**Smart Contracts Overview**

It should be noted in this video Dan said that the amount of gas is static to the opcodes used. What he meant is that the amount of gas is deterministically calculated based on the opcodes used. For simple opcodes, like ADD or MUL the gas amount is a fixed amount. For more complicated opcodes, like SSTORE, there are several factors that go into the gas calculation and there are even gas refunds for clearing of a storage slot (see evm codes for more details).

1. **Boolean**

Storage Variables Let's begin with storage variables! These variables are stored in a contract's permanent data storage on the blockchain. When you modify a storage variable in a transaction, the updated value becomes globally accessible for subsequent reads and interactions. In Solidity, declaring a storage variable is as simple as declaring the variable inside of the contract:

**contract Contract {**

**bool myVariable;**

**}**

The Contract now has a boolean storage variable called myVariable! Sweet. Curious about the initial value of myVariable? Since it's uninitialized, its storage location holds 0x0. In Solidity, that translates to a default value. For booleans like myVariable, the default is false. Therefore, upon contract deployment, myVariable starts off as false. Now we're going to do two things to our variable: make it public and give it an initial value of true:

**contract Contract {**

**bool public myVariable = true;**

**}**

See how we added the keyword public here? This automatically creates a getter function for the variable. Now we can access the value in myVariable by calling a function on the contract with that very name: myVariable().

1. **Unsigned Intigers**

Unsigned Integers What the heck is an unsigned integer? First, let's go over integers. Integers include all positive and negative numbers without fractions. The numbers -2,-1,0,1,2 are all integers. This range continues in both the positive and negative direction as far as you can count! To determine if the number is above or below zero we use the sign: + or -. An unsigned integer is an integer that has no sign. Solidity has a specific data type for unsigned integers called uint. A uint can be suffixed with the number of bits reserved for it. For instance uint8 means that there are eight bits provided for the value of the variable. What is the range of unsigned integers in eight bits? Eight bits can range from 00000000 to 11111111. This range can represent 256 unique values. Since the range of unsigned integer values does not include negative numbers, it is simply 0 to 255. In decimal, the unsigned value of 00000000 is 0 and the value of 11111111 is 255. So, the max value a uint8 can store is 255. You can calculate the max value of a uint by doing (2 ^ n) - 1, where n is the number of bits it can use to store the value. Wondering what happens if we add two uint8 values together whose sum will exceed 255? Let's take a look at this in details.

1. **Signed Intigers**

Signed Integers Now that we know what an unsigned integer is, let's take a look at a signed integer. A signed integer can be declared with the keyword int. Just like uint, the keyword int is short for a data type that will store 256 bits of memory, int256. Since an integer is signed, the range covers both negative and positive numbers. Let's compare the range of a uint8 to an int8: uint8: Ranges from 0 to 255 int8: Ranges from -128 to 127 Notice that both ranges cover a total of 256 values, which is the total number of possible values that can be expressed with 8 bits.

1. **String Literals**

String Literals We can create strings of characters using double quotes, the string literal "Hello World" is perfectly valid in Solidity. You'll often see fixed values described as a literal. The value "Hello World" can be described as a string literal which differentiates it from the string data type. Any fixed value could be a literal, "Hello World", 42, or true. A string literal can be stored in both the bytes and string types:

**bytes msg1 = "Hello World";**

**string msg2 = "Hello World";**

For a long human-readable message it is recommended to use string since it will be easier to read the values from the blockchain storage from the outside (like for a front-end application). If the string is shorter than 32 bytes, it is more efficient to store it in a fixed-size byte array like bytes32. This simplifies the computation since the memory is allocated ahead of time. On the other hand, both string and bytes will allocate their memory dynamically depending on the size of the string. How many characters can be stored in bytes32? Well this is actually depends on the characters themselves! Many characters in UTF-8 encoding can be represented with 1 byte while others are represented with several bytes. For instance c is encoded by 0x63, while ć is encoded by 0xc487. So the maximum values would be: **bytes32 msg1 = "cccccccccccccccccccccccccccccccc";**

**bytes32 msg2 = "ćććććććććććććććć";**

Adding a character to either string will result in a compile-time error since the string literal would no longer fit into 32 bytes. Quite often long strings are stored seperately on other distributed services like IPFS, with a hash representation stored on the blockchain (since storage on a blockchain is expensive!). For example, you could write a legal document and hash the contents along with digital signatures to prove that it was signed. As long as the original document is preserved it can be easily proven that it was signed by rehashing the contents.

1. **Enum**

Enum Type The Enum Type helps us write clean code! Consider this example:

**if(player.movement == 0) {**

**// player is moving up**

**}**

**else if(player.movement == 1)**

**{**

**// player is moving left**

**}**

Those comments are helpful, but they aren't exactly a foolproof plan! The movement number is being generated somewhere else in the code. If that ever changed, it would break our code! Plus, without the comments, there would be no way to tell which direction is which! An enum can clean this up! Let's see:

**enum Directions = { Up, Left, Down, Right }**

**if(player.movement == Directions.Up) {**

**}**

**else if(player.movement == Directions.Left)**

**{**

**}**

Much cleaner! Not only are the numbers replaced with clear directions, we also have a structure for defining all our directions. We can share this structure, Directions, with other contracts to ensure that if the numbers change they won't break the rest of the code!