**CODELANDCS BLOCKCHAIN DEVELOPMENT SYLLABUS**

**WEEK 3**

**DAY 2**

**DATA TYPES CONTINUED**

**USER DEFINED VARIABLES**

User-defined types in Solidity allow you to create custom data types that can be used in your smart contracts. This is a powerful feature that can help you organize your code, make it more readable and maintainable, and enable you to create more complex data structures.

There are several types of user-defined types in Solidity, including structs, enums. Let's take a closer look at each of these.

**STRUCTS**

Structs are one of the user-defined types available in Solidity, and they are used to define custom data structures that contain multiple variables of different types. Structs can be very useful for organizing your code and creating more complex data structures that represent real-world objects or concepts.

To define a struct in Solidity, you use the **struct keyword** followed by the name of the struct and the list of variables that it contains.

- We can also access the variables of a struct using the **dot notation**

- In addition to defining structs, you can also use them to **create arrays** of structs.

- We can equally nest a struct within a struct.

**ENUMS**

Enums are used to define a set of **named values** that represent **possible states for a variable**. Enums are used to restrict the possible values of a variable to a predefined set of values. Enums are also called **enumerated types** or enumeration. Enums are similar to arrays, but they can only have a fixed set of values.

Enums in Solidity are defined using the **enum keyword,** followed by the identifier name**.**

The values of the enum are automatically assigned an integer value, starting from 0 and incrementing by 1 for each value.

**VARIABLE SIZED TYPES**

Variable sized types are data types that can hold a variable amount of data. Unlike fixed sized types such as uint8 or bool, variable sized types can store a variable amount of data depending on the input. Some examples of variable sized types in Solidity include strings, bytes, mappings and arrays.

**STRING**

In programming, a string is a sequence of characters. In Solidity, a string is a sequence of UTF-8 encoded characters, which means that it can include any Unicode character.

In Solidity, you can declare a string variable using the **"string"** keyword.

There are several built-in functions that you can use to manipulate strings in Solidity. These include:

* **"bytes(string)":** This function converts a string to a byte array.
* **"string(bytes)":** This function converts a byte array to a string.
* **"bytes.length":** This function returns the length of a byte array.
* **"string.length":** This function returns the length of a string.
* **"string.charAt(uint)":** This function returns the character at the specified index in a string.
* **"string.substr(uint,uint)":** This function returns a substring of a string, starting from the specified index and with the specified length.

It's important to note that **strings in Solidity are not immutable.** This means that you can change the value of a string variable after it has been assigned a value. However, you should be careful when modifying strings, as doing so can be inefficient and lead to high gas costs.

In conclusion, strings are an important data type in Solidity, and they are used in many aspects of smart contract development. By understanding how to declare, assign, concatenate, and manipulate strings, you can write more powerful and flexible smart contracts that are better suited to your needs.

**ARRAYS**

Arrays are a fundamental data structure used in many programming languages, and Solidity is no exception. In Solidity, arrays are used to store collections of values of the same type. They can be either **fixed-length** or **variable-length** and can be of any type, **including arrays of arrays.** Arrays are commonly used in Solidity for storing and manipulating data. They can be used to store data of any type, including other arrays.

**Fixed-Length Arrays**

In Solidity, fixed-length arrays are declared with a **specific number** of elements. The length of the array is fixed, and once it is defined, it cannot be changed. Fixed-length arrays are useful when the number of elements in the array is known in advance. For example, an array of 10 integers can be declared as **int[10].**

**Variable-Length Arrays**

Variable-length arrays, on the other hand, **can change in size during runtime.** They are declared without a specific number of elements, and their size can be changed using the push() and pop() functions. When declaring a variable-length array in Solidity, you can use the dynamic keyword to indicate that the size of the array is variable. For example, an array of integers can be declared as **int[].**

**Multidimensional Arrays**

Solidity also supports multidimensional arrays, which are **arrays of arrays.** Multidimensional arrays can be declared as an array of arrays, where each element of the outer array is itself an array. For example, a two-dimensional array of integers can be declared as **int[][2].**

**Accessing Array Elements**

In Solidity, you can access an element in an array using its index. The index starts at 0 for the first element and increases by 1 for each subsequent element. You can also access multiple elements in an array using a for loop or by using the array's length property.

**Array Functions**

Solidity provides several built-in functions for working with arrays. These include **push(),** **pop(),** **length,** and **splice().** The push() function adds an element to the end of the array, while the pop() function removes an element from the end of the array. The length property returns the number of elements in the array, and the splice() function can be used to remove or replace elements in an array.

**Best Practices When Working With Arrays**

When working with arrays in Solidity, it is important to be mindful of gas costs. Accessing and modifying arrays can be expensive, especially when working with variable-length arrays. It is also important to consider security concerns when working with arrays, as they can be vulnerable to attacks such as buffer overflow and integer overflow.

**Conclusion**

Arrays are a fundamental data structure in Solidity, used for storing and manipulating data of the same type. They can be either fixed-length or variable-length, and can be of any type, including arrays of arrays. When working with arrays in Solidity, it is important to be mindful of gas costs and security concerns. By understanding how to use arrays in Solidity effectively and efficiently, developers can create more efficient and secure smart contracts on the Ethereum blockchain.

**MAPPINGS**

A mapping is a collection of key-value pairs where each key is unique and maps to a specific value. Mappings are similar to hash tables or dictionaries in other programming languages. In Solidity, mappings are defined using the **keyword mapping**, followed by the type of the key and the value.

**Syntax and Usage of Mappings in Solidity**

Mappings can be used to store any type of data, including other mappings, arrays, and structures. Here are some examples of how mappings can be used in Solidity:

**Example 1: Storing Balances**

The most common usage of mappings in Solidity is to **store balances of cryptocurrency tokens.** Each address has a balance associated with it, which is updated whenever a transfer is made.

**Example 2: Storing Structs**

Mappings can also be used to store structs. In this example, we define a struct named Person that has two fields: name and age. We then create a mapping named people that maps addresses to Person structs.

**Limitations of Mappings in Solidity**

While mappings are a powerful data structure in Solidity, there are some limitations to their usage. Here are some of the key limitations:

* Mappings **cannot be iterated over**: Unlike arrays, mappings do not have a fixed length and cannot be iterated over. This makes it difficult to perform certain operations, such as finding the key with the maximum value or sorting the keys.
* Mappings **cannot be passed as function arguments**: Mappings are a complex data type and cannot be passed as function arguments. This means that if you want to pass a mapping to a function, you need to pass each key-value pair individually or create a separate data structure that contains the mapping.
* Mappings **cannot be used in memory or storage arrays**: Mappings are designed to be used as standalone data structures and cannot be used as elements in memory or storage arrays. This is because mappings do not have a fixed length, and **Solidity requires fixed-length arrays to be used in memory** and storage.
* Mappings **can consume a lot of gas**: When a mapping is updated, **Solidity needs to perform a hash function to determine the location of the value in memory.** This can consume a significant amount of gas, especially if the mapping is large or frequently updated. To reduce gas consumption, it's important to minimize the number of mapping updates and use other data structures when possible.

**BUILT IN VARIABLES**

The most commonly used built-in variables in Solidity are as follows:

**msg.sender**

The msg.sender variable is used to represent the Ethereum address of the account that is **currently interacting with the contract.** This is useful for determining who initiated a transaction or for implementing access control in a smart contract.

**msg.value**

The msg.value variable is used to represent the amount of Ether that was sent with a transaction. This is useful for contracts that require payments to be made in order to execute certain functions.

**block.timestamp**

The block.timestamp variable is used to represent the timestamp of the block that the current transaction is included in. This is **useful for implementing time-based functions in a smart contract.**

**block.difficulty**

The block.difficulty variable is used to represent the difficulty level of the current block. This is useful for **verifying that a block has been mined correctly** and for calculating the amount of work that was done to mine the block.

**block.number**

The block.number variable is used to represent the number of the current block. This is **useful for tracking the progress of the blockchain** and for determining how many blocks have been mined since the genesis block.

**now**

The now variable is a shorthand for block.timestamp. It is used to represent the current timestamp and is useful for implementing time-based functions in a smart contract.

**tx.origin**

The tx.origin variable is used to represent the address of the account that originally sent the transaction. This is useful for implementing access control in a smart contract.

However, it should be noted that using tx.origin for access control can be dangerous, as it is possible **for an attacker to spoof the origin address** of a transaction.

**gasleft()**

The gasleft() function is used to represent the amount of gas remaining in the current transaction. This is **useful for optimizing gas usage** in a smart contract.