Solidity Hack 101 - level 3

Introduction To If/Else Statements

In Solidity, developers can use if/else statements to control the flow of their code based on specific conditions. These conditions are evaluated using various comparison operators, including less than (<), greater than (>), equal to (==), and more. Depending on whether the conditions are met or not, different blocks of code are executed.

Syntax:

The basic syntax of an if/else statement in Solidity is as follows:

```
if (condition) {
// code Executed if the Condition is true
} else if (condition) {
// code Executed if the condition is true
} else {
// code Executed if none of the above conditions are true
}
```

Example Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
contract IfElse {
function foo(uint x) public pure returns (uint) {
if (x < 10) {
return 0;
} else if (x < 20) {
return 1;
} else {
return 2;
}
}
function ternary(uint _x) public pure returns (uint) {
// if (_x < 10) {
// return 1;
// }
// return 2;
// shorthand way to write if/else statement
// the "?" operator is called the ternary operator
return _x < 10 ? 1 : 2;
}
}
```

Introduction To For or While loops

For Loop

In Solidity, for loops are used to iterate over a set of values or perform a specific block of code a certain number of times. Here's the syntax for a for loop in Solidity

```
for (initialization; condition; increment) {
    // Code to be executed
}
```

Example Code

```
pragma solidity ^0.8.0;

contract ExampleContract {
    function sumNumbers(uint256 n) public pure returns (uint256) {
        uint256 sum = 0;

        for (uint256 i = 1; i <= n; i++) {
            sum += i;
        }

        return sum;
    }
}
```

In the above example, the `sumNumbers` function takes an input `n` and calculates the sum of numbers from 1 to `n`. The for loop initializes a variable `i` to 1, executes the loop as long as `i` is less than or equal to `n`, and increments `i` by 1 in each iteration. Inside the loop, the value of `i` is added to the `sum` variable.

For instance, if you call `sumNumbers(5)`, the function will return 15 because it calculates the sum of numbers 1 + 2 + 3 + 4 + 5.

Things To be Know

When utilizing loops in Solidity, it is crucial to exercise caution to avoid unintended consequences. Here are some considerations and best practices to bear in mind:

1. Gas Limit: Loops in Solidity consume gas with each iteration. Unbounded loops or loops with excessively large iteration counts can lead to gas limit exceeded errors, resulting in transaction failures. Always ensure that your loops have a reasonable upper bound or iterate over a fixed-sized array or a known set of elements.
2. Loop Efficiency: It is generally advisable to minimize the amount of computation performed within loops, as each iteration contributes to gas costs. If feasible, move complex computations or external interactions outside the loop to optimize gas usage.
3. Storage Costs: Be mindful of storage costs when working with loops. Assigning values to storage variables within loops can result in multiple storage writes, which can be expensive. Consider utilizing memory variables or optimizing data structures to reduce storage operations.
4. Eventual Consistency: Due to the asynchronous nature of blockchain execution, changes made within a loop may not be immediately visible to other contracts or external systems. Keep this in mind when designing smart contracts and ensure that eventual consistency is taken into account.