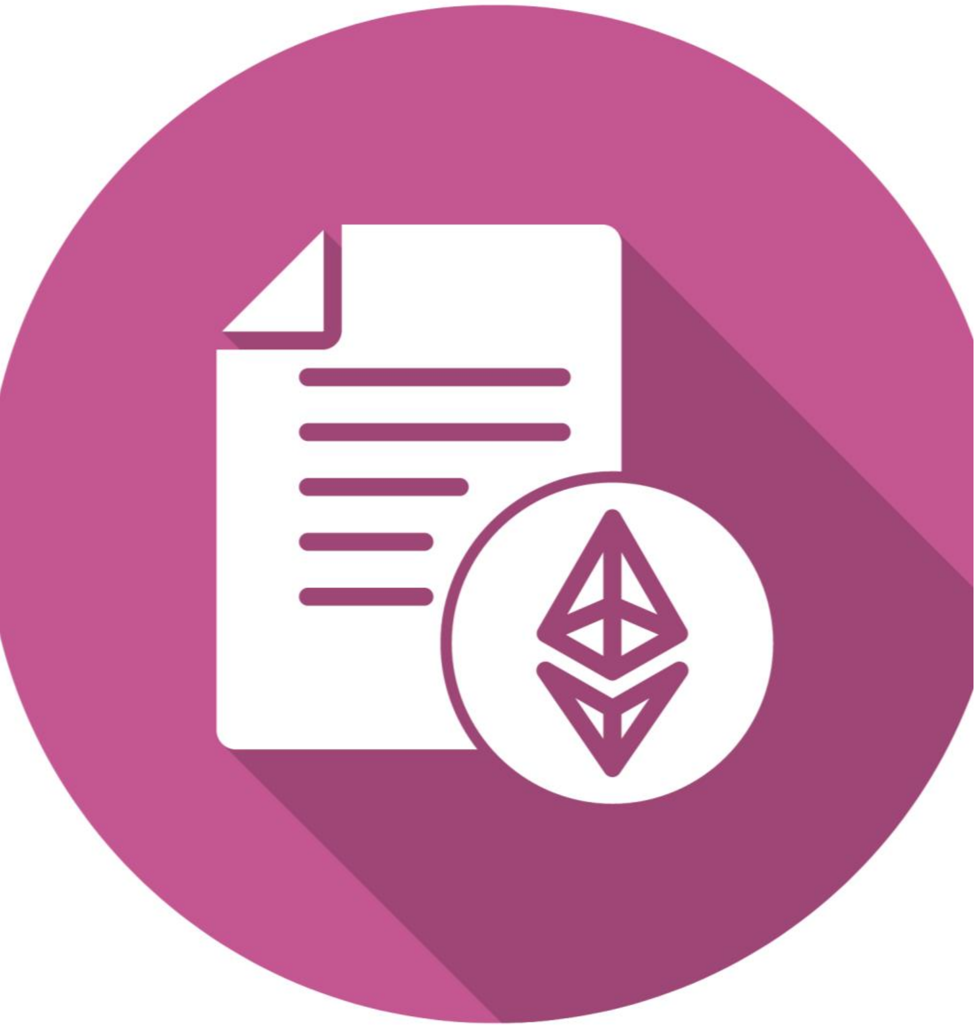
Solidity Basics – 2

Contents

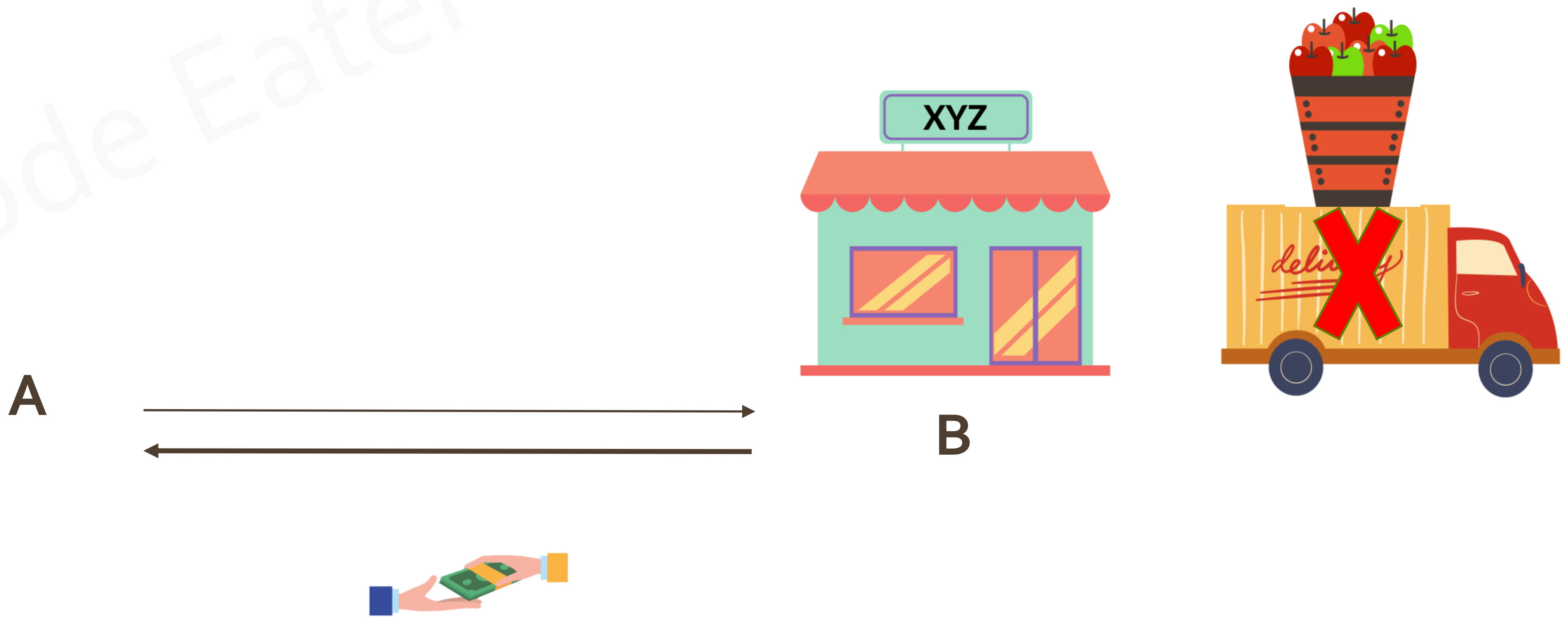


What is a smart contract ?

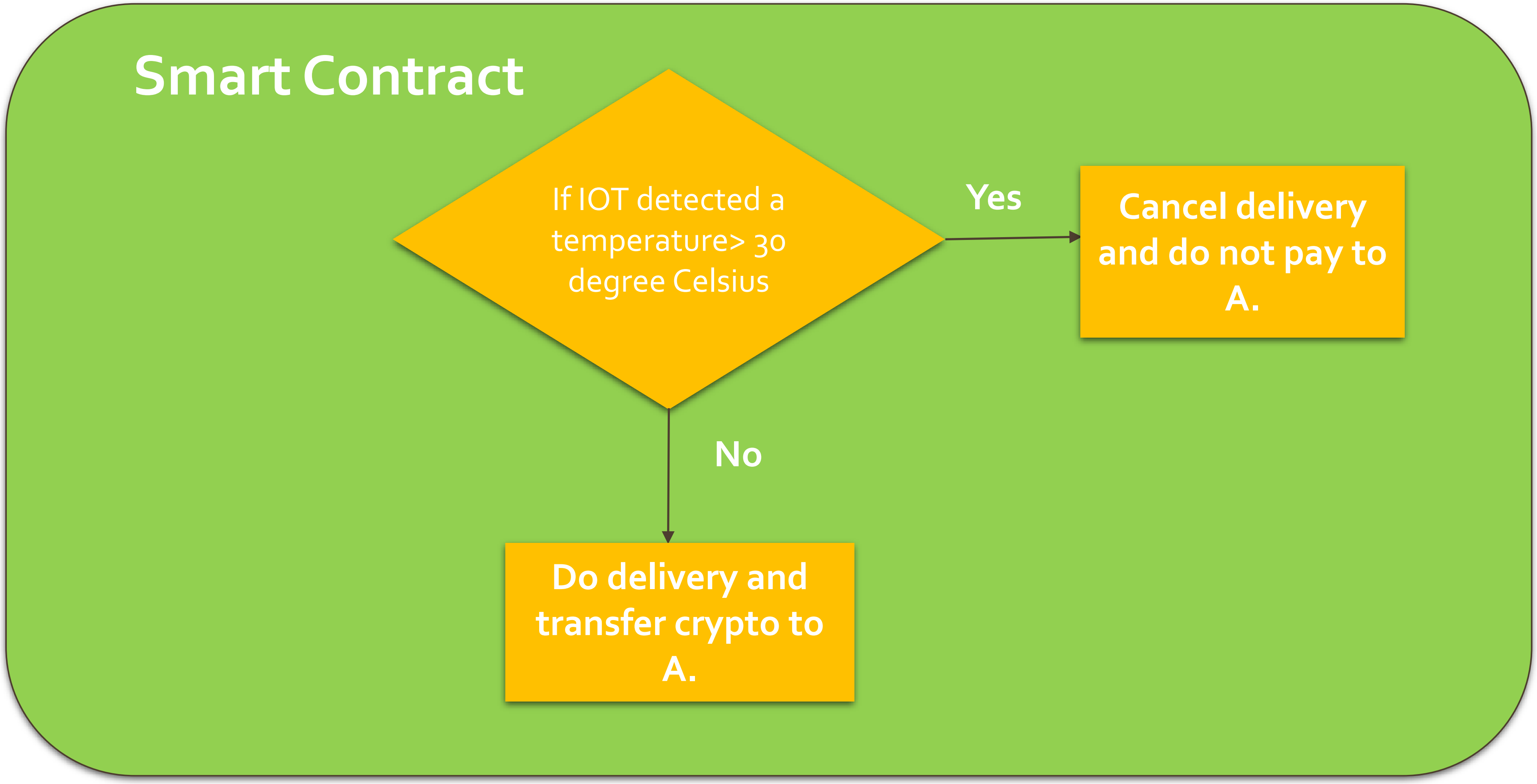
- Smart contracts are **simply programs stored on a blockchain .**

Smart Contract Application

Smart Contract Application



Smart Contract Application


Smart Contract
If IOT detected a temperature> 30 degree Celsius
Yes
Cancel delivery and do not pay to A.
No
Do delivery and transfer crypto to A.

Note-Assuming optimum temperature <30 degree Celsius.

Smart Contract Features

- Smart Contracts are immutable as they get stored on Blockchain.
- Smart contract contracts have their own accounts where it can store cryptocurrency.
- No human intervention is required for cryptocurrency transfer or receiving.

Question Time

What is solidity?

- Solidity is an object-oriented programming language for implementing smart contracts for the ethereum blockchain.
- High-level statically typed programming language.
- Case sensitive.
- With Solidity you can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

Note – You should follow established development best-practices when writing your smart contracts.

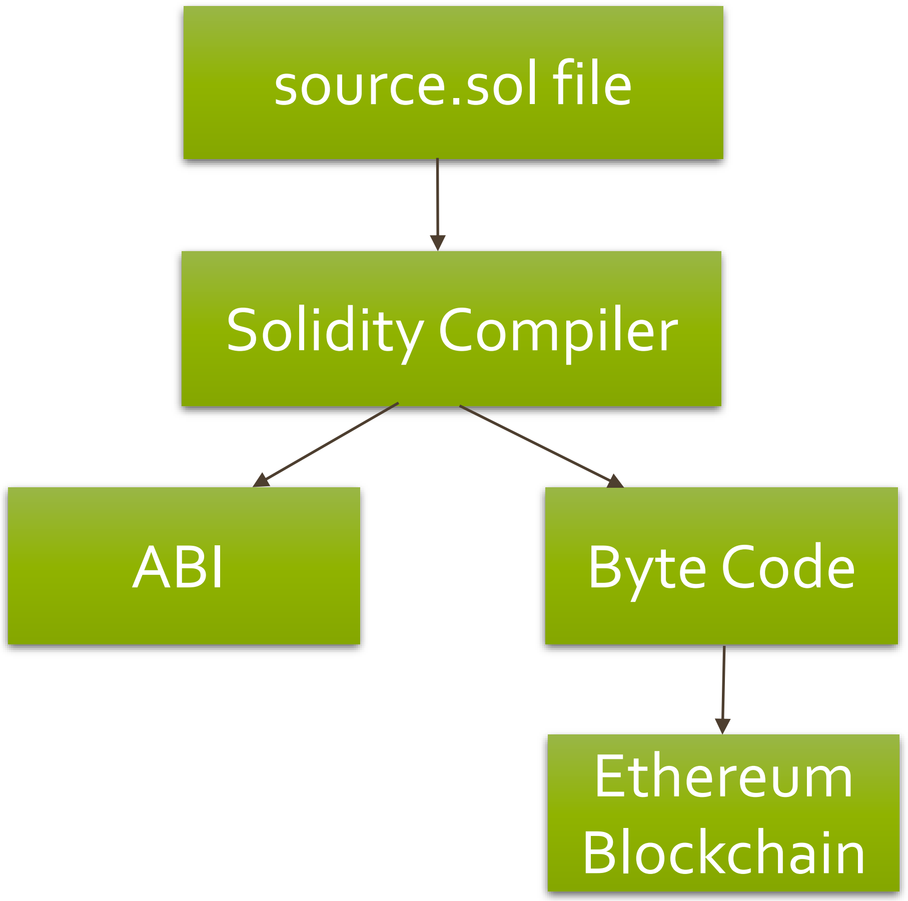
Question Time

Remix IDE

- Sample program with Remix IDE

Question Time

Solidity Compilation Process



Solidity Compilation Process

- Contract bytecode is public in readable form.
- Contract doesn’t have to be public.
- Bytecode is immutable because it is getting stored on Blockchain.
- ABI act as a bridge between applications and smart contract.
- ABI and Bytecode cannot be generated without source code.

Question Time

SPDX

- Trust in smart contracts can be better established if their source code is available. Since making source code available always touches on legal problems with regards to copyright, the Solidity compiler encourages the use of machine-readable SPDX license identifiers. Every source file should start with a comment indicating its license.
- Before publishing, consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file.
- Use "SPDX-License-Identifier: UNLICENSED" for non-open-source code.
- Please see SPDX for more information.

State Variables

- Permanently stored in contract storage.
- Cost gas(expensive) .
- Reading of state variable is free but writing to it is costly.

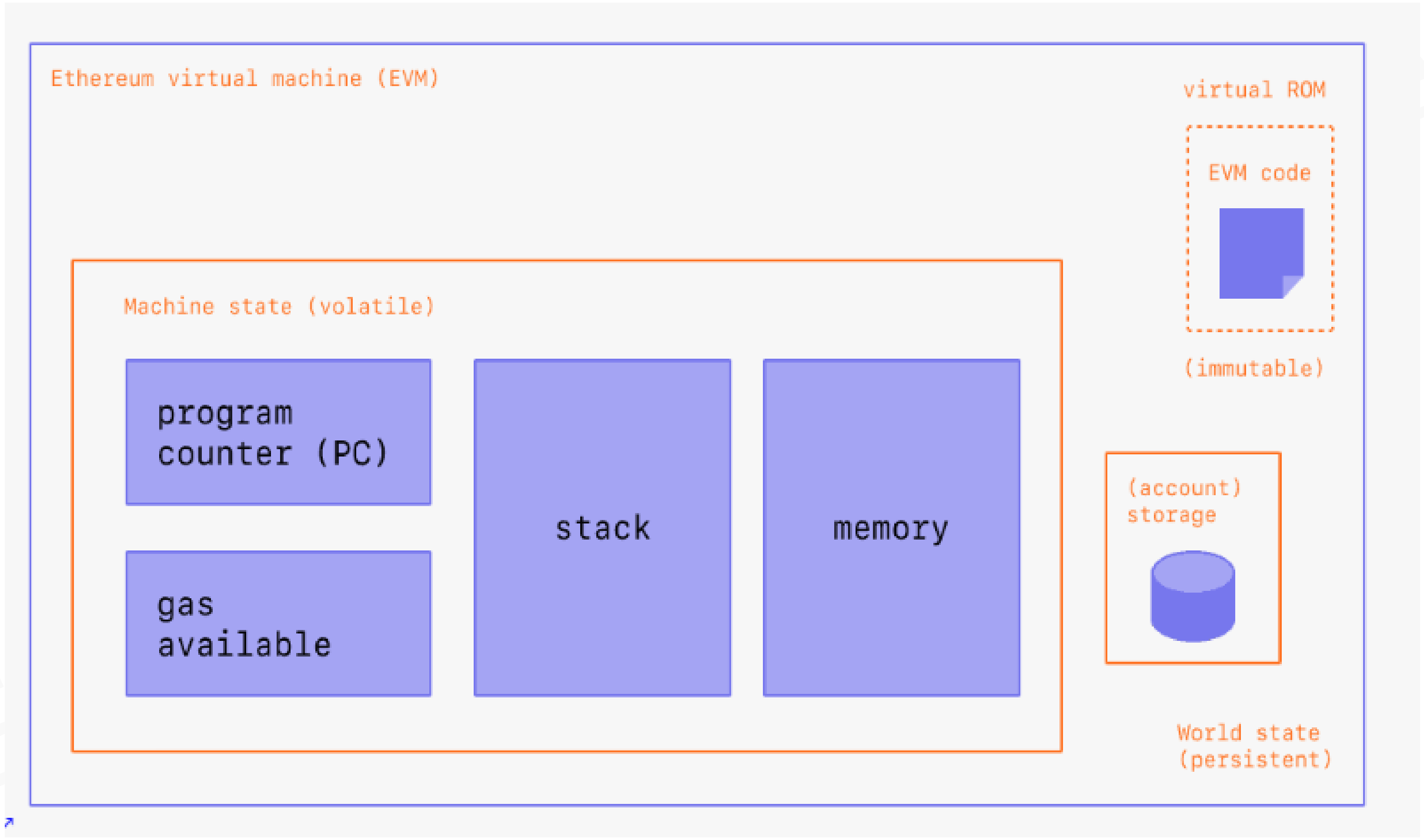
Question Time

Local Variables

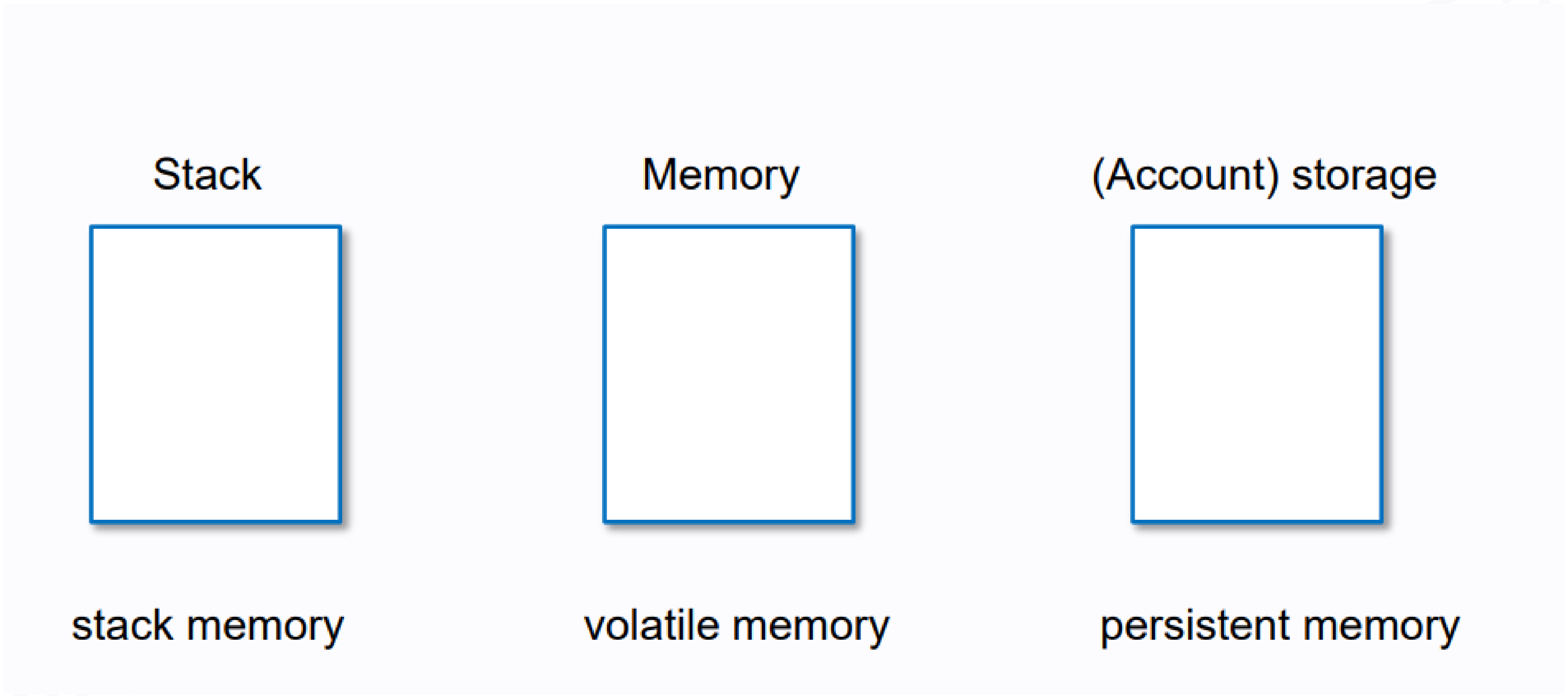
- Declared inside functions and are kept on the stack , not on storage.
- Don’t cost gas.

Question Time

EVM


Ethereum virtual machine (EVM)
virtual ROM
EVM code
(immutable)
Machine state (volatile)
program counter (PC)
gas available
stack
memory
(account) storage
World state (persistent)

Storage Area



Important

- State variables are always in storage.
- Function arguments are stored in either memory or calldata.
- Local variables of basic data type (i.e. neither array, nor struct nor mapping) are stored in the stack.

Question Time

Functions

- When you declare a public state variable a getter function is automatically created.
- For public state variables a getter() function is automatically created.

View Vs Pure

Function Type	State Variable	
	Read	Write
View	✓	✗
Pure	✗	✗

```
// SPDX-License-
Identifier: GPL-3.0

pragma solidity
>=0.7.0 <0.9.0;

contract demo{
  uint public num;


  uint abc;
 function setter()
public { //we are
writing to the state
variable
   uint check=abc;
   num=2;
 }
 function getter()
public view
returns(uint){//read
ing from the state
```

Quiz Time 1 – 10 min

Question Time

Constructor

- Executed only once.
- You can create only one constructor and that is optional.
- A default constructor is created by the compiler if there is no explicitly defined constructor.

Question Time

Basic Data Types

- Integer Data Type
- Bool Data Type
- Address Data Type
- Bytes Data Type

Integer Data Type

int

uint

Signed and Unsigned integers can be of various sizes.

int8 to int256

uint8 to uint256

int alias to int256

uint alias to uint256

By default int and uint are initialized to zero.

Overflow get detected at compile time.

Integer Data Type

Range	
int8 : - 128 to +127	**uint8** : 0 to 255
int16 : - 32768 to +32767	**uint16** : 0 to 65535
-2^(n-1) to 2^(n-1)-1	0 to 2^(n)-1

Question Time

Bool Data Type

- bool public value = true;
- Bool data type value can be either true or false.
- By default value is false if not initialized.

Question Time

Bytes Data Type

- Bytes data type is used to store strings. Range - bytes1, bytes2, ...,bytes32.
- It stores characters.
- bytes1 public arr1="a";
- bytes2 public arr2="ab";
- bytes3 public arr3="abc";
- Everything that will be stored in the bytes array will be in hexadecimal number.
- arr3 will look this

61	62	63
0	1	2

- Click Here – Character To Hexadecimal Table
- Padding of 0 takes place if initialized characters are less than the byte size.

Bytes Data Type

```
contract demo{
    bytes2 public arr1="ab";

    function returnByte() public view returns(bytes1)
    {
        return arr1[0];
    }
}

Output - 0x61
```

Example - 1

Bytes Data Type

```
contract demo{
   bytes2 public arr1="ab";

   function returnByte() public view returns(bytes2)
   {
      return arr1;
   }
}
```

```
Output - 0x6162
```

Example - 2

Question Time

Address Data Type

- address public addr = "0xBE4024Fa7461933F930DD3CEf5D1a01363E9f284"
- The `address` type is a 160-bit value that does not allow any arithmetic operations.

Question Time

Conditionnels

Question Time

Require

Question Time

Modifier

```
contract demo{

    modifier onlytrue {
       require(false==true,"_a is not equal to true");
     _;
    }
    function check1() public pure onlytrue returns(uint){
        return 1;
    }
    function check2() public pure onlytrue returns(uint){
        return 1;
    }
    function check3() public pure onlytrue returns(uint){
        return 1;
    }


   }
```

Question Time

Loop

```
contract demo{

    function check1() public pure{
        for(uint i=0;i<7;i++){

        }

        while(true==true){

        }

        do{

        }while(true==true);
    }

}
```

Question Time

Visibility

	PUBLIC	PRIVATE	INTERNAL	EXTERNAL
Outside World	✓			✓
Within Contract	✓	✓	✓	
Derived Contract	✓		✓	✓
Other Contracts	✓			✓

Quiz Time 2 – 10 min

Question Time

Notes

Smart Contract

A smart contract is a self-executing program that runs on a blockchain and is capable of automatically enforcing the rules and regulations encoded within it. Smart contracts are often referred to as the "building blocks" of decentralized applications (dApps), as they enable developers to create sophisticated decentralized systems that are transparent, secure, and tamper-proof.

In the context of Ethereum, the most widely used blockchain for smart contracts, a smart contract is written in Solidity, a high-level programming language that is specifically designed for creating decentralized applications. Once a smart contract is deployed to the blockchain, it becomes part of the immutable ledger, which means that its code and execution cannot be altered or tampered with.

Smart contracts are capable of automating a wide range of processes and transactions, from simple escrow agreements to complex financial instruments, supply chain management systems, and more. They work by defining the rules and conditions under which a transaction can occur, and then automatically executing the transaction once those conditions are met.

Smart Contract

One of the key benefits of smart contracts is that they eliminate the need for intermediaries, such as banks or other financial institutions, to verify and process transactions. This not only reduces the cost and complexity of transactions, but also enables greater transparency, security, and efficiency in the overall process.

Overall, smart contracts are a powerful tool for creating decentralized, trustless systems that are resistant to censorship and tampering, and can enable new forms of economic and social interaction on a global scale.

Smart Contract Applications

Smart contracts have a wide range of potential applications across various industries and use cases. Here are four examples:

1. Decentralized Finance (DeFi): Smart contracts are at the core of many decentralized finance (DeFi) applications, which aim to provide traditional financial services, such as loans, insurance, and asset trading, without the need for intermediaries. Smart contracts enable DeFi platforms to automatically execute transactions based on predefined rules and conditions, providing users with greater transparency, security, and efficiency in financial transactions.

2. Supply Chain Management: Smart contracts can be used to track and verify the authenticity, origin, and movement of goods across a supply chain. By creating a transparent and tamper-proof record of each transaction, smart contracts can help reduce fraud, improve efficiency, and increase trust between different parties in the supply chain.

Part 1 of 2

Smart Contract Applications

3. Voting Systems: Smart contracts can be used to create transparent and secure voting systems that can prevent fraud and ensure the integrity of the election process. By encoding the rules and conditions of the voting process into the smart contract, the results can be automatically and accurately calculated, without the need for a central authority to verify the results.

4. Real Estate Transactions: Smart contracts can be used to automate the process of buying and selling real estate, by encoding the rules and conditions of the transaction into the contract. This can help reduce the cost and complexity of real estate transactions, while also providing greater transparency and security for buyers and sellers.

These are just a few examples of the many potential applications of smart contracts. As the technology continues to mature, it is likely that we will see even more innovative use cases emerge in a wide range of industries and contexts.

Solidity

Solidity is a high-level programming language used for developing smart contracts on the Ethereum blockchain. It is specifically designed to create decentralized applications (dApps) that can be deployed and executed on the Ethereum Virtual Machine (EVM).

Solidity is an object-oriented language and draws inspiration from C++, Python, and JavaScript. It is statically typed, which means that the data types of variables and functions must be declared explicitly. Solidity also supports inheritance, interfaces, and libraries, which enable developers to create modular and reusable code.

Smart contracts written in Solidity are stored on the Ethereum blockchain, which means that they are publicly accessible and immutable. Once deployed, a smart contract cannot be modified, which ensures that its execution is transparent and tamper-proof.

Solidity is widely used in the Ethereum ecosystem and has become the de facto language for developing smart contracts on the platform. Its popularity is due in part to its developer-friendly syntax and extensive documentation, which make it accessible to developers with varying levels of experience in programming and blockchain development.

EVM

The Ethereum Virtual Machine (EVM) is a runtime environment that enables the execution of smart contracts on the Ethereum blockchain. It is a virtual machine that is completely isolated from the underlying operating system and hardware, which means that smart contracts executed on the EVM are executed in a deterministic and secure environment.

The EVM is responsible for processing and executing transactions on the Ethereum blockchain, including executing smart contracts written in programming languages like Solidity, Vyper, and others. It provides a standardized set of instructions, called opcodes, that define the rules and conditions under which smart contracts can be executed.

One of the key features of the EVM is its ability to execute Turing-complete code, which means that smart contracts written in Ethereum's programming languages can perform complex calculations and operations, similar to those found in traditional software programs.

The EVM is a crucial component of the Ethereum ecosystem and is responsible for ensuring that smart contracts are executed correctly and transparently. Its ability to provide a secure and isolated runtime environment for smart contracts is one of the key reasons why Ethereum has become a popular platform for building decentralized applications (dApps).

Bytecode and ABI

Bytecode and ABI are two important concepts in Ethereum development that are closely related to smart contracts and the Ethereum Virtual Machine (EVM).

Bytecode refers to the low-level machine code that is produced when a smart contract is compiled from its high-level programming language (such as Solidity) into the binary format that can be executed by the EVM. In other words, bytecode is the actual code that is stored on the Ethereum blockchain and executed by the EVM to perform the operations defined by the smart contract.

ABI stands for Application Binary Interface and refers to the interface that defines how to interact with a smart contract, including the function signatures, input and output parameters, and other details about the contract's interface. The ABI is used to create the necessary function calls and parameters that can be used to interact with a deployed smart contract on the Ethereum blockchain.

In practical terms, the bytecode and ABI are used in different ways by developers. The bytecode is generated by the compiler when a smart contract is compiled and is typically deployed to the blockchain using a tool like Remix or Truffle. The ABI, on the other hand, is used by developers to interact with the smart contract from a web interface or application, allowing them to call functions, read data, and perform other operations on the contract.

State Variables

In Solidity, state variables are variables that are permanently stored on the blockchain and hold the state of a smart contract. They can be accessed and modified by functions within the contract and can be considered as the contract's long-term memory.

State variables are declared at the contract level using the `storage` keyword. They can be defined as various types such as `uint`, `bool`, `address`, `string`, `struct`, `mapping`, and more.

When a smart contract is deployed to the Ethereum blockchain, its state variables are initialized with default values, such as zero or an empty string. These values can be changed over time through transactions that call the contract's functions.

It is important to note that state variables can be modified and updated by functions within the same contract, but they cannot be accessed or modified directly by external contracts or accounts. To interact with state variables, external contracts or accounts must call functions in the smart contract that provide access to the state variables.

Overall, state variables are a critical component of Solidity programming, as they allow smart contracts to maintain state and store data on the Ethereum blockchain.

Local Variables

In Solidity, local variables are variables that are defined within the scope of a function or code block, and are only accessible within that scope. They are temporary variables that are created when a function or code block is executed and are destroyed once the execution of the function or code block is complete.

Local variables are declared using the standard syntax for declaring variables in Solidity, specifying the data type and variable name. For example, `uint256 myNumber = 10;` would declare a local variable of type `uint256` with the name `myNumber` and assign it the initial value of 10.

Local variables can be used for a variety of purposes in Solidity, such as performing calculations, storing temporary values, and passing data between functions. They can be used in conjunction with state variables to perform complex operations and manage the state of a smart contract.

It is important to note that local variables are different from state variables in that they are temporary and do not persist between function calls or transactions. They are created and destroyed with each execution of a function or code block, and their values cannot be accessed outside of that scope.

Overall, local variables are a fundamental concept in Solidity programming, providing a way to store and manipulate data within the context of a specific function or code block.

Functions

Functions in Solidity are the basic building blocks for smart contracts. They are similar to functions in other programming languages and allow smart contracts to perform various operations and manipulate data. Functions can be defined at the contract level and can be called from within the same contract or from other contracts.

In Solidity, functions are defined using the `function` keyword, followed by the function name, any parameters that the function accepts, and the function body. For example:

```
function addNumbers(uint256 a, uint256 b) public pure returns (uint256) {
   return a + b;
}
```

In this example, we define a function called `addNumbers` that accepts two parameters of type `uint256` and returns their sum. The `public` keyword indicates that the function can be called from outside the contract, and the `pure` keyword specifies that the function does not modify the state of the contract.

Functions

Functions in Solidity can also have different access modifiers, such as `public`, `private`, `internal`, and `external`, which determine who can call the function and from where. In addition, functions can have return values, multiple return values, and can modify the state of the contract by updating state variables.

Overall, functions are a crucial component of Solidity programming, providing a way for smart contracts to perform various tasks and interact with the Ethereum blockchain.

Constructor

In Solidity, a constructor is a special function that is automatically executed when a contract is deployed to the Ethereum network. The constructor function is used to initialize the state variables of the contract and perform any other setup operations that may be required.

To define a constructor in Solidity, you use the `constructor` keyword followed by any parameters that the constructor accepts and the constructor body. For example:

```
contract MyContract {
    uint256 public myNumber;

    constructor(uint256 _initialValue) {
        myNumber = _initialValue;
    }
}
```

Constructor

In this example, we define a constructor function that accepts a single parameter of type `uint256` and assigns it to the `myNumber` state variable.

It's important to note that a contract can have only one constructor, and the constructor function must have the same name as the contract. Additionally, if a contract does not have a constructor defined, a default constructor with no parameters is provided.

Overall, the constructor function is an essential part of Solidity programming, allowing developers to initialize the state of a contract and perform any other necessary setup operations when deploying a contract to the Ethereum network.

View Vs Pure

In Solidity, `view` and `pure` are two different keywords that can be used to specify the state mutability of a function.

A `view` function is a read-only function that does not modify the state of the contract. It can access the contract's state variables and read data from them, but it cannot modify them. The `view` keyword is used to indicate that a function is a read-only function. For example:

```
function getMyNumber() public view returns (uint256) {
   return myNumber;
}
```

In this example, the `getMyNumber` function is marked as `view` because it does not modify the state of the contract. It simply returns the value of the `myNumber` state variable.

View Vs Pure

On the other hand, a `pure` function is a function that does not read or modify the state of the contract. It is used for functions that perform calculations or transformations on input parameters, and the output is solely dependent on the input parameters. The `pure` keyword is used to indicate that a function is a pure function. For example:

```
function addNumbers(uint256 a, uint256 b) public pure returns (uint256) {
   return a + b;
}
```

In this example, the `addNumbers` function is marked as `pure` because it only performs a calculation on the input parameters and does not read or modify the state of the contract.

Overall, `view` and `pure` are important keywords in Solidity that help to ensure that functions are correctly marked as read-only or pure, which can improve the security and efficiency of smart contracts.

Require

In Solidity, the `require` statement is used to check a condition and throw an error if the condition is not met. It is a way to add input validation to smart contracts and ensure that they function correctly and securely.

The syntax of the `require` statement is as follows:

```
require(condition, error message);
```

The `condition` is a Boolean expression that is evaluated. If the expression evaluates to `false`, then the transaction will be reverted and the error message will be returned to the sender.

Modifier

In Solidity, a `modifier` is a way to add code that can be executed before or after a function call. Modifiers are typically used to validate inputs, check for authorization, or modify the behavior of a function in some way.

The syntax of a modifier is as follows:

```
modifier modifierName(arguments) {
   // modifier code
   _; // this is a placeholder for the function code
}
```

The underscore (`_`) is used to indicate where the function code should be inserted. When a function with a modifier is called, the modifier code is executed first, and then the function code is executed in place of the `_` placeholder.

Modifier

For example, suppose we have a smart contract with a function that requires the caller to be the owner of the contract:

```
function doSomething() public onlyOwner {
   // function code
}
```

In this example, `onlyOwner` is a modifier that checks if the caller of the function is the owner of the contract. The modifier code might look something like this:

```
modifier onlyOwner() {
   require(msg.sender == owner, "Only the owner can call this function");
   _;
}
```

Modifier

When `doSomething()` is called, the `onlyOwner` modifier is executed first. If the caller is not the owner of the contract, then the function call will be reverted with the error message "Only the owner can call this function". If the caller is the owner, then the function code will be executed in place of the `_` placeholder.

Modifiers can also be used to pass arguments to functions, like this:

```
modifier checkValue(uint amount) {
   require(msg.value >= amount, "Not enough ether sent");
   _;
}
```

Modifier

In this example, `checkValue` is a modifier that checks if the amount of ether sent with the function call is greater than or equal to a specified amount. The amount is passed to the modifier as an argument, like this:

```
function buy(uint amount) public payable checkValue(amount) {
   // function code
}
```

In this example, the `buy` function is only executed if the `checkValue` modifier is successful, meaning that the caller has sent enough ether. If the modifier fails, then the transaction will be reverted with the error message "Not enough ether sent".

Thank You

- Please Like and Subscribe :)
- Instagram - @codeeater21
- LinkedIn - @KshitijWeb3