Solidity

Solidity is the type and compiles language (when compiling its byte code execute in EVM)
It's compiler **solc** which convert solidity into byte code

Solidity is the **statically-typed**(not like the JS the type of the variable has to be mentioned like string, int, or bool) not the **dynamic**

High-Level Construct

1. Pragma
2. Comments
3. Import
4. Contract/library/interface

The library can be used by contract

Ethereum **Natural Specification(Natspec)** for comments for documentation

- /// for single line
- /**---*/ for multilines

Structure of Contract

1. State Varaibles
2. Function
3. Modifiers
4. Structure definition
5. Events
6. Enum

State variable are the global variable (these are store permanently in blockchain) also called storage variables

The Ethereum Virtual Machine has three areas where it can store items.

1. The first is "storage", where all the contract state variables reside. Every contract has its own storage and it is persistent between function calls and quite expensive to use.

2. The second is "memory", this is used to hold temporary values. It is erased between (external) function calls and is cheaper to use.Its defined inside the functions.
3. The third one is the stack, which is used to hold small local variables. It is almost free to use, but can only hold a limited amount of values

Qualifiers

1. Internal
2. Public
3. Private
4. Constant

Internal:
By default state variable is internal if nothing is specified (this is available inside the contract and childs of the contract).These varaibles can not be accessed from outside for modification but can be accessed for viewing.
Private:
Only accessible inside the contract not for childes and outside.
Public:
Accessable publically .Solidity compiler generates the getter function for each public state varaible
Constant:
Immutable .The value should be assigned at the time of declaration.Constant can be used with all qualifiers.

Internal and pivate variables can be access inside the contract and for outside access modifiers or function is needed.

DataTypes:

1. **Bool**
2. **String**
3. **Int/uint**
4. **Bytes**
5. **Address**
6. **Mapping**
7. **Enum**
8. **Struct**
9. **Array**

Unit ⇒ for possitive number

Uint8 ⇒ 8bit (0-255)

int8⇒ (-128 ==> +127)

Type	Minimum	Maximum
Int8	-128	127
Int16	-32,768	32,767
Int32	-2,147,483,648	2,147,483,647
Int64	- 9,223,372,036,854,775,808	9,223,372,036,854,775,807
Int	- 9,223,372,036,854,775,808	9,223,372,036,854,775,807
UInt8	0	255
UInt16	0	65,535
UInt32	0	4,294,967,295
UInt64	0	18,446,744,073,709,551,615
UInt	0	18,446,744,073,709,551,615

Functions:

Anonymous function which can be used to receive the ether if no other function is available which is aslo called fallback function.(only one unmamed function in contract)

Function Visibility Qualifiers

1. Public
2. Private
3. Internal
4. External ⇒ only accessible for outside the contract not inside the contract

Function Additional Qualifiers:

1. View
2. Pure
3. Payable

⇒ **view can not modify state variables but can return or access the state level variable**
You can not call function from inside if that other function is not in view.

```
function1 () public view;
function2() public view {
   function1();
}
```

This is because if view is not present then this function may update the state variable which is against the view rules(only accessible not modified)

⇒ **Pure can not access or modified the state level variables.**
You can not call function from inside if that other function is not **pure**

```
function1 () public pure;
function2() public pure {
   function1();
}
```

This is because if pure is not present then this function may access the state variable which is against the pure rules(not accessable)

⇒ payable is used for receiving ethereum.

Receiving Ethereum on Remix:
⇒ using **payable** qualifiers we can receive ether

```
function recievePayment () public payable{

        }

        function checkBalance () public view returns (uint){
           return address(this).balance;
        }
```

Receive Ethereum through ganache and test with truffle:
Contract file is

```
contract PaymentContract {

                function recievePayment () public payable {
                }
                function checkbalance() public view returns(uint) {
                    return address(this).balance;
                }
```

```
}
```

2_custom file is ⇒

```
const Second = artifacts.require("PaymentContract.sol");


module.exports = function (deployer) {
  deployer.deploy(Second);
};
```

Testfile is ⇒

```
uint public initialBalance = 10 ether;

    function testBalance() public{
        PaymentContract meta =
PaymentContract(DeployedAddresses.PaymentContract());
        meta.recievePayment.value(2 ether)();

        Assert.equal(meta.checkbalance(),2 ether,'Balance should be 2');
    }
```

Return from Functions:

Solidity provides two different syntaxes for returning data from function

1. Function mention datatype return without naming return variable
2. Function datatype and variable name in return

```
contract ReturnFunction {

  function testAge() public view returns(uint){
   //   uint  age = 50;
      return ;
  }

}
```

It will through error return argument is compulsory.if return is not used bydefault it returns 0 (if string ⇒ empty string and for bool ⇒ false will return)

2.Return with name

```
contract ReturnFunction {

  function testAge() public view returns(uint a){
      a = 50  ;
   //   return age;
  }

}
```

It will return a = 50; even if not used return keyword

```
function testAge() public view returns(uint a){
  a = 50  ;
 return 100;
}
```

⇒ it will return a 100

Function Return multiple values:

⇒ return(a,b,c) using tuple

```
function testAge() public view returns(int,bool){

  return (50,true);
 }
```

⇒ it will throw error if we dont used tuple ()

```
function testAge() public view returns(int,bool){

  return 50,true;
 }
```

If only dataTypes are passed but variables name are not initailzed then default values will return

```
function testAge() public view returns(int,bool){
  int a = 60;
  bool isFeePaid = true;

  // this will return default values ==> 0,false because varaible names are not
initalized with dataType
 }
```

Enumeration:

⇒its a custom user defined data type
⇒ defined at contract level

```
contract ReturnFunction {

  enum Gender {
      male,
      female
  }
   function testAge(Gender g) public view returns(Gender){
          Gender userData = g;
          return userData;
   }

}
```

Structure:
⇒ state leve; variable
⇒ custom user defined data type which contain multiple data type
⇒ composite data type
⇒ contains declare variables only (not the functions,code or expression) not the initialze variable
⇒ for **instance** no **new** keyword is required
⇒ struct can not be passed in function argument
⇒ struct can not be return from function

Modifiers:

⇒ modifier change the behaviour of function
⇒ modifier is like the function but you can not directly call a modifier like the function.
⇒ modifier call before the function call
⇒ modifier simple the code which provide the facility to add a filter or any validation outside the function

⇒ we can apply same authentication to multiple functions using modifier

⇒ address (datatype) represent ethereum address or external own account address

```
  contract First{

              modifier testAge(uint a){
                  if (a > 45){
                      _;
                  }
              }

              function checkAge(uint b) public view testAge(b) returns(string
memory){
                  return 'You can access the site.';
              }
              function checkAge2(uint c) public view testAge(c) returns(string
memory){
                  return 'You can access the site.';
              }

}
```

Events:

⇒ You can create the events in contract and fire the events but contract can not listen to the event itself.

⇒ you can find the information of events in the transaction blocks logs

⇒ events can listen in wallets ,webAPI etc

⇒ events are the part of the block

```
contract First {
    uint age;
    event userAge(uint,string );

    function testUser(uint a) public {
        age = a;
        emit userAge(age,'Hello World');
    }
}
```