SOLIDITY OVERVIEW

Solidity is a contract-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM)

Solidity is statically typed programming language

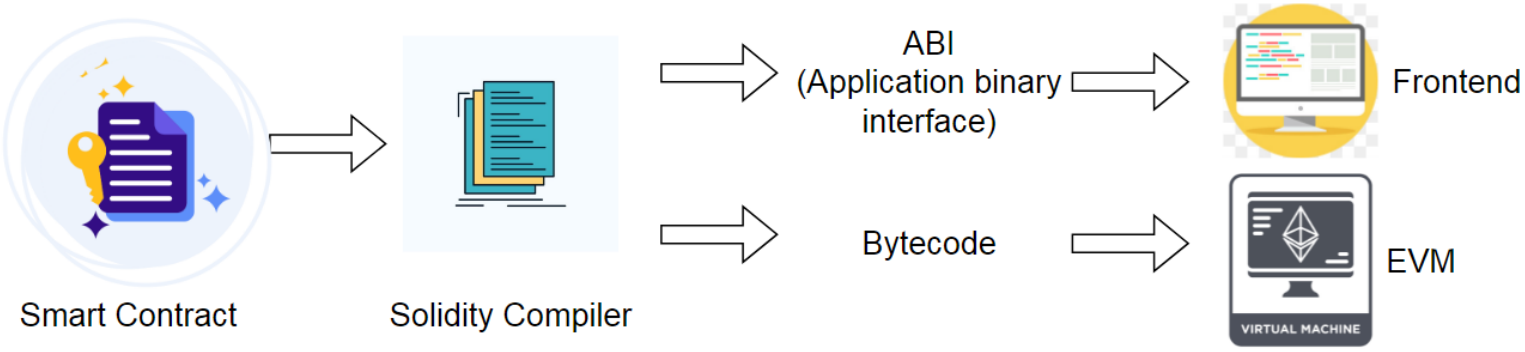
SMART CONTRACTS

Smart contracts are simply **programs stored on a blockchain that run when predetermined conditions are met**. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary's involvement or time loss.

ETHEREUM VIRTUAL MACHINE(EVM)

The Ethereum Virtual Machine is **the software platform that developers can use to create decentralized applications (DApps) on Ethereum**. This virtual machine is where all Ethereum accounts and smart contracts live.

SOLIDITY COMPILATION PROCESS



Before deploy smart contract, we need to compile solidity code to Bytecode for EVM(Ethereum Virtual Machine). You got two stuff after compiling solidity code:

- Bytecode/EVM code
- ABI(Application Binary Interface)

Bytecode is code EVM execute in blockchain network. ABI defines the actions you interact with smart contract. The actions in ABI mean the functions of smart contract.

PREREQUISITE

- Previous experience with any programming language like C, Python, or JavaScript.

Solidity Types

Value Types

Value type variables store their own data. These are the basic data types provided by solidity. These types of variables are always passed by value. The variables are copied wherever they are used in function arguments or assignment. Value type data types in solidity are listed below:

- **Boolean:** This data type accepts only two values True or False.
- **Integer:** This data type is used to store integer values, *int* and *uint* are used to declare *signed* and *unsigned integers* respectively.
- **Address:** Address hold a 20-byte value which represents the size of an Ethereum address. An address can be used to get balance or to transfer a balance by *balance* and *transfer* method respectively.
- **Bytes and Strings:** Bytes are used to store a fixed-sized character set while the string is used to store the character set equal to or more than a byte. The length of bytes is from 1 to 32, while the string has a dynamic length. Byte has an advantage that it uses less gas, so better to use when we know the length of data.
- **Enums:** It is used to create user-defined data types, used to assign a name to an integral constant which makes the contract more readable, maintainable, and less prone to errors. Options of enums can be represented by unsigned integer values starting from 0.

Type & Variables.sol

```sol
//SPDX-License-Identifier: Unlicensed

pragma solidity >=0.7.0;

contract Variable {
    uint videos = 30;
    int playlist = 3;
    bool active = true;
    bytes4 symbol = "web3";
    string name = "web3Mantra";
}
```

Int Range Formula:- positive 2^(n-1) - 1 negative 2^(n-1)

Uint Range Formula:- (2^n) -1

Solidity Variables

Solidity supports three types of variables.

State Variables − Variables whose values are permanently stored in a contract storage.

Local Variables − Variables whose values are present till function is executing.

Global Variables − Special variables exists in the global namespace used to get information about the blockchain.

Solidity is a statically typed language, which means that the state or local variable type needs to be specified during declaration. Each declared variable always have a default value based on its type. There is no concept of "undefined" or "null".

Block and Transaction Properties

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block when `blocknumber` is one of the 256 most recent blocks; otherwise returns zero
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas
- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Members of Address Types

`<address>.balance` (`uint256`)

balance of the Address in Wei

`<address>.code` (`bytes memory`)

code at the Address (can be empty)

`<address>.codehash` (`bytes32`)

the codehash of the Address

`<address payable>.transfer(uint256 amount)`

send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable

`<address payable>.send(uint256 amount) returns (bool)`

send given amount of Wei to Address, returns `false` on failure, forwards 2300 gas stipend, not adjustable

`<address payable>.send(uint256 amount) returns (bool)`

send given amount of Wei to Address, returns `false` on failure, forwards 2300 gas stipend, not adjustable

`<address>.call(bytes memory) returns (bool, bytes memory)`

issue low-level `CALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

`<address>.delegatecall(bytes memory) returns (bool, bytes memory)`

issue low-level `DELEGATECALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

`<address>.staticcall(bytes memory) returns (bool, bytes memory)`

issue low-level `STATICCALL` with the given payload, returns success condition and return data, forwards all available gas, adjustable

https://docs.soliditylang.org/en/v0.8.10/units-and-global-variables.html#special-variables-and-functions

Solidity Variable Scopes

Public − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.

Internal − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this.

Private − Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

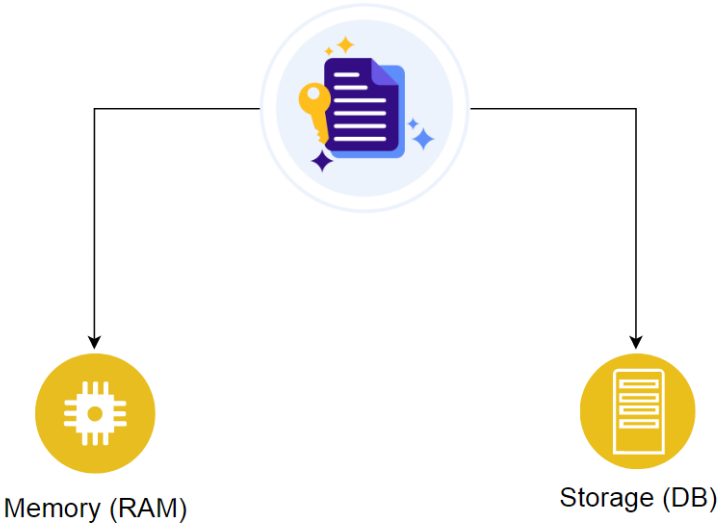
Solidity Functions

A function is a block of code that performs a task. it can be called and reused multiple times. You can pass information to a function and it can send information back

Functions.sol | Type & Variables.sol

```
1  // SPDX-License-Identifier: Unlicensed
2
3  pragma solidity >=0.7.0;
4
5  contract functions {
6      uint public val = 4;
7
8      function add() public pure returns(uint) {
9          // code
10         return 3 + 5;
11     }
12 }
13
14 // Pure ->> Pure function do not view and change state variables
15 // view ->> view function can only view state varibale cannot change them
16 // public --> generate a getter function to state variables
17 // returns --> use to specify the return data type of function
```

Storage v/s Memory



[Storage vs Memory in Solidity - GeeksforGeeks]

Storage and Memory keywords in Solidity are analogous to Computer's hard drive and Computer's RAM. Much like RAM, Memory in Solidity is a temporary place to store data whereas Storage holds data between function calls. The Solidity Smart Contract can use any amount of memory during the execution but once the execution stops, the Memory is completely wiped off for the next execution. Whereas Storage on the other hand is persistent, each execution of the Smart contract has access to the data previously stored on the storage area.

Every transaction on Ethereum Virtual Machine costs us some amount of Gas. The lower the Gas consumption the better is your Solidity code. The Gas consumption of Memory is not very significant as compared to the gas consumption of Storage. Therefore, it is always better to use Memory for intermediate calculations and store the final result in Storage.

1. State variables and Local Variables of structs, array are always stored in storage by default.
2. Function arguments are in memory.
3. Whenever a new instance of an array is created using the keyword 'memory', a new copy of that variable is created. Changing the array value of the new instance does not affect the original array.

Solidity Operators

In any programming language, operators play a vital role i.e. they create a foundation for the programming. Similarly, the functionality of Solidity is also incomplete without the use of operators. Operators allow users to perform different operations on operands. Solidity supports the following types of operators based upon their functionality.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment operators
6. Conditional Operator

Arithmetic Operators

These operators are used to perform arithmetic or mathematical operations. Solidity supports the following arithmetic operators :

Operator	Denotation	Description
Addition	+	Used to add two operands
Subtraction	-	Used to subtract the second operand from first
Multiplication	*	Used to multiply both operands
Division	/	Used to divide numerator by denominator
Modulus	%	Gives the remainder after integer division
Increment	++	Increases the integer value by one
Decrement	--	Decreases the integer value by one

Relational Operators

These operators are used to compare two values. Solidity supports the following relational operators :

Operator	Denotation	Description
Equal	==	Checks if two values are equal or not, returns true if equals, and vice-versa
Not Equal	!=	Checks if two values are equal or not, returns true if not equals, and vice-versa
Greater than	>	Checks if left value is greater than right or not, returns true if greater, and vice-versa
Less than	<	Checks if left value is less than right or not, returns true if less, and vice-versa
Greater than or Equal to	>=	Checks if left value is greater and equal than right or not, returns true if greater and equal, and vice-versa
Less than or Equal to	<=	Checks if left value is less than right or not, returns true if less and equals, and vice-versa

Logical Operators

These operators are used to combine two or more conditions. Solidity supports the following arithmetic operators :

Operator	Denotation	Description
Logical AND	&&	Returns true if both conditions are true and false if one or both conditions are false
Logical OR	||	Returns true if one or both conditions are true and false when both are false
Logical NOT	!	Returns true if the condition is not satisfied else false

1. The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
5. The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
6. The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it.

Assignment Operator

These operators are for the assignment of value to a variable. The operand at the left side is variable while operand at the right side is value. Solidity supports the following arithmetic operators :

Operator	Denotation	Description
Simple Assignment	=	Simply assigns the value at the right side to the operand at the left side
Add Assignment	+=	Adds operand at the right side to operand at the left side and assigns the value to left operand
Subtract Assignment	-=	Subtracts operand at the right side from operand at the left side and assigns the value to left operand
Multiply Assignment	*=	Multiplies both the operands and assign the value to left operand
Divide Assignment	/=	Divides operand at the left side by operand at the right side and assign the value to left operand
Modulus Assignment	%=	Divides operand at the left side by operand at the right side and assign the remainder to left operand

Conditional Operators

It is a ternary operator that evaluates the expression first then checks the condition for return values corresponding to true or false.

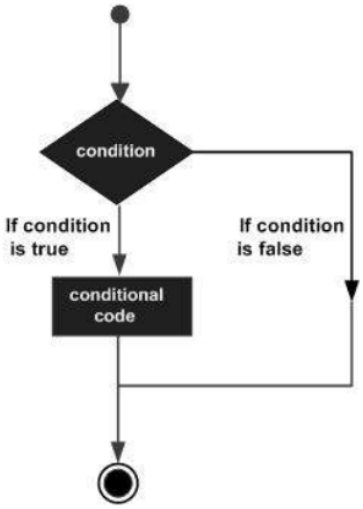
Syntax:

```
if condition true ? then A: else B
```

DECISION MAKING

Flow Chart of if-else

The following flow chart shows how the if-else statement works.



Solidity supports the following forms of **if..else** statement −

Sr.No	Statements & Description
1	if statement The if statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally.
2	if...else statement The 'if...else' statement is the next form of control statement that allows Solidity to execute statements in a more controlled way.
3	if...else if... statement. The if...else if... statement is an advanced form of if...else that allows Solidity to make a correct decision out of several conditions.

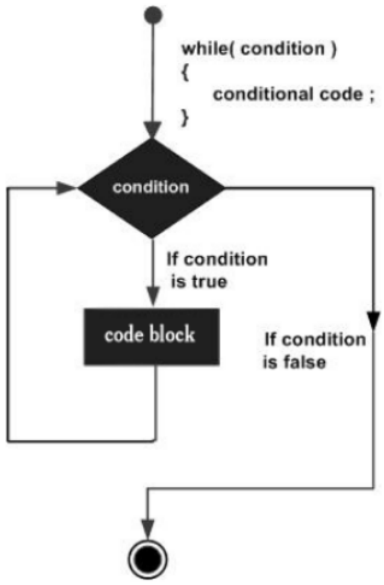
Loops

while loop

The most basic loop in Solidity is the **while** loop which would be discussed in this chapter. 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.

Flow Chart

The flow chart of **while loop** looks as follows −



Syntax

The syntax of **while loop** in Solidity is as follows −

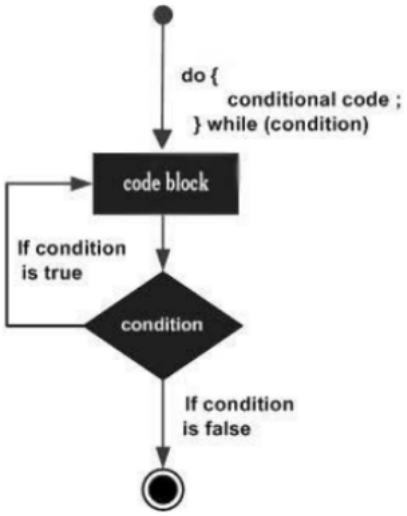
```
while (expression) {
   Statement(s) to be executed if expression is true
}
```

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**.

Flow Chart

The flow chart of a **do-while** loop would be as follows −



Syntax

The syntax for **do-while** loop in Solidity is as follows −

```
do {
   Statement(s) to be executed;
} while (expression);
```

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.

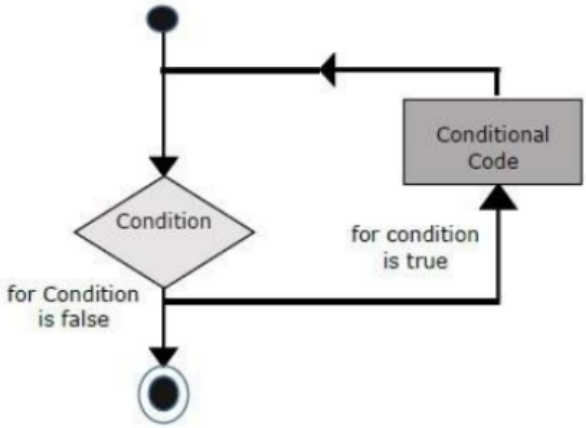
The **test statement** which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.

The **iteration statement** where you can increase or decrease your counter.

You can put all the three parts in a single line separated by semicolons.

Flow Chart

The flow chart of a **for** loop in Solidity would be as follows −



Syntax

The syntax of **for** loop is Solidity is as follows −

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

Solidity Arrays

Arrays are data structures that store the fixed collection of elements of the same data types in which each and every element has a specific location called index. Instead of creating numerous individual variables of the same type, we just declare one array of the required size and store the elements in the array and can be accessed using the index. In Solidity, an array can be of fixed size or dynamic size.

Declaring Arrays

To declare an array of fixed size in Solidity, the programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type[arraySize] arrayName;
```

Fixed-size Arrays

The size of the array should be predefined. The total number of elements should not exceed the size of the array. If the size of the array is not specified then the array of enough size is created which is enough to hold the initialization.

Dynamic Array:

The size of the array is not predefined when it is declared. As the elements are added the size of array changes and at the runtime, the size of the array will be determined.

Array Operations

1. Accessing Array Elements: The elements of the array are accessed by using the index.

2. Length of Array: Length of the array is used to check the number of elements present in an array. The size of the memory array is fixed when they are declared, while in case the dynamic array is defined at runtime so for manipulation length is required.

3. Push: Push is used when a new element is to be added in a dynamic array. The new element is always added at the last position of the array.

4. Pop: Pop is used when the last element of the array is to be removed in any dynamic array.

Solidity Struct

Struct types are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

Title
Author
Subject
Book ID

Defining a Struct

To define a Struct, you must use the **struct** keyword. The struct keyword defines a new data type, with more than one member. The format of the struct statement is as follows −

```
struct struct_name {
   type1 type_name_1;
   type2 type_name_2;
   type3 type_name_3;
}
```

Example

```
struct Book {
   string title;
   string author;
   uint book_id;
}
```

Example

Try the following code to understand how the structs works in Solidity.

```
pragma solidity ^0.5.0;

contract test {
   struct Book {
      string title;
      string author;
      uint book_id;
   }
   Book book;

   function setBook() public {
      book = Book('Learn Java', 'TP', 1);
   }
   function getBookId() public view returns (uint) {
      return book.book_id;
   }
}
```

Solidity Enums

Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

Example

Try the following code to understand how the enum works in Solidity.

```
pragma solidity ^0.5.0;

contract test {
   enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
   FreshJuiceSize choice;
   FreshJuiceSize constant defaultChoice = FreshJuiceSize.MEDIUM;

   function setLarge() public {
      choice = FreshJuiceSize.LARGE;
   }
   function getChoice() public view returns (FreshJuiceSize) {
      return choice;
   }
   function getDefaultChoice() public pure returns (uint) {
      return uint(defaultChoice);
   }
}
```

Solidity Mapping

Mapping is a reference type as arrays and structs. Following is the syntax to declare a mapping type.

```
mapping(_KeyType => _ValueType)
```

Where

_KeyType − can be any built-in types plus bytes and string. No reference type or complex objects are allowed.

_ValueType − can be any type.

Considerations

Mapping can only have type of **storage** and are generally used for state variables.

Mapping can be marked public. Solidity automatically create getter for it.

Example

Try the following code to understand how the mapping type works in Solidity.

```
pragma solidity ^0.5.0;

contract LedgerBalance {
   mapping(address => uint) public balances;

   function updateBalance(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}
```

CONSTRUCTOR

Constructor is a special function declared using **constructor** keyword. It is an optional funtion and is used to initialize state variables of a contract. Following are the key characteristics of a constructor.

A contract can have only one constructor.

A constructor code is executed once when a contract is created and it is used to initialize contract state.

After a constructor code executed, the final code is deployed to blockchain. This code include public functions and code reachable through public functions. Constructor code or any internal method used only by constructor are not included in final code.

A constructor can be either public or internal.

A internal constructor marks the contract as abstract.

In case, no constructor is defined, a default constructor is present in the contract.

```
pragma solidity ^0.5.0;

contract Test {
   constructor() {}
}
```

In case, base contract have constructor with arguments, each derived contract have to pass them.

Base constructor can be initialized directly using following way −

```
pragma solidity ^0.5.0;

contract Base {
   uint data;
   constructor(uint _data) {
      data = _data;
   }
}
contract Derived is Base (5) {
   constructor() {}
}
```

Base constructor can be initialized indirectly using following way −

```
pragma solidity ^0.5.0;

contract Base {
   uint data;
   constructor(uint _data) {
      data = _data;
   }
}
contract Derived is Base {
   constructor(uint _info) Base(_info * _info) {}
}
```

Solidity Events

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

An event can be declared using event keyword.

```
//Declare an Event
event Deposit(address indexed _from, bytes32 indexed _id, uint _value);

//Emit an event
emit Deposit(msg.sender, _id, msg.value);
```

Example

Try the following code to understand how an event works in Solidity.

First Create a contract and emit an event.

```
pragma solidity ^0.5.0;

contract Test {
   event Deposit(address indexed _from, bytes32 indexed _id, uint _value);
   function deposit(bytes32 _id) public payable {
      emit Deposit(msg.sender, _id, msg.value);
   }
}
```

Payable

In **Solidity**, we can use the keyword `payable` to specify that an address or a function can receive Ether.

```
pragma solidity >=0.7.0;

contract payables {
    address payable public owner;

    constructor() {
        owner = payable(msg.sender);
    }

    function transferEth() payable public {
        owner.transfer(msg.value);
    }
}
```

Error Handling

Solidity provides various functions for error handling. Generally when an error occurs, the state is reverted back to its original state. Other checks are to prevent unauthorized code access. Following are some of the important methods used in error handling −

assert(bool condition) − In case condition is not met, this method call causes an invalid opcode and any changes done to state got reverted. This method is to be used for internal errors.

require(bool condition) − In case condition is not met, this method call reverts to original state. - This method is to be used for errors in inputs or external components.

require(bool condition, string memory message) − In case condition is not met, this method call reverts to original state. - This method is to be used for errors in inputs or external components. It provides an option to provide a custom message.

revert() − This method aborts the execution and revert any changes done to the state.

revert(string memory reason) − This method aborts the execution and revert any changes done to the state. It provides an option to provide a custom message.

To handle errors, Solidity **undoes changes** that might have caused issues.

`assert` checks for **internal** errors. `require` analyzes conditions.

Exceptions that Solidity `revert` generates can contain **error strings**.

Function Modifiers

Function Modifiers are used to modify the behaviour of a function. For example to add a prerequisite to a function.

First we create a modifier with or without parameter.

```
contract Owner {
  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }
   function changePrice(uint _price) public onlyOwner {

       price = _price;
    }


}
```

Units

In solidity we can use wei, finney, szabo or ether as a suffix to a literal to be used to convert various ether based denominations. Lowest unit is wei and 1e12 represents 1 x 10^{12}.

```
assert(1 wei == 1);
assert(1 szabo == 1e12);
assert(1 finney == 1e15);
assert(1 ether == 1e18);
assert(2 ether == 2000 fenny);
```

Time Units

Similar to currency, Solidity has time units where lowest unit is second and we can use seconds, minutes, hours, days and weeks as suffix to denote time.

```
assert(1 seconds == 1);
assert(1 minutes == 60 seconds);
assert(1 hours == 60 minutes);
assert(1 day == 24 hours);
assert(1 week == 7 days);
```

Cryptographic functions

A cryptographic hash function is an algorithm that takes an arbitrary amount of data as input and produces the enciphered text of fixed size. Even a slight change in the input gives a completely different output.

Solidity provides the following cryptographic functions:

Function	Properties
keccak256(bytes memory) returns (bytes32)	Computes the Keccak-256 hash of the input
sha256(bytes memory) returns (bytes32)	Computes the SHA-256 hash of the input

ripemd160(bytes memory) returns (bytes20)	Compute RIPEMD-160 hash of the input
sha256(bytes memory) returns (bytes32)	Computes the SHA-256 hash of the input
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)	Recover the address associated with the public key from Elliptic curve signature used for cryptography or return Zero if an error occurs. The parameters correspond to ECDSA Signature values.

Inheritance

Inheritance is a way to extend functionality of a contract. Solidity supports both single as well as multiple inheritance. Following are the key highlighsts.

A derived contract can access all non-private members including internal methods and state variables. But using this is not allowed.

Function overriding is allowed provided function signature remains same. In case of difference of output parameters, compilation will fail.

We can call a super contract's function using super keyword or using super contract name.

In case of multiple inheritance, function call using super gives preference to most derived contract.

Example

```
pragma solidity ^0.5.0;

contract C {
   //private state variable
   uint private data;

   //public state variable
   uint public info;

   //constructor
   constructor() public {
      info = 10;
   }
   //private function
   function increment(uint a) private pure returns(uint) { return a + 1; }

   //public function
   function updateData(uint a) public { data = a; }
   function getData() public view returns(uint) { return data; }
   function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}
//Derived Contract
contract E is C {
   uint private result;
   C private c;
   constructor() public {
      c = new C();
   }
   function getComputedResult() public {
      result = compute(3, 5);
   }
   function getResult() public view returns(uint) { return result; }
   function getData() public view returns(uint) { return c.info(); }
}
```

Function Visibility

public

Finally, a public function can be called from all potential parties.

Unless otherwise specified, all functions are made public by default.

Private

A private function is one that can only be called by the main contract itself. Although it's not the default, it is generally good practice to keep your functions private unless a scope with more visibility is needed.

Internal

An internal function can be called by the main contract itself, plus any derived contracts. As with private functions, it's generally a good idea to keep your functions internal wherever possible.

External

An external function can only be called from a third party. It cannot be called from the main contract itself or any contracts derived from it. External functions have the benefit that they can be more performant due to the fact that their arguments do not need to be copied to memory. So, where possible, it's advisable to keep logic that only needs to be accessed by an external party to an external function.

Library

Libraries are similar to Contracts but are mainly intended for reuse. A Library contains functions which other contracts can call. Solidity have certain restrictions on use of a Library. Following are the key characteristics of a Solidity Library.

- Library functions can be called directly if they do not modify the state. That means pure or view functions only can be called from outside the library.
- Library can not be destroyed as it is assumed to be stateless.
- A Library cannot have state variables.
- A Library cannot inherit any element.
- A Library cannot be inherited.

Example

Try the following code to understand how a Library works in Solidity.

```
pragma solidity ^0.5.0;

library Search {
   function add(uint a, uint b) public pure returns (uint) {
      return a + b;
   }
}
contract Test {
   function getAddition() view returns(uint){
      uint result = Search.add(4, 5);
      return result;
   }
}
```

Interfaces

Interfaces are similar to abstract contracts and are created using **interface** keyword. Following are the key characteristics of an interface.

Interface can not have any function with implementation.

Functions of an interface can be only of type external.

Interface can not have constructor.

Interface can not have state variables.

Interface can have enum, structs which can be accessed using interface name dot notation.

Example

Try the following code to understand how the interface works in Solidity.

```
pragma solidity ^0.5.0;

interface Calculator {
   function getResult() external view returns(uint);
}
contract Test is Calculator {
   constructor() public {}
   function getResult() external view returns(uint){
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
```

Abstract Contracts

Abstract Contract is one which contains at least one function without any implementation. Such a contract is used as a base contract. Generally an abstract contract contains both implemented as well as abstract functions. Derived contract will implement the abstract function and use the existing functions as and when required.

In case, a derived contract is not implementing the abstract function then this derived contract will be marked as abstract.

Example

Try the following code to understand how the abstract contracts works in Solidity.

```
pragma solidity ^0.5.0;

contract Calculator {
   function getResult() public view returns(uint);
}
contract Test is Calculator {
   function getResult() public view returns(uint) {
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

Virtual & Override

Function Overriding

A function that allows an inheriting contract to override its behavior will be marked at `virtual`. The function that overrides that base function should be marked as `override`.

```
pragma solidity >=0.5.0 <0.7.0;

contract Base1
{
function foo() virtual public {}
}

contract Base2
{
function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
// Derives from multiple bases defining foo(), so we must
explicitly
// override it
function foo() public override(Base1, Base2) {}
}
```