**Zombies attack their victims**

Zombies attack their victims is the second course on cryptozombies.io and it delves deeper into solidity and more advanced concepts. For the first time in the course the concept of address is touched upon, which is one of the central ideas to blockchain. As has been elaborated in the course, an address is owned by a specific user (or smart contract) and holds an amount of Ether, the famous currency based on the Ethereum blockchain.

Addresses are highly important since only external callers can execute functions in solidity, and it is often very important to get a hold of who issued that call. For example we might not want to allow users to abuse zombie creation and limit to 1 (or more) zombies per account. This is achieved by keeping a map in storage that holds the list of zombies where the key is the address. To access the address you need only type msg.sender, which is a global variable that always contains a valid address on the blockchain. Since it has been mentioned previously that only entities with an address can execute code, there will always be a msg.sender to access.

Mappings are not too dissimilar to maps in other programming languages, solidity makes using this data structure pretty simple. They live on the blockchain, not in memory, and you simply declare the data type of the key and the data type of the value. They are in my opinion most similar to the same data structure from Go, Go also only requires specifying a key and a value, and it is so easy to use that you often don’t need to initialize it, it will figure out that a nil value is an empty map (in Solidity this isn’t required, the mapping is a permanent storage object which can benefit from certain assumptions). Ultimately mappings are most often used to correlate addresses with other objects, and to that end they serve their purpose quite well.

Solidity supports inheritance between contracts, which is similar to inheritance between classes from OOP. It can be used as logical inheritance (such as the example given that a Cat is an Animal), but also just as a way to group code and avoid extremely large contracts. Inheriting from a contract allows the subclass to access everything except private attributes and methods. Solidity has private and public modifiers, but also internal and external. Internal fields are accessible to contracts that inherit from this contract and inside the parent contract, while external fields are accessible only from other contracts.

Another concept borrowed from OOP is interfacing. Continuing the trend of keeping things very simple, an interface contract is a contract that contains just functions with no bodies, and no state variables. If you define an interface with the name of a contract already on the blockchain you will gain access to those methods on the blockchain, and you will call methods from those other contracts. The example given in the course allows you to get the traits of a cat from a random seed.

The last feature presented in the course is another feature borrowed from Go, that of multiple return values. If you’re familiar with Go, then it will feel very familiar to use the same feature in Solidity, with the exception that Solidity uses parenthesis to accept multiple parameters from a method. In my opinion this is a feature that more programming languages should implement, you shouldn’t create structs just to return slightly more data for some methods. Although Go’s reason for doing this is its own distinct error handling, we are yet to see whether Solidity follows the same pattern for error handling as Go.

Finally, I believe the course has expanded our basic Solidity knowledge with more crucial features, all while applying them in an application where we feel they serve a practical purpose, which makes learning much more fun and engaging.