ETHEREUM 2.0 MASTERY PROGRAM

Instructor: Raja Rizwan Saleem

MODULE-2

BLOCKCHAIN AND SMART CONTRACT BASICS

Raja Rizwan Saleem
Lead Blockchain Trainer

Class-10

5. Modifiers continued....

Modifiers practical coding

solidity

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not the owner");
    _;
}
```

solidity

```
modifier validAmount(uint256 _amount) {
    require(_amount > 0, "Amount must be greater than zero");
    _;
}
```

Structure of a contract

A contract consists of the following multiple constructs

1. State variables
2. Function definitions
3. Enumeration definitions
4. Structure definitions
5. Modifier definitions
6. **Event declarations**

6. Events

Events

1. Events are fired from contracts such that anybody interested in them can trap/catch them and execute code in response. Events in Solidity are used primarily for informing the calling
2. application about the current state of the contract by means of the logging facility of EVM. They are used to notify applications about changes in contracts and applications can
3. use them to execute theirdependent logic

Events

1. Event is an inheritable member of a contract.
2. An event is emitted, it stores the arguments passed in transaction logs.
3. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain.
4. An event generated is not accessible from within contracts, not even the one which have created and emitted them.

Event Declaration and Usage

```
event ageRead(uint);

function getValues(uint a) public returns (uint){
    emit ageRead(a);

    return 56;
}
```

Event Declaration and Usage

```
pragma solidity ^0.8.1;

// Creating a contract
contract eventExample {


// Declaring state variables
uint256 public value = 0;

// Declaring an event
event Increment(address owner);


// Defining a function for logging event
function getValue(uint _a, uint _b) public {
      emit Increment(msg.sender);
      value = _a + _b;
} }
```

Event Declaration and Usage

```
contract First{

    uint age;
    event ageRead(uint);

    function updateAge(uint _age) public {

        age = _age;
        emit ageRead(age);
    }

}
```

For Loop

For Loop

1. Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.
2. The ***for*** statement creates a loop that is executed as long as a condition is true.
3. It will only stop when the condition becomes false.
4. ***for*** is a keyword in Solidity and it informs the compiler that it contains information about looping a set of instructions.

For Loop

```
for (initialization; condition; expression) {
    // code to be executed
}
```

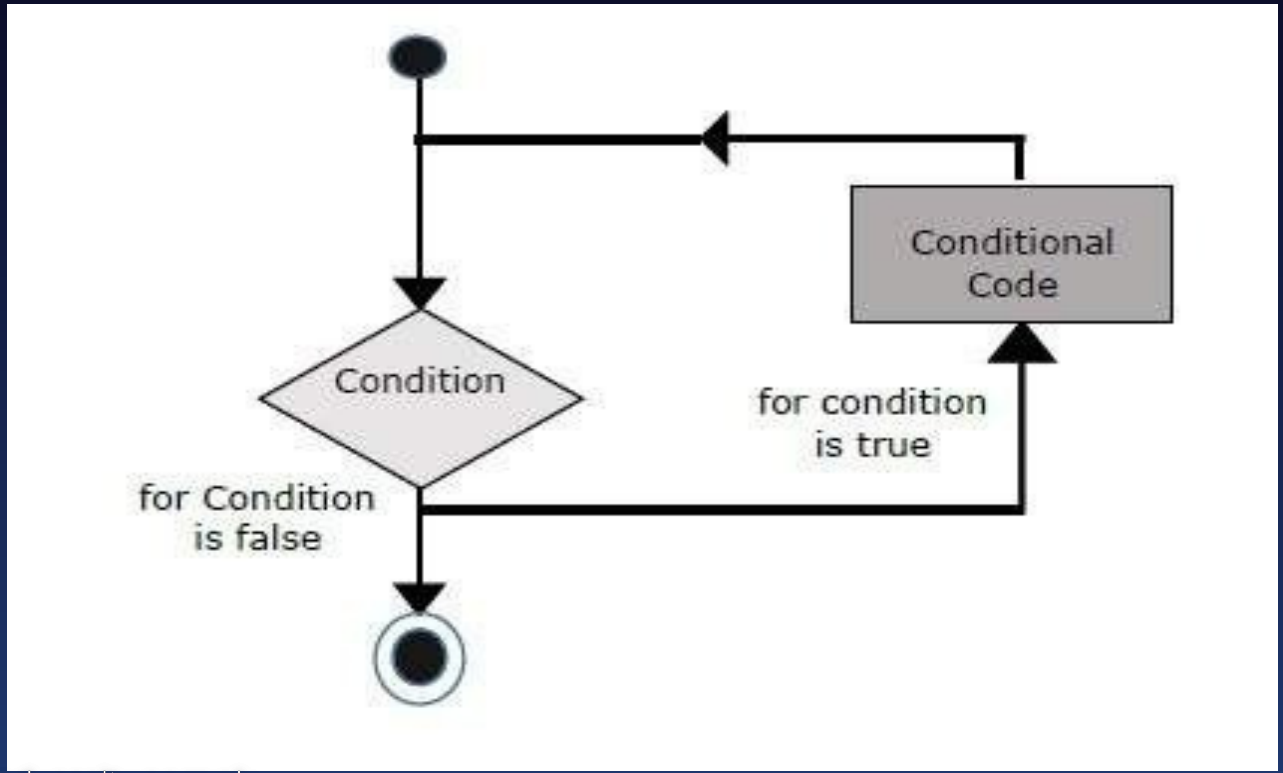
1. Initialization is done (one time) before the execution of the code block.
2. Condition for executing the code block and exit loop
3. Expression is executed (every time) after the code block has been executed.

For Loop

The for loop is the most compact form of looping. It includes the following three important parts –

- The loop initialization where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.
- The test statement which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.
- The iteration statement where you can increase or decrease your counter.

For Loop - Flow Chart



For Loop

```
for (initialization; test condition; iteration statement)
{
   Statement(s) to be executed if test condition is
true
}
```

For Loop

```
pragma solidity ^0.5.0;
contract Types {

    uint[] data;

    function loop(      ) public returns(uint[] memory){
    for(uint i=0; i<5; i++){
        data.push(i);
    }
    return data;
    }
}
```

For Loop

```
pragma solidity ^0.8.0;
// for loop test
contract whileTest {

      uint result = 0;
      function sum() public returns(uint data){
      for(uint i=0; i<10; i++){
            result = result+i;
       }
         return result;
      }
}
```

Infinite Loop

1. All 3 statement in loop are options, in that case it will be infinite loop
2. Also if you do not provide condition in loop it will make loop infinite
3. In Solidity every expression and statement cost gas unit
4. If you create infinite loop, it will consume all gas provided and then transaction will failed because gas provided by user is limited so transaction will fail once gas finished

Infinite Loop

```
for ( ; ; ){
    // code to be executed
}
```

Break in Loops

Break

1. There are times when you would like to stop the iteration of loop in between and jump out or exit from the loop without executing the conditional test again.
2. The **break** statement helps us do that.
3. It helps us terminate the loop by passing the control to the first instruction after the loop.

Break

```
event logInt(int);
function iteration() public{
    for (int i = 0; i < 8; i++){
        if(i ==     4) {
            break;
        }
        emit logInt(i);
    } }
```

Continue in Loops

Continue

1. Loops are based on expressions.
2. The logic of the expression decides the continuity of the loop.
3. However, there are times when you are in between loop execution and would like to go back to the first line of code without executing the rest of the code for the next iteration.
4. The **continue** statement helps us do that.

Continue

```
event logInt(int);
function iteration() public{
    for (int i = 0; i < 8; i++){
        if(i ==     4) {
            continue;
        }
        emit logInt(i);
    } }
```

While Loop

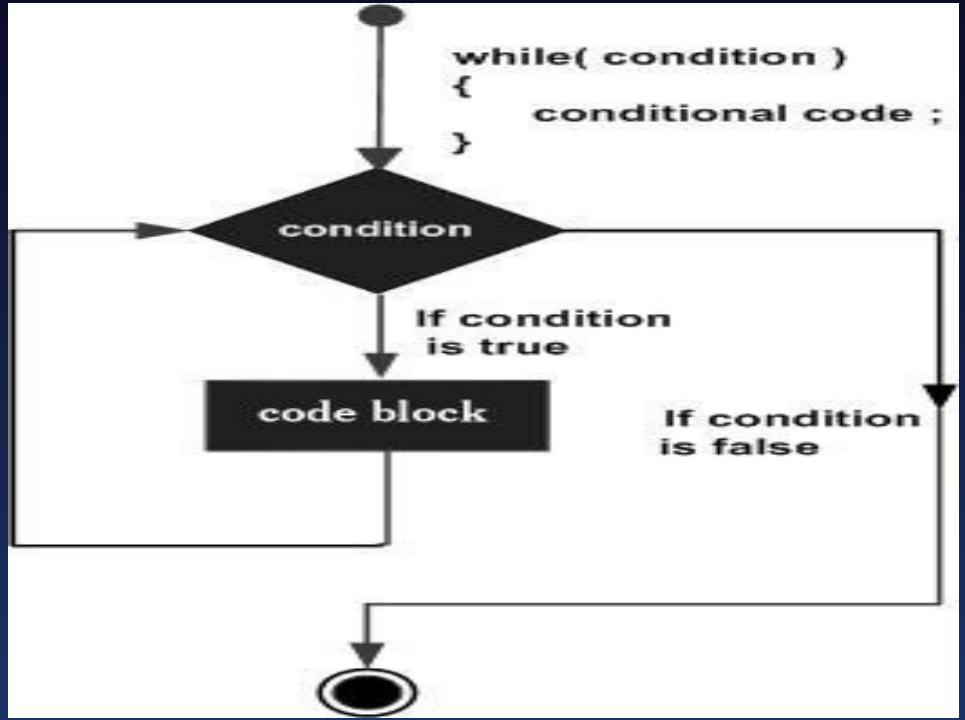
While Loop

1. ***while*** is a keyword in Solidity and it informs the compiler that it contains a decision control instruction.
2. If this expression evaluates to true then the code instructions that follow in the pair of double-brackets { and } should be executed.
3. The while loop keeps executing until the condition turns false.

While Loop

The purpose of a **while** loop is to execute a statement or code block repeatedly as long as an **expression** is true. Once the expression becomes **false,** the loop terminates.

While Loop


while(condition)
{
conditional code ;
}
condition
If condition
is true
code block
If condition
is false

While Loop

Syntax:

```
while (condition) {
    // code block to be executed
}
```

While Loop

1. In this example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

```
while(i<8) {
    emit logInt(i);
    i++;
}
```

2. You will use while loop when execution is dependent on user input

While Loop

```
pragma solidity ^0.8.1;
contract whileTest {
    event Loguint(uint8);

    function demoWhileLoop () public{
        uint8  i  =  0;
        while (i<8){

            emit Loguint(i);
            i++;
        }
    }
}
```

while Loop

```
pragma solidity ^0.5.0;
contract Types {
            uint[]     data; // Declaring a dynamic array
            uint8 j = 0;      // Declaring state variable
    function loop() public view returns(uint[] memory){
    while(j < 5) {
        j++;
        data.push(j);
    }
    r e t u r n   d a t a ;
    }
}
```

While Loop

```
pragma solidity ^0.5.0;
// While loop test
contract whileTest {

    uint8 i = 0;
    uint8 result = 0;

    function sum() public returns(uint data){
    while(i < 3) {

        i++;
        result=result+i;
     }
      return result;
    }
```

While Loop

```
pragma solidity ^0.5.0;
// While loop test
contract whileTest {
      uint8 i = 0;
      uint8 result = 0;

      function sum() public returns(uint data){
      do{
            i++;
            result=result+i;
               }
      while(i < 3) ;
         return result;
      }}
```

Do/While Loop

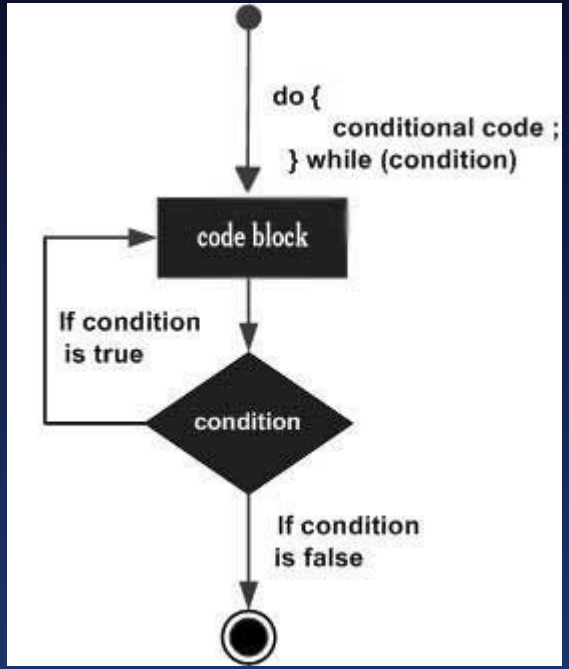
Do/While Loop

1. The do/while loop is a variant of the while loop. This loop will
2. execute the code block once, before
   checking if the condition is true, then it will repeat the loop
   as long as the condition is true.
3. You will use do/while loop when execution is dependent
   on user input but it needs to run code block at least once

Do/While Loop

The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is **false**.

Do/While Loop



Do/While Loop

Syntax:

```
do{
    // code block to be executed
}
while (condition);
```

Do/While Loop

```
int i=0;
do{
    emit logInt(i);      i++;
}
while (i < 10);
```

Do/While Loop

```
pragma solidity ^0.8.1;
contract whileTest {
    event Loguint(uint8);

    function demoDoWhileLoop () public{
        uint8 i = 8;
        do {
        emit Loguint(i);
            i++;
        }
        w h i l e   ( i < 8 ) ;

        } }
```

Do/While Loop

```
pragma solidity ^0.5.0;
contract Types {
    uint[]   data;
    uint8 j = 0;

    function loop( ) public returns(uint[] memory){
    do{
        j++;
        data.push(j);
    }

    while(j < = 5) ;
    return data;
    }}
```

THANK-YOU