**CODELANDCS BLOCKCHAIN DEVELOPMENT SYLLABUS**

**WEEK 4**

**DAY 3**

**WRITING SMART CONTRACTS**

Writing smart contracts is an exciting and rapidly growing field that offers developers the opportunity to build cutting-edge decentralized applications that can transform the way we interact with each other and our world. By leveraging the power of blockchain technology, smart contracts offer a range of benefits over traditional software development, including immutability, transparency, and automated enforcement of rules and conditions.

If you're new to the world of smart contract development, the first thing you'll need to do is familiarize yourself with the basics of blockchain technology. Blockchains are distributed databases that store information across a network of nodes, allowing for secure and transparent tracking of transactions and data.

The most popular blockchain network for smart contracts is Ethereum, which uses a programming language called Solidity. Solidity is a high-level programming language that allows developers to write code that can be compiled into bytecode and executed on the Ethereum Virtual Machine (EVM).

When writing a smart contract, you'll need to think carefully about the rules and conditions that will govern the behavior of the contract. This might involve defining **variables**, **functions**, and **events** that will determine how the contract operates.

One important consideration when writing smart contracts is **security**. **Because smart contracts are immutable once deployed**, any errors or vulnerabilities in the code can have serious and potentially irreversible consequences. Developers must take care to thoroughly test their code and implement robust security measures to prevent attacks.

Another **key concept** in smart contract development is **gas**, which is the unit of measurement used to calculate the **cost of executing a contract on the Ethereum network.** Because each transaction on the network requires a certain amount of computational resources, developers must carefully optimize their code to ensure that it is as efficient as possible.

One of the most exciting aspects of smart contract development is the **opportunity to build decentralized applications (dApps)** that can operate without the need for intermediaries such as banks, financial institutions, or other centralized authorities. This can help to reduce costs, increase transparency, and empower individuals to take control of their own data and assets.

Overall, writing smart contracts is a challenging and rewarding endeavor that requires a deep understanding of blockchain technology, programming languages, and security best practices. By leveraging the power of decentralized networks, smart contracts offer the potential to transform the way we interact with each other and our world, and to build a more decentralized, secure, and transparent future for all.

**INTER CONTRACT EXECUTION**

Inter-contract execution is a critical concept in smart contract development that refers to the ability of **smart contracts to communicate and interact with each other on the blockchain.** This functionality is essential for building complex decentralized applications (dApps) **that involve multiple contracts working together** to perform various functions.

The most common way to achieve inter-contract execution is through the **use of contract addresses.** Each contract deployed on the blockchain has a unique address that can be used to interact with that contract. Other contracts can use these addresses to call functions and read data from the target contract.

One important consideration when building inter-contract systems is ensuring that the contracts can **communicate** with each other **securely and efficiently.** This requires careful attention to issues such as contract **permissioning**, **data serialization** and **deserialization**, and **gas optimization.**

One popular approach to implementing inter-contract communication is to use an intermediary contract, sometimes called a **"facilitator"** or **"bridge"** contract. This contract acts as a middleman between two or more contracts, allowing them to communicate with each other in a secure and standardized way. The **intermediary contract can also enforce certain rules or standards** that apply to all contracts that use it, such as data formatting or access control.

Another important **consideration** when building inter-contract systems is the issue of **contract upgradability**. Smart contracts are intended to be immutable once deployed, which means that any changes to a contract's code or functionality require deploying a new contract. This can be problematic when trying to build a system of interdependent contracts, as changing one contract may require changing multiple others.

One approach to addressing this issue is to use **"proxy contracts"**, which act as a front-end to the target contract and can be upgraded without changing the underlying contract code. Proxy contracts can also be used to redirect calls to multiple versions of a contract, allowing for gradual upgrades and testing.

Overall, inter-contract execution is a key concept in smart contract development that enables the creation of complex decentralized applications. By allowing contracts to communicate and interact with each other, developers can build more powerful and flexible systems that are capable of performing a wide range of functions. However, inter-contract communication also introduces additional complexity and requires careful consideration of issues such as security, gas optimization, and contract upgradability.

**INHERITANCE**

Inheritance is an essential feature of object-oriented programming that allows a class to inherit the properties and methods of another class. In Solidity, inheritance is used to define relationships between smart contracts. **Solidity** **allows us to create new smart contracts by inheriting from existing ones.** This feature helps us to reuse code and make it more modular.

Inheritance in Solidity allows a smart contract to inherit properties and methods from a parent contract. This inheritance relationship allows the child contract to reuse code from the parent contract, reducing the amount of code that needs to be written and making the code more modular.

**Syntax for Inheritance:**

The syntax for inheritance in Solidity is similar to other programming languages. We use the **keyword “is”** to specify the parent contract from which we want to inherit.

**Types of Inheritance in Solidity:**

Solidity supports four types of inheritance:

**Single Inheritance:** In this type of inheritance, a child contract can inherit from **only one parent contract.**

**Multiple Inheritance:** In this type of inheritance, a child contract can inherit from multiple parent contracts.

**Hierarchical Inheritance:** In this type of inheritance, a child contract inherits from a parent contract, and the parent contract inherits from another parent contract. This forms a hierarchical structure of inheritance.

**Hybrid Inheritance:** This type of inheritance combines multiple inheritance and hierarchical inheritance.

**Visibility Modifiers and Inheritance:**

Solidity provides visibility modifiers to control the visibility of properties and methods in a contract. When a contract inherits from another contract, the visibility of properties and methods of the parent contract also gets inherited.

The visibility modifiers in Solidity are:

**Public:** can be accessed from any contract or external account.

**Private:** can only be accessed from within the same contract.

**Internal:** can be accessed from within the same contract and any contracts that inherit from it.

**External:** can only be accessed from outside the contract.

When a contract inherits from a parent contract, the child contract can access the public and internal properties and methods of the parent contract. The child contract cannot access the private properties and methods of the parent contract.

**Overriding Functions in Inheritance:**

Solidity allows us to override the functions of a parent contract in the child contract. When a function is overridden, the function in the child contract will be executed instead of the function in the parent contract.

To override a function in Solidity, we **use the override keyword.** If we try to override a function without using the override keyword, Solidity will throw a compilation error.

**Abstract Contracts and Inheritance:**

Solidity also supports abstract contracts, **which are contracts that cannot be instantiated directly.** Abstract contracts are used as base contracts that **provide a common interface for a group of related contracts.**

Abstract contracts can have abstract functions, which are functions that do not have an implementation. Abstract functions are used to define a common interface for a group of related contracts. Any contract that inherits from an abstract contract must implement all the abstract functions defined in the abstract contract.

It is important to note that a contract can inherit from multiple contracts in Solidity using comma-separated lists. However, if two or more inherited contracts define a function with the same name, the inheriting contract must override the function and explicitly call the desired implementation using the **super keyword.**