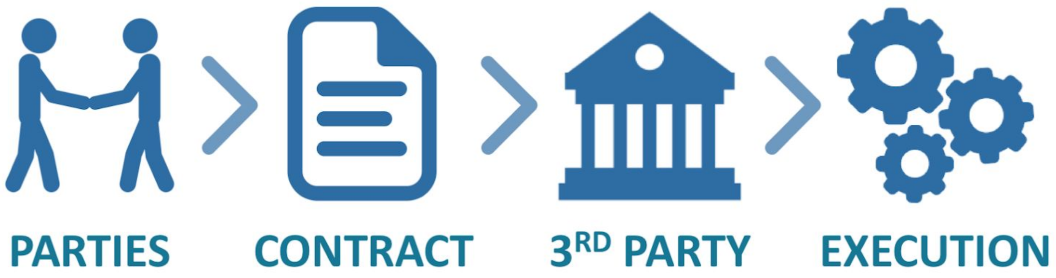
Smart Contract

Definition - Smart Contract

Smart contracts help you exchange money, property, shares, or anything of value in a transparent, conflict-free way while avoiding the services of a middleman.

Contract VS Smart Contract

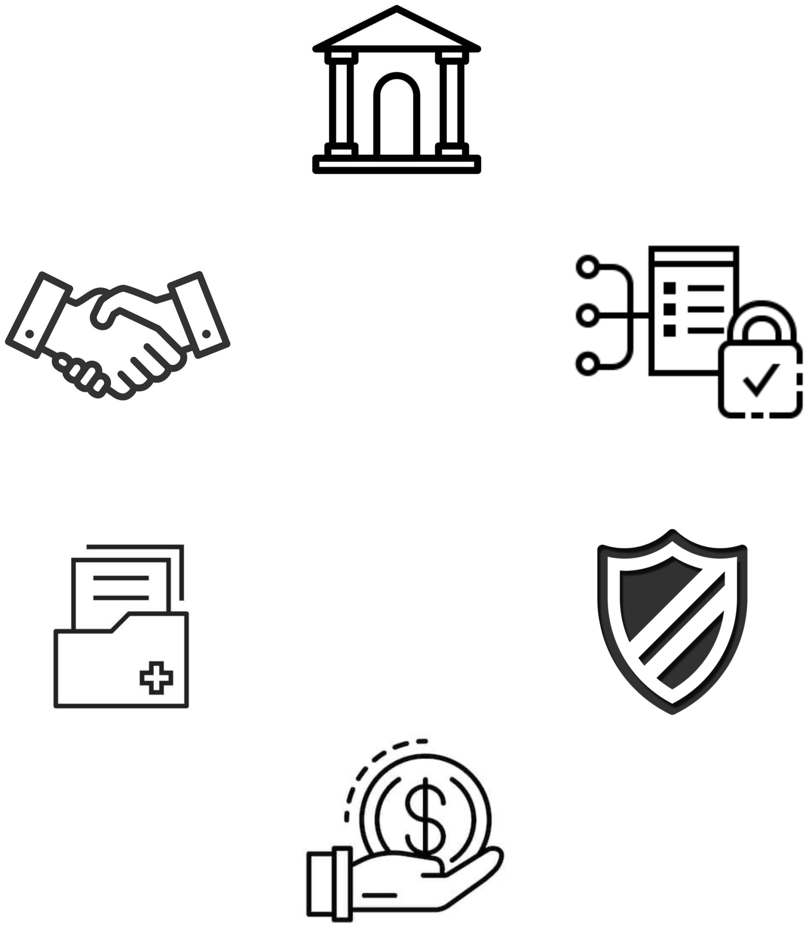
TRADITIONAL CONTRACT



SMART CONTRACT

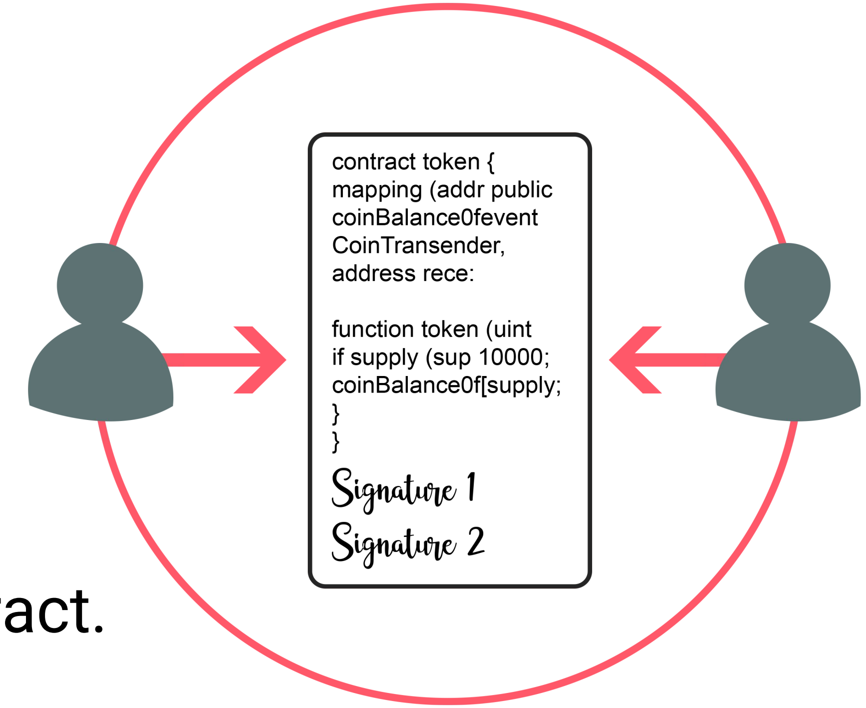
Benefits - Smart Contract

- Direct dealing with Customers.
- Resistance to Failure.
- Immutability.
- Fraud Reduction.
- Cost Efficiency.
- Record Keeping.



Features - Smart Contract

- Contract written as code into the Blockchain.
- Contract is part of the Public Blockchain.
- Parties involved in the contract are anonymous.
- Contract execute itself when condition are met.
- Regulators use blockchain to keep an eye on contract.



K B A

Solidity

Definition - Solidity

Created By : **Gavin Wood**

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.

Features - Solidity

- Influenced by **C++, Python and JavaScript**.
- Designed to target the **Ethereum Virtual Machine (EVM)**.
- Solidity is **statically typed**.
- Supports:
  - Inheritance
  - Libraries
  - Complex user-defined types

Code 1 - Solidity

```
pragma solidity ^0.5.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Code 1 (Review) - Solidity

```
pragma solidity ^0.5.0;
```

Defining solidity version: 0.5.0

```
contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Contract Name: SimpleStorage

Global Variable: storedData (Private)

Function 1: set (public) inputting uint

Function 2: get (public) returning uint

Four High-Level Constructs - Solidity

- Pragma
- Comments
- Import
- Contracts/library/interface

Pragma - Solidity

Pragma is generally the first line of code within any Solidity file. `pragma` is a directive that specifies the compiler version to be used for current Solidity file.

Syntax: `pragma Solidity <<version number>>;`

Eg: `pragma Solidity ^0.5.0;`

`pragma experimental ABIEncoderV2;` (Returning Structure)

Comments - Solidity

Used for describing the code, author, etc

- Single-line comments (//)
- Multi-line comments (/*...*/)

Eg:

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Import - Solidity

- Solidity supports import statements.
- At a global level, you can use import statements of the following form:

```
import "filename";
```

- Eg:

```
import "my.sol";
```


Contract - Solidity

Syntax:

```
contract contractName {

}
```

Eg:

```
contract SimpleStorage{

}
```

State Variable - Solidity

State variables are variables whose values are permanently stored in contract storage. Changing value requires gas.

```
pragma solidity ^0.5.0;

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

Datatype - Solidity

Different datatypes:

1. Value types
2. Reference Types

Value type - Solidity

Variables of these types will always be passed by value.

- Booleans - bool
- Integer - int, int8, int16, int32, .. int256
- Unsigned Integer - uint, uint8, uint16, uint32, .. uint256
- Fixed point numbers (not yet supported) - fixed
- Address - address
- Enum - enum
- Fixed size byte array - bytes, bytes1, bytes2, .. bytes32
- String literals - string

Operators - Solidity

- **Booleans**: “bool”
  - Possible values are “true” and “false”
  - Operators : !, && (and), || (or) etc

- **Integers**: int/uint (8 to 256)
  - Operators:
    - comparison: <=, <,==,>=,>
    - Bit operators: &,|,^,~
    - Shift operators: <<, >>
    - Arithmetic operators: +,-,*,/,%,**

Operators - Solidity

- **Address**: address, address payable
  - Address payable: address you can send ether to.
  - Operators: <=,<,==,<,!=,>=,>
  - Members of addresses :balance transfer and Send
- **Enum Types**
  - Enums are used to build up a set of values. Internally enums are treated as uints.

```
enum daysOfTheWeek {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
daysOfTheWeek today = daysOfTheWeek.Tue;
```

Eg 1: Hello World - Solidity

Creating a “Hello World” contract !

Eg 2: Arithmetic Operation - Solidity

Creating a smart contract for adding 2 numbers !

Reference Types - Solidity

These variable types are passed by reference and better classified as data structures.

- Structure - struct
- Arrays - [ ]
- Mapping - mapping

Structure - Solidity

A structure allows you to combine several types together.

```
struct  Payment {
   uint amount;
   uint invoiceNumber;
   address account;
}

Payment myPay = Payment(500, 1207, accountNumber);
```

or

```
Payment myPay = Payment({account : accountNumber, amount:
500, invoiceNumber: 1207});
```

Eg 3: Structure - Solidity

```
pragma solidity ^0.5.0;

contract KYC{
    struct details{
        string name;
        uint age;
    }
}
```

Arrays - Solidity

Arrays can be fixed or dynamically sized and can be inline as a literal

- Fixed size array declaration - `uint[10] fixedArray;`
- Update the value in a fixed array - `fixedArray[3] = 22;`
- Dynamic array - `uint[] dynamicArray;`
- Add a new value to the end of a dynamic array - `dynamicArray.push(22);`

Eg 4: Array

Insert a number to an array and display the last inserted value !

Mapping - Solidity

Mappings are **key/value pairs** stored without respect to order.

- Declare a mapping - mapping(address => uint) public accountBalances;
- You look up the value by the key - uint amount = accountBalances[account];
- You set the value by specifying a key and value -

```
accountBalances[account]= 22;
```

- You cannot iterate over a mapping. If a key value does not exist the mapping will return 0. Keys cannot be reference types.

Eg 5: Mapping - Solidity

```
pragma solidity ^0.5.0;

contract KYC{
   struct details{
      string name;
      uint age;
   }

   mapping(address => details) person;
}
```

Memory vs Storage vs Constant - Solidity

Memory

- Used to hold temporary values.
- It is erased between (external) function calls and is cheaper to use.

```
uint memory tempVar = 100;
```

Storage

- All the contract state variables reside.
- Every contract has its own storage and it is persistent between function calls.
- Quite expensive to use.

Constant

- You can set a constant as long as it is a value type and it is set by literals.

```
uint constant max = 2**8;
```

Functions - Solidity

Functions are the executable units of code within a contract.

```
function funcName(params) [visibliy [mutability [returns(params)]]]{

}
```

```
function set(uint x) public {
  // …
}
```

Visibility - Solidity

- **external**
  External functions are part of the contract interface, which means they can be called from other contracts and via transactions.

- **public**
  Public functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function is generated.

Visibility - Solidity

- **internal**
  Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it).

- **private**
  Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

State Mutability - Solidity

- **Default**
  - A function can access and modify all contract's components.

- **View**
  - Function can wrap complex operations, or just provide a way to expose internal component in a safe way.

- **Pure**
  - Doesn't use any internal variables of the contract, either for a read or a write.

- **Payable**
  - provide a mechanism to collect / receive funds in ethers to your contract.

View Function - Solidity

Functions can be declared `view` in which case they promise not to modify the state.

```
pragma solidity ^0.5.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + now;
    }
}
```

Pure Function - Solidity

Functions can be declared `pure` in which case they promise not to read from or modify the state.

```
pragma solidity ^0.5.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

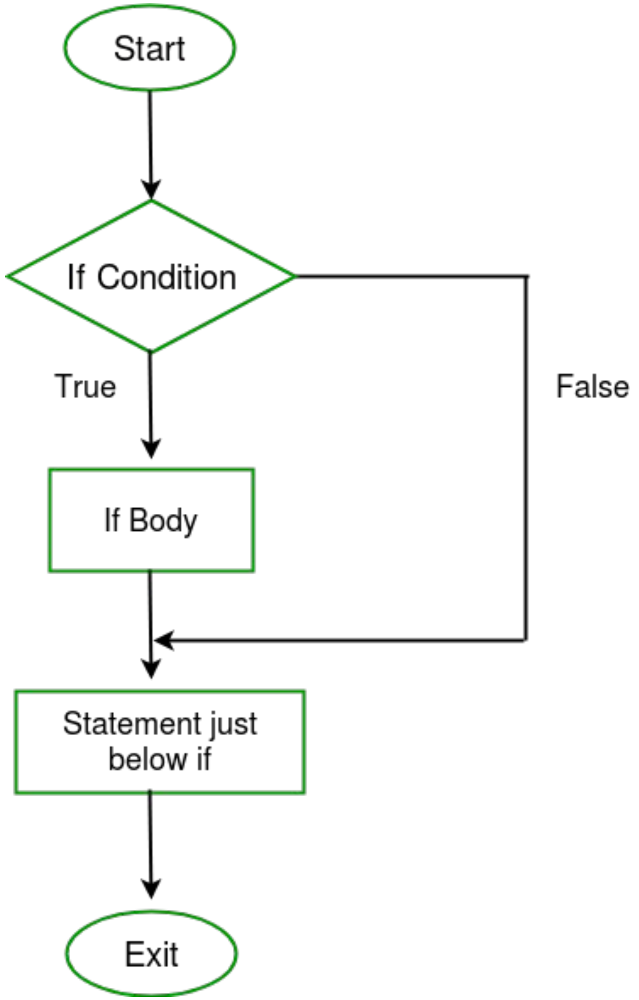
IF Statement

Syntax :

```
if(condition) {
    //if body
}
```

Eg:

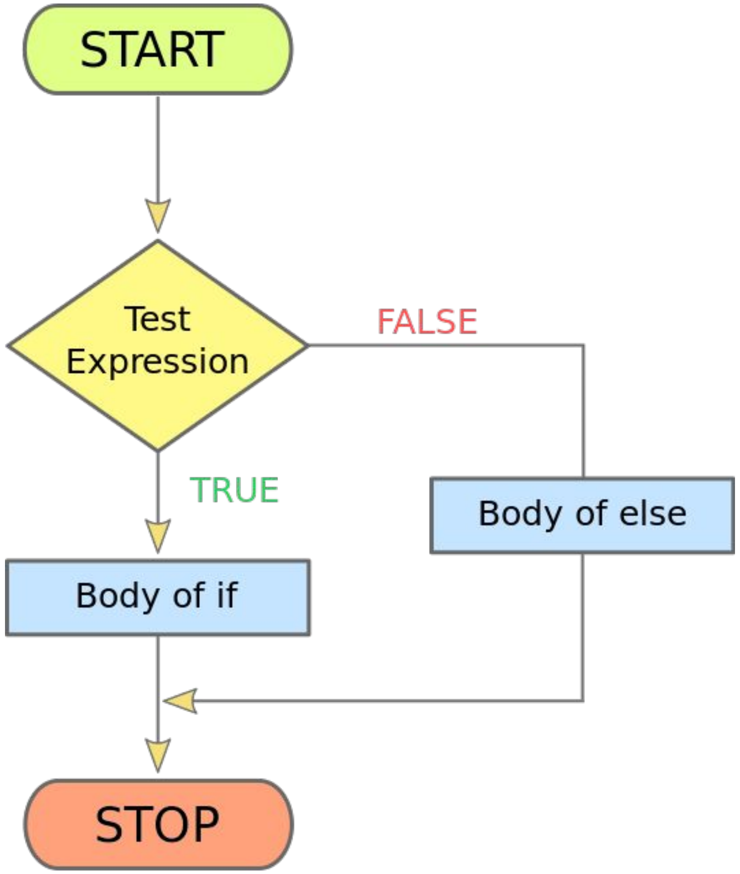
```
if(2==2){
    //i got executed
}
```



IF ELSE Statement

Syntax :

```
if(condition) {
    //if body
}
else {
    //else body
}
```



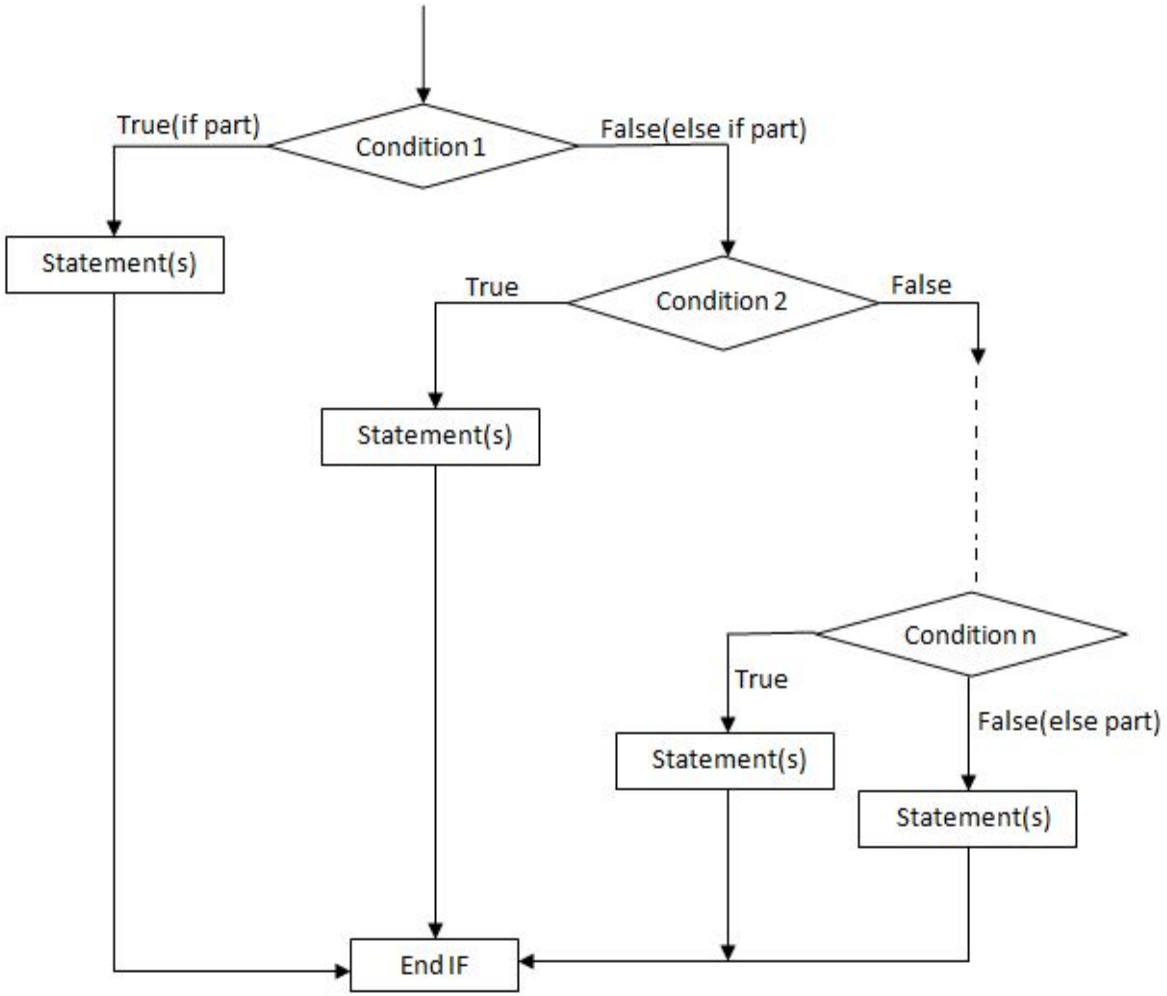
Eg 6: if else statement - Solidity

Find greatest among 2 numbers ?

IF ELSE IF.. ELSE Statement

Syntax :

```
if(condition1) {
    //condition1 body
}
else if(condition2) {
    //condition2 body
}
..
..
else{
    //else body
}
```

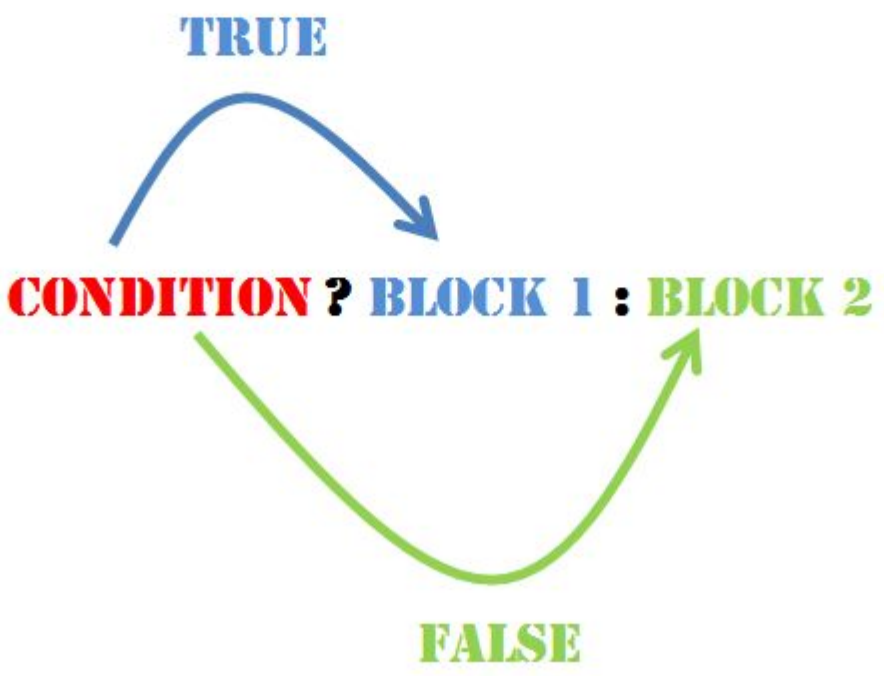


Eg 7: if else if .. else statement - Solidity

Find greatest among 3 numbers ?

Ternary Operator - Solidity

An expression a ? b : c evaluates to b if the value of a is true, and otherwise to c .



Eg 8: Ternary Operator

Find greatest among 2 numbers ?

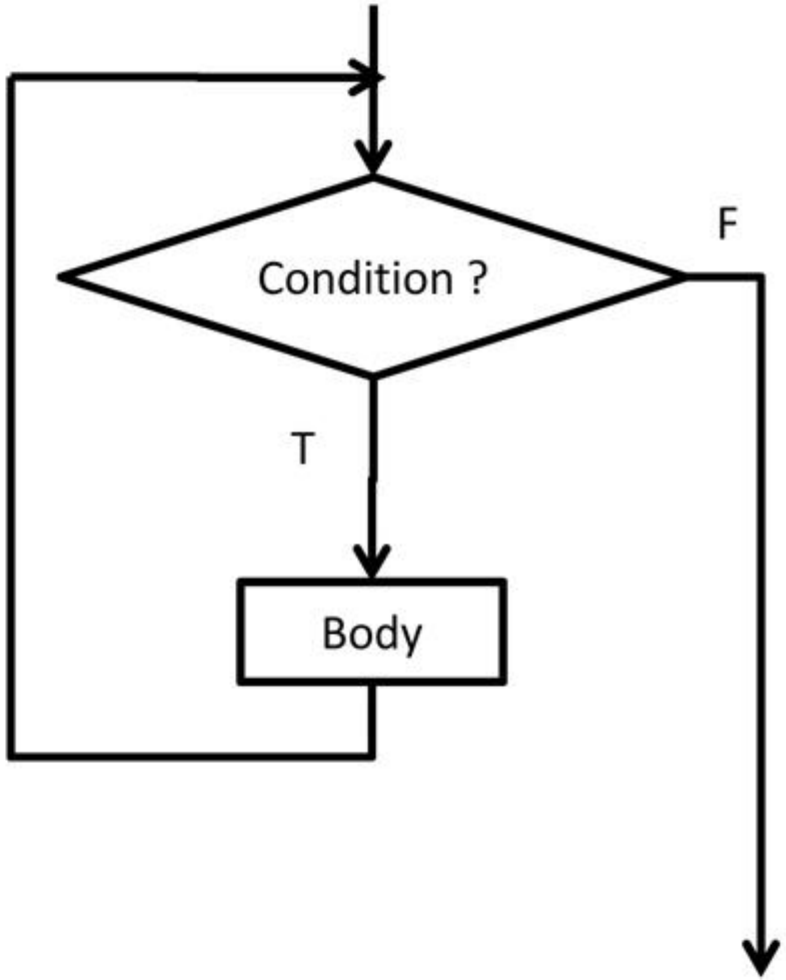
Looping Statements - Solidity

WHILE Loop

- ENTRY Controlled Loop.

  Syntax:

```
while(condition){
    //loop body
}
```



Eg 9: While

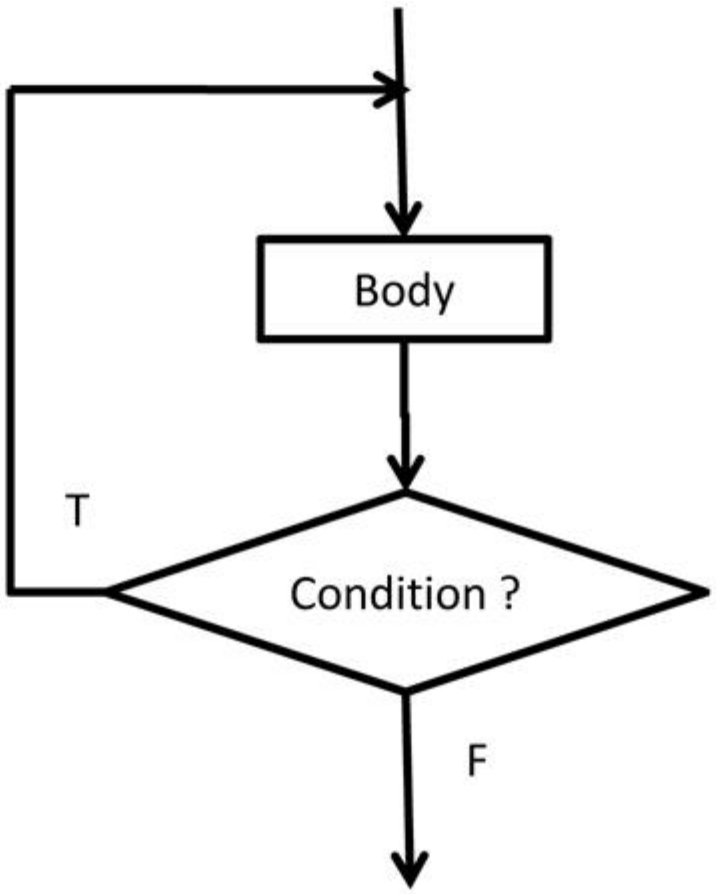
Find sum of digits ?

DO WHILE Loop

- EXIT Controlled Loop.
- Note : semi colon for while

Syntax:

```
do {
    //loop body
}
while(condition);
```



Eg 10: Do While

Find sum of digits ?

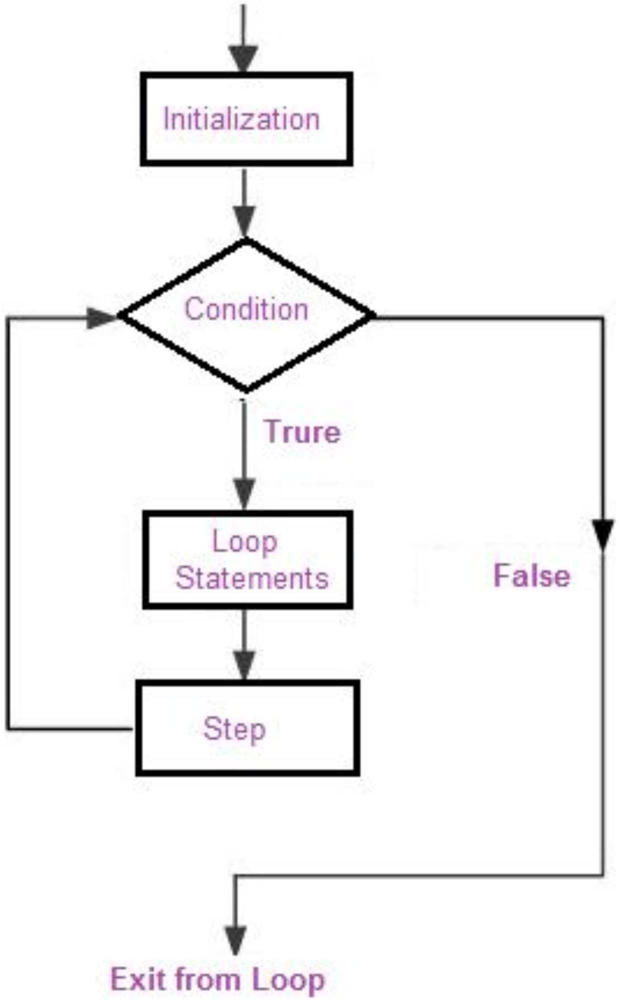
Looping Statements - Solidity

FOR Loop

- Special type of while
- Contains:
  - Initialization
  - Condition
  - Body of loop
  - value updation

Syntax:

```
for(initialization; condition; updation/steps){
    body of loop
}
```



Eg 11: For Loop

Find factorial of a number ?

Break Statement - Solidity

When break statement is encountered, the loop will be terminated

```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

Continue Statement - Solidity

When continue statement is encountered, the remaining portion in loop will be skipped.

```
while (test Expression)
{
      // codes
      if (condition for continue)
      {
         continue;
      }
      // codes
}
```

```
for (init, condition, update)
{
      // codes
      if (condition for continue)
      {
         continue;
      }
      // codes
}
```

Exercise 1

Create a smart contract to collect the details of multiple students and display them on demand ?

THANK YOU