**CODELANDCS BLOCKCHAIN DEVELOPMENT SYLLABUS**

**WEEK 3**

**DAY 1**

**SMART CONTRACT STRUCTURE**

Solidity is a programming language that is primarily used for developing smart contracts on the Ethereum blockchain. The structure of a Solidity code is similar to that of other programming languages. Here is an overview of the different elements that make up a Solidity code:

**Version pragma:** The first line of a Solidity code is usually a version pragma, which specifies the version of Solidity that the code is written in. This helps ensure that the code is compatible with the version of the Solidity compiler being used.

**Import statements:** Solidity code can include import statements that bring in other Solidity contracts or libraries that are needed for the code to function properly.

**Contract declaration:** A Solidity contract is declared using the contract keyword, followed by the name of the contract. The contract declaration is followed by the contract body, which is enclosed in curly braces.

**State variables:** State variables are variables that are **stored permanently in the contract's storage.** They are declared at the top level of the contract, outside of any function. State variables can be accessed and modified by any function in the contract.

**Functions:** Solidity contracts contain functions, which are blocks of code that can be called by external users or by other functions within the contract. Functions can either be **view functions** (which do not modify the state of the contract) or **non-view functions** (which do modify the state of the contract). Functions are **declared** using the **function keyword,** followed by the name of the function, any parameters that it takes, and the function body enclosed in curly braces.

**Events:** Events are used to log important information about transactions that occur on the Ethereum blockchain. Events are declared using the **event keyword,** followed by the name of the event and any parameters that it takes.

**Modifiers:** Modifiers are used to **add extra functionality** to functions in a contract. They can be used to restrict access to a function, or to add additional checks before a function is executed. Modifiers are declared using the **modifier keyword.**

**Structs and enums:** Solidity also supports the use of structs and enums, which are used to define custom data types.

Overall, the structure of a Solidity code is similar to that of other programming languages, but it also includes elements that are specific to developing smart contracts on the Ethereum blockchain.

**SOLIDITY DATATYPES AND VARIABLES**

Solidity is a programming language that is specifically designed for developing smart contracts and communicating with the Ethereum blockchain. In this lecture, we will be discussing the various data types and structures available in Solidity, which are essential for understanding how to develop efficient and secure smart contracts.

Before kick off, it is worthy to underscore that variables that store data types in solidity can be classified in to three:

* fixed sized types
* user defined types
* variable sized types

**FIXED SIZED TYPES**

Fixed-size variables in Solidity are used to represent data of a specific size. This means that the **amount of memory** used to store the variable is **fixed at compile time**, which makes it more efficient than dynamic variables that can change in size during runtime. Solidity supports several fixed-size variable types, including:

* **bool** - a boolean value that can only have the values true or false.
* **uint8 to uint256** - unsigned integers that range from 8 bits to 256 bits in size.
* **int8 to int256** - signed integers that range from 8 bits to 256 bits in size.
* **address** - a 20-byte value that represents an Ethereum address.
* **bytes1 to bytes32** - a fixed-size array of bytes, where the size can be between 1 and 32 bytes.

**BOOLEANS**:

Boolean is a data type that can have only two values: true and false. It is named after **George Boole, a 19th-century mathematician** who created the branch of algebra that deals with logical reasoning.

In Solidity, Boolean data types are used to represent logical values. They are used in **programming to test conditions** and control the flow of execution of a program. For example, in a **voting application**, a Boolean variable could be used to represent whether a particular candidate has been voted for or not.

In Solidity, Boolean data types are declared using the keyword **"bool"**. Here is an example of **how to declare a Boolean variable**:

As you can see, a Boolean variable can either be true or false. You can also initialize a Boolean variable when you declare it, as shown in the example above.

Now, let's take a look at how we can use Boolean data types in Solidity. One common use case is in **if statements**. If statements are used to test a condition and execute a block of code if the condition is true. **Here is an example:**

In this example, the block of code will only be executed if the Boolean variable "isTrue" is true. If "isTrue" is false, the block of code will be skipped.

Another common use case for Boolean data types is in **while loops**. While loops are used to repeatedly execute a block of code while a condition is true. **Here is an example:**

In this example, the block of code will be executed repeatedly as long as the Boolean variable "isTrue" is true. Once "isTrue" becomes false, the loop will exit.

Boolean data types can also be used in **function parameters** and return values. For example, a function that checks if a number is even could return a Boolean value indicating whether the number is even or not. **Here is an example:**

In this example, the function takes a uint256 parameter "\_number" and returns a Boolean value indicating whether "\_number" is even or not.

Finally, it's worth noting that Boolean data types can also be used in conjunction with other logical operators such as **AND (&&)** and **OR (||)** to test multiple conditions. **Here is an example:**

In conclusion, Boolean data types are a fundamental data type in Solidity that are used to represent logical values. They are used in if statements, while loops, function parameters and return values, and in conjunction with other logical operators to test multiple conditions.

**INTEGERS**

In general, an integer is a whole number that can be either positive, negative, or zero. An **unsigned integer** (uint) is a positive whole number, while a **signed integer** (int) can either be positive, negative, or zero.

In Solidity, there are several types of signed and unsigned integers that you can use in your code. These include:

* int8, int16, int32, int64, int128, and int256, which are signed integers with varying bit lengths (from 8 to 256 bits).
* uint8, uint16, uint32, uint64, uint128, and uint256, which are unsigned integers with varying bit lengths (from 8 to 256 bits).

It's important to note that in Solidity, integers are represented using **two's complement notation.** This means that the most significant bit of a signed integer represents the sign of the number. If the most significant bit is 0, the number is positive. If the most significant bit is 1, the number is negative.

Let's take a look at some examples of how to declare and use signed and unsigned integers in Solidity.

To declare a signed integer variable in Solidity, you use the **"int"** keyword followed by the number of bits you want to allocate for the variable. For example, to declare a signed 16-bit integer variable named "myInt", **you would write:**

To declare an unsigned integer variable, you use the **"uint"** keyword followed by the number of bits you want to allocate for the variable. For example, to declare an unsigned 8-bit integer variable named "myUint", **you would write:**

Once you have declared your integer variables, you can use them in your code just like any other variable. For example, you can assign values to them using the "=" operator, perform arithmetic operations on them, and use them in control structures like "if" statements and loops.

One thing to keep in mind when using signed integers is that they can **overflow or underflow** if you try to store a value that is too large or too small for the number of bits allocated to the variable. This can result in unexpected behavior and bugs in your code. To avoid this, you should always check the range of values that your signed integer variable can hold before assigning a value to it.

In conclusion, signed and unsigned integers are essential data types in Solidity, and they are used in many aspects of smart contract development. By understanding how to declare and use these data types, you can write more robust and reliable smart contracts that are better suited to your needs.

**ADDRESS**

An address in Solidity represents an Ethereum account, which is used to send and receive Ether and interact with smart contracts. In Solidity, addresses are 20 bytes long and are represented using the **"address"** data type.

Let's start by looking at how you can declare and initialize address variables in Solidity. To declare an address variable named "myAddress", **you would write:**

To initialize an address variable **with a specific value,** you can use the "address()" function, **like so:**

Once you have an address variable, there are several built-in functions and operators that you can use to work with it. These include:

* **"balance":** This function returns the balance of an address in Wei.
* **"transfer":** This function sends a certain amount of Ether from one address to another.
* **"send":** This function is similar to "transfer", but it returns a boolean value indicating whether the transfer was successful or not.
* **"call":** This function is used to call a function on another contract and can return a boolean value indicating whether the call was successful or not.
* **"this"**: This operator returns the address of the current contract.

It's important to note that addresses can also be passed as function arguments and returned as function return values. In addition, Solidity has several modifiers that can be used to check the validity of addresses, such as **"payable"** and **"nonpayable"**.

In terms of security, it's important to be careful when working with addresses, as they can be easily manipulated by attackers. For example, an attacker can try to send Ether to an address that is not a contract, resulting in the Ether being lost forever. To prevent these types of attacks, it's important to use best practices when handling addresses in your smart contracts.

**BYTES32**

In Solidity, bytes32 is a fixed-size array of 32 bytes, or 256 bits. It is commonly used to store and manipulate data in a compact and efficient manner.

The bytes32 type is often used for cryptographic purposes, such as storing hashes or signatures. For example, the SHA-256 hash of astring can be represented as a bytes32 value. **Here's an example** of how to compute a SHA-256 hash in Solidity:

The bytes32 type can also be used to store other types of data, such as fixed-length strings or integers. **For example**, here's how to store a fixed-length string as a bytes32 value:

In this example, the string "Hello, world!" is automatically converted to a bytes32 value and assigned to the myString variable.

One thing to keep in mind when using bytes32 values is that they are **fixed-size,** which means that they **always occupy 32 bytes of memory**, **even if the actual data being stored is smaller.** This can lead to wasted memory if you're storing small amounts of data in a large number of bytes32 values. To address this, Solidity provides dynamic-length byte arrays (bytes) and string types (string) that can be used to store variable-length data more efficiently.

Overall, bytes32 is a powerful and flexible data type in Solidity that is commonly used for cryptographic purposes and efficient data storage.