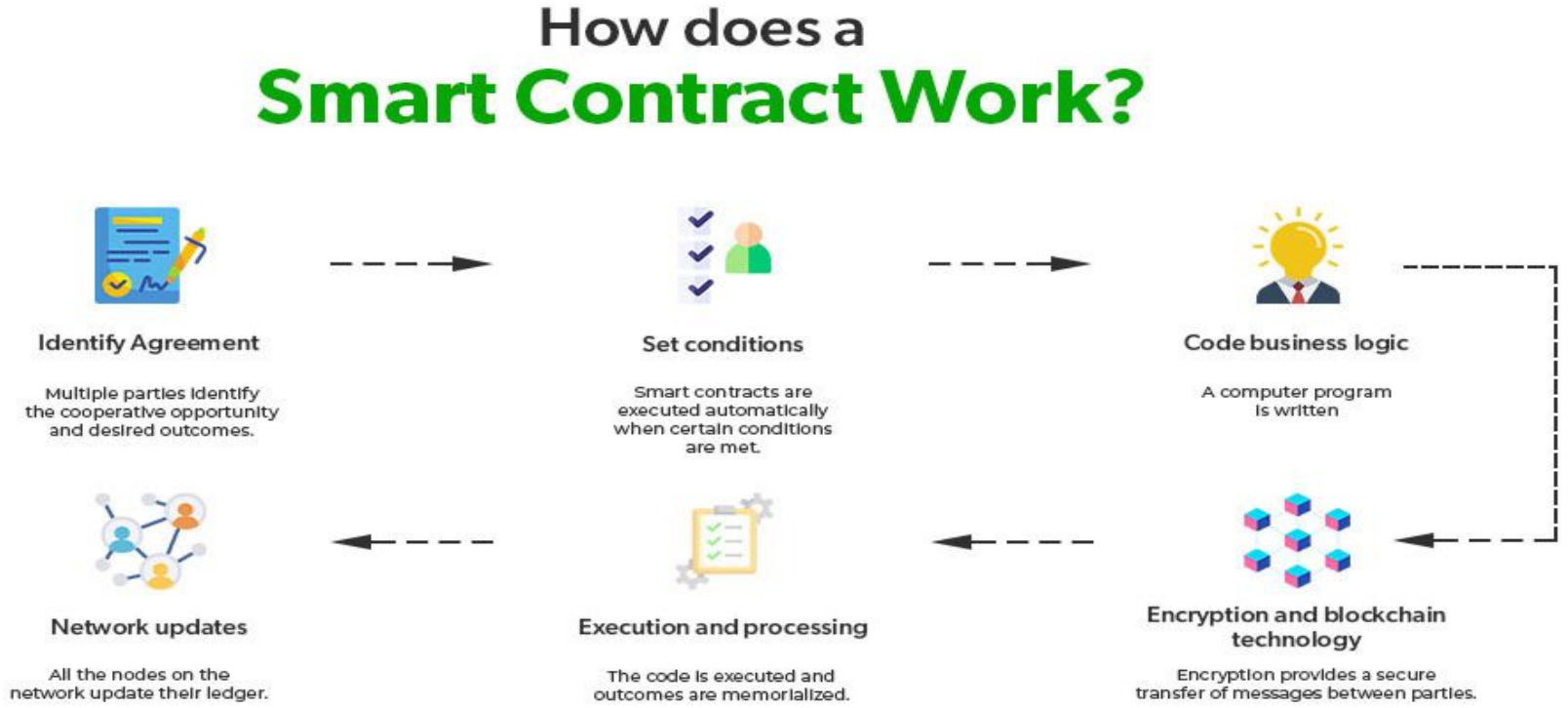
A self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code.

Investopedia

Smart Contracts



Working of a Smart Contract



Working of a Smart Contract

Structure of a Smart contract

A contract consists of the following multiple constructs

1. State variables
2. Function definitions
3. Enumeration definitions
4. Structure definitions
5. Modifier definitions
6. Event declarations

1. State variables

What are variables?

1. Variables in programming refer to storage location that can contain values.
2. These values can be changed during runtime.
3. The variable can be used at multiple places within code and they will all refer to the value stored within it.
4. Solidity provides two types of variable—state and memory variables (memory variables will be discussed later).

State Variables

Variables declared in a contract that are not within any function are called state variables.

State variables in Solidity are permanently stored in a blockchain/Ethereum ledger by miners

State Variables

Solidity supports three types of variables.

- **State Variables** − Variables whose values are permanently stored in a contract storage.
- **Local Variables** − Variables whose values are present till function is executing.
- **Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.

Solidity is a statically typed language, which means that the state or local variable type needs to be specified during declaration. Each declared variable always have a default value based on its type. There is no concept of "undefined" or "null".

State Variables

1. State variables are permanently stored in a blockchain/Ethereum ledger by miners
2. State variables store the current values of the contract.
3. The allocated memory for a state variable is statically assigned and it cannot change (the size of memory allocated) during the lifetime of the contract.
4. Each state variable has a type that must be defined statically

State Variables

Variables whose values are permanently stored in a contract storage.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result; //access the local variable
   }
}
```

Local vaíiablcs

Variables whose values are available only within a function where it is defined. Function parameters are always local to that function.

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return result; //access the local variable
   }}
```

State and Local variables

```
pragma solidity ^0.5.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;
   }
   function getResult() public view returns(uint){
      uint a = 1; // local variable
      uint b = 2;
      uint result = a + b;
      return storedData; //access the state variable
   }
}
```

Visibility Qualifiers in Solidity

State variables Visibility quali fiers

1. internal
2. private
3. public
4. constant

Control Variables Visibility

Visibility modifiers **restrict** who can use values of Solidity variables. Following is a list of modifiers that change these permissions:

`public`: anyone can get the value of a variable. (state variable)

`external`: only external functions can get the value of a local variable. It is not used on state variables.

`internal`: only functions in this contract and related contracts can get values.

`private`: access limited to functions from this contract.

Control Variables Visibility

public: This visibility makes function access directly from outside. They become part of the contracts interface and can be called both internally and externally.

internal: By default, the state variable has internal qualifier if nothing is specified. It means that this function can only be used within the current contract and any contract that inherits from it. These functions cannot be accessed from outside. They are not part of the contracts interface.

private: Private functions can only be used in contracts declaring them. They cannot be used even within derived contracts. They are not part of the contracts interface.

external: This visibility makes function access directly from externally but not internally. These functions become part of the contracts interface.

State variable qualifier -- internal

1. ***Internal***
   a. **By default**, the state variable has the internal qualifier if nothing is specified.
   b. It means that this variable can only be used within current contract functions and any contract that inherits from them.
   c. These variables cannot be accessed from outside for modification; however, they can be viewed.

State variable qualifier -- internal

1. Example

```
int internal age1 = 40;

int age2 = 45;
```

State variable qualifier -- private

1. ***private***
   a. This qualifier is like internal with additional constraints.
   b. Private state variables can only be used in contracts declaring them.
   c. They cannot be used even within derived contracts.
2. Example

```
int private age = 40;
```

State variable qualifier -- public

1. ***public***
   a. This qualifier makes state variables access directly.
   b. The Solidity compiler generates a getter function for each public state variable.
2. Example

```
int public age = 40;
```

State variable qualifier -- constant

1. ***constant***
   a. This qualifier makes state variables immutable.
   b. The value must be assigned to the variable at declaration time itself.
   c. In fact, the compiler will replace references of this variable everywhere in code with the assigned value.
2. Example

```
int constant age = 40;
```

State variable qualifier -- constant

1. ***constant***
   a. Constant can be used with other three qualifiers
2. Example

```
int constant CONSTANT_AGE = 40;// its internal
int internal constant CONSTANT_AGE_2 = 40;
int private constant CONSTANT_AGE_3 = 40;
int public constant CONSTANT_AGE_4 = 40;
int constant private CONSTANT_AGE_5 = 40;
```

State variable qualifier - Demo

```
contract First {

    int internal age = 40;
    int public age2 = 56;
    int private age3 = 60;

    function getAge() public view returns (int){
        return age;
    }

    function getAgeNew() public view returns (int){
        return age3;
    }
}
```

State variable qualifier - Demo

```
contract First {

    int internal age = 40;
    int public age2 = 56;
    int private constant age3 = 60;

    function getAge() public view returns (int){
        return age*2;
    }

    function getAgeNew() public returns (int){
        age3 = 45;
        return age3;
    }
```

State variable qualifier - Demo

```
contract First {

    int internal age;
    int public age2 = 56;
    int private constant age3 = 60;

    function getAge() public view returns (int){
        return age;
    }

    function getAgeNew() public returns (int){
        return age3;
    }

}
```

State variable qualifier - Demo

```
contract First {

    int internal age;
    int public age2 = 56;
    int private constant age3 = 60;

    function updateAge()public {
        age = 12;
    }
    function getAge() public view returns (int){
        return age;
    }

    function getAgeNew() public returns (int){
        return age3;
```

Data Types

1. State variables has an associated data type
2. A data type helps us determine the memory requirements for the variable and ascertain the values that can be stored in them.
3. For example, a state variable of type **uint8** also known as **unsigned integer** is allocated a predetermined memory size and it can contain values ranging from 0 to 255.

Data Types


Data Types in Solidity
01
Mapping
02
Boolean
03
Integer
04
String
05
Address
06
Enum
07
Function
08
Array
09
Struct

Data Types

Declaration and initialization

```
int age = 45;
bool isFeePaid = true;

string name = "Rizwan";
address myaccount = 0xAf8C299e754F7D418e47F6680036ddbf6C888EED;

bytes1 b = "A";

mapping (int => string) list;
```

Thanks

End of Module-2 (Class-4)