Smart Contract and Solidity Programming

A Smart Contract (or cryptocontract) is a computer program that directly and automatically controls the transfer of digital assets between the parties under certain conditions. A smart contract works in the same way as a traditional contract while also automatically enforcing the contract. Smart contracts are programs that execute exactly as they are set up(coded, programmed) by their creators. Just like a traditional contract is enforceable by law, smart contracts are enforceable by code.

- The bitcoin network was the first to use some sort of smart contract by using them to transfer value from one person to another.
- The smart contract involved employs basic conditions like checking if the amount of value to transfer is actually available in the sender account.
- Later, the Ethereum platform emerged which was considered more powerful, precisely because the developers/programmers could make custom contracts in a Turing-complete language.
- It is to be noted that the contracts written in the case of the bitcoin network were written in a Turing-incomplete language, restricting the potential of smart contracts implementation in the bitcoin network.
- There are some common smart contract platforms like Ethereum, Solana, Polkadot, Hyperledger fabric, etc.

History:

In 1994, **Nick Szabo**, a legal scholar, and a cryptographer recognized the application of a decentralized ledger for smart contracts. He theorized that these contracts could be written in code which can be stored and replicated on the system and supervised by the network of computers that constitute the blockchain. These smart contracts could also help in transferring digital assets between the parties under certain conditions.

Features of Smart Contracts

The following are some essential characteristics of a smart contract:

1. **Distributed:** Everyone on the network is guaranteed to have a copy of all the conditions of the smart contract and they cannot be changed by one of the parties. A smart contract is replicated and distributed by all the nodes connected to the network.
2. **Deterministic:** Smart contracts can only perform functions for which they are designed only when the required conditions are met. The final outcome will not vary, no matter who executes the smart contract.
3. **Immutable:** Once deployed smart contract cannot be changed, it can only be removed as long as the functionality is implemented previously.
4. **Autonomy:** There is no third party involved. The contract is made by you and shared between the parties. No intermediaries are involved which minimizes bullying and grants full authority to the dealing parties. Also, the smart contract is maintained and executed by all the nodes on the network, thus removing all the controlling power from any one party's hand.
5. **Customizable:** Smart contracts have the ability for modification or we can say customization before being launched to do what the user wants it to do.
6. **Transparent:** Smart contracts are always stored on a public distributed ledger called blockchain due to which the code is visible to everyone, whether or not they are participants in the smart contract.
7. **Trustless:** These are not required by third parties to verify the integrity of the process or to check whether the required conditions are met.

8. **Self-verifying:** These are self-verifying due to automated possibilities.
9. **Self-enforcing:** These are self-enforcing when the conditions and rules are met at all stages.

Capabilities of Smart Contracts

1. **Accuracy:** Smart contracts are accurate to the limit a programmer has accurately coded them for execution.
2. **Automation:** Smart contracts can automate the tasks/ processes that are done manually.
3. **Speed:** Smart contracts use software code to automate tasks, thereby reducing the time it takes to maneuver through all the human interaction-related processes. Because everything is coded, the time taken to do all the work is the time taken for the code in the smart contract to execute.
4. **Backup:** Every node in the blockchain maintains the shared ledger, providing probably the best backup facility.
5. **Security:** Cryptography can make sure that the assets are safe and sound. Even if someone breaks the encryption, the hacker will have to modify all the blocks that come after the block which has been modified. Please note that this is a highly difficult and computation-intensive task and is practically impossible for a small or medium-sized organization to do.
6. **Savings:** Smart contracts save money as they eliminate the presence of intermediaries in the process. Also, the money spent on the paperwork is minimal to zero.
7. **Manages information:** Smart contract manages users' agreement, and stores information about an application like domain registration, membership records, etc.
8. **Multi-signature accounts:** Smart contracts support multi-signature accounts to distribute funds as soon as all the parties involved confirm the agreement.

How Do Smart Contracts Work?

A smart contract is just a digital contract with the security coding of the blockchain.

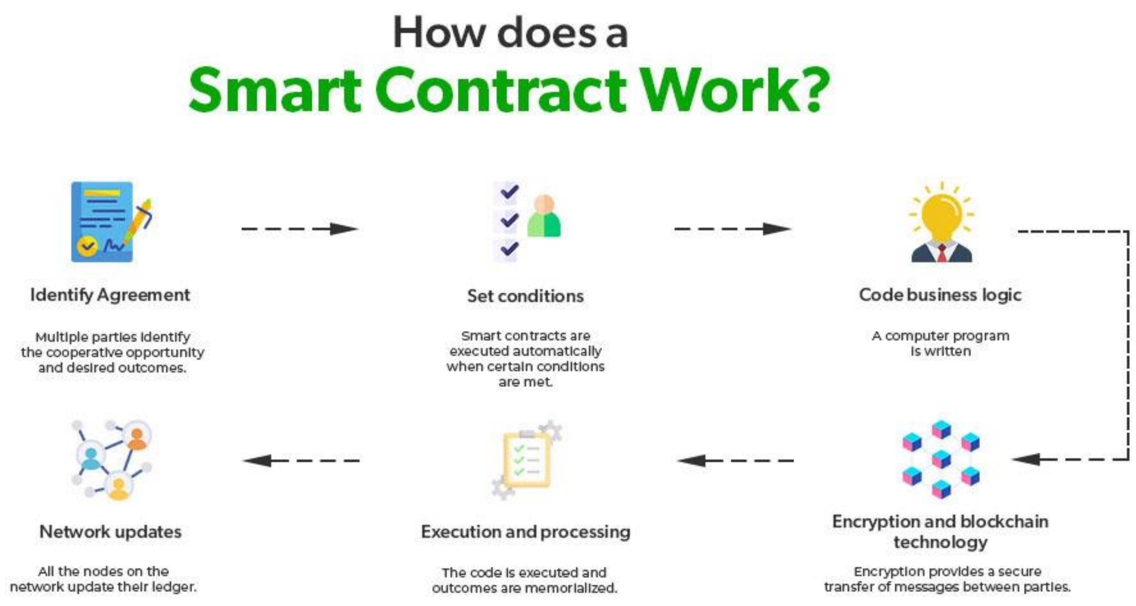
- It has details and permissions written in code that require an exact sequence of events to take place to trigger the agreement of the terms mentioned in the smart contract.
- It can also include the time constraints that can introduce deadlines in the contract.
- Every smart contract has its address in the blockchain. The contract can be interacted with by using its address presuming the contract has been broadcasted on the network.

The idea behind smart contracts is pretty simple. They are executed on a basis of simple logic, IF-THEN for example:

- **IF** you send object A, **THEN** the sum (of money, in cryptocurrency) will be transferred to you.
- **IF** you transfer a certain amount of digital assets (cryptocurrency, for example, ether, bitcoin), **THEN** the A object will be transferred to you.
- **IF** I finish the work, **THEN** the digital assets mentioned in the contract will be transferred to me.

Note: The WHEN constraint can be added to include the time factor in the smart contracts.
It can be seen that these smart contracts help set conditions that have to be fulfilled for the terms of the contract agreement to be executed. There is no limit on how much IF or THEN you can include in your intelligent contract.

Smart Contract Working


How does a
Smart Contract Work?
Identify Agreement
Multiple parties identify the cooperative opportunity and desired outcomes.
Set conditions
Smart contracts are executed automatically when certain conditions are met.
Code business logic
A computer program is written
Encryption and blockchain technology
Encryption provides a secure transfer of messages between parties.
Execution and processing
The code is executed and outcomes are memorialized.
Network updates
All the nodes on the network update their ledger.

- **Identify Agreement:** Multiple parties identify the cooperative opportunity and desired outcomes and agreements could include business processes, asset swaps, etc.
- **Set conditions:** Smart contracts could be initiated by parties themselves or when certain conditions are met like financial market indices, events like GPS locations, etc.
- **Code business logic:** A computer program is written that will be executed automatically when the conditional parameters are met.
- **Encryption and blockchain technology:** Encryption provides secure authentication and transfer of messages between parties relating to smart contracts.
- **Execution and processing:** In blockchain iteration, whenever consensus is reached between the parties regarding authentication and verification then the code is executed and the outcomes are memorialized for compliance and verification.
- **Network updates:** After smart contracts are executed, all the nodes on the network update their ledger to reflect the new state. Once the record is posted and verified on the blockchain network, it cannot be modified, it is in append mode only.

Applications of Smart Contracts

1. **Real Estate:** Reduce money paid to the middleman and distribute between the parties actually involved. For example, a smart contract to transfer ownership of an apartment once a certain amount of resources have been transferred to the seller's account(or wallet).
2. **Vehicle ownership:** A smart contract can be deployed in a blockchain that keeps track of vehicle maintenance and ownership. The smart contract can, for example, enforce vehicle maintenance service every six months; failure of which will lead to suspension of driving license.
3. **Music Industry:** The music industry could record the ownership of music in a blockchain. A smart contract can be embedded in the blockchain and royalties can be

credited to the owner's account when the song is used for commercial purposes. It can also work in resolving ownership disputes.

4. **Government elections:** Once the votes are logged in the blockchain, it would be very hard to decrypt the voter address and modify the vote leading to more confidence against the ill practices.
5. **Management:** The blockchain application in management can streamline and automate many decisions that are taken late or deferred. Every decision is transparent and available to any party who has the authority (an application on the private blockchain). For example, a smart contract can be deployed to trigger the supply of raw materials when 10 tonnes of plastic bags are produced.
6. **Healthcare:** Automating healthcare payment processes using smart contracts can prevent fraud. Every treatment is registered on the ledger and in the end, the smart contract can calculate the sum of all the transactions. The patient can't be discharged from the hospital until the bill has been paid and can be coded in the smart contract.

Example Use cases:

1. Smart contracts provide utility to other contracts. For example, consider a smart contract that transfers funds to party A after 10 days. After 10 days, the above-mentioned smart contract will execute another smart contract which checks if the required funds are available at the source account (let's say party B).
2. They facilitate the implementation of 'multi-signature' accounts, in which the assets are transferred only when a certain percentage of people agree to do so
3. Smart contracts can map legal obligations into an automated process.
4. If smart contracts are implemented correctly, can provide a greater degree of contractual security.

Advantages of Smart Contracts

1. **Recordkeeping:** All contract transactions are stored in chronological order in the blockchain and can be accessed along with the complete audit trail. However, the parties involved can be secured cryptographically for full privacy.
2. **Autonomy:** There are direct dealings between parties. Smart contracts remove the need for intermediaries and allow for transparent, direct relationships with customers.
3. **Reduce fraud:** Fraudulent activity detection and reduction. Smart contracts are stored in the blockchain. Forcefully modifying the blockchain is very difficult as it's computation-intensive. Also, a violation of the smart contract can be detected by the nodes in the network and such a violation attempt is marked invalid and not stored in the blockchain.
4. **Fault-tolerance:** Since no single person or entity is in control of the digital assets, one-party domination and situation of one part backing out do not happen as the platform is decentralized and so even if one node detaches itself from the network, the contract remains intact.
5. **Enhanced trust:** Business agreements are automatically executed and enforced. Plus, these agreements are immutable and therefore unbreakable and undeniable.
6. **Cost-efficiency:** The application of smart contracts eliminates the need for intermediaries (brokers, lawyers, notaries, witnesses, etc.) leading to reduced costs. Also eliminates paperwork leading to paper saving and money-saving.

Challenges of Smart Contracts

1. **No regulations:** A lack of international regulations focusing on blockchain technology (and related technology like smart contracts, mining, and use cases like cryptocurrency) makes these technologies difficult to oversee.
2. **Difficult to implement:** Smart contracts are also complicated to implement because it's still a relatively new concept and research is still going on to understand the smart contract and its implications fully.
3. **Immutable:** They are practically immutable. Whenever there is a change that has to be incorporated into the contract, a new contract has to be made and implemented in the blockchain.
4. **Alignment:** Smart contracts can speed the execution of the process that span multiple parties irrespective of the fact whether the smart contracts are in alignment with all the parties' intention and understanding.

Solidity

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, supports inheritance, libraries and complex user-defined types programming language.

Solidity is used to create contracts for system such as voting, crowdfunding, blind auctions, and multi-signature wallets.

Ethereum

Ethereum is a decentralized ie. blockchain platform that runs smart contracts i.e. applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference.

Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine, also known as EVM, is the runtime environment for smart contracts in Ethereum. The Ethereum Virtual Machine focuses on providing security and executing untrusted code by computers all over the world.

The EVM specialised in preventing Denial-of-service attacks and ensures that programs do not have access to each other's state, ensuring communication can be established without any potential interference.

The Ethereum Virtual Machine has been designed to serve as a runtime environment for smart contracts based on Ethereum.

Smart Contract

A smart contract is a computer protocol intended to digitally facilitate, verify, or enforce the negotiation or performance of a contract. Smart contracts allow the performance of credible transactions without third parties. These transactions are trackable and irreversible.

The concept of smart contracts was first proposed by Nick Szabo in 1994. Szabo is a legal scholar and cryptographer known for laying the groundwork for digital currency.

This chapter explains how we can setup Solidity compiler on CentOS machine. If you do not have a Linux machine then you can use our Online Compiler for small contracts and for quickly learning Solidity.

A Solidity source files can contain an any number of contract definitions, import directives and pragma directives.

Following is an example of a Solidity file

```
pragma solidity >=0.4.0 <0.6.0;
contract SimpleStorage {
   uint storedData;
   function set(uint x) public {
      storedData = x;
   }
   function get() public view returns (uint) {
      return storedData;
   }
}
```

Pragma

The first line is a pragma directive which tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality. A pragma directive is always local to a source file and if you import another file, the pragma from that file will not automatically apply to the importing file.

So a pragma for a file which will not compile earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0 will be written as follows −

```
pragma solidity ^0.4.0;
```

Here the second condition is added by using ^.

Contract

A Solidity contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereumblockchain.

The line uintstoredData declares a state variable called storedData of type uint and the functions set and get can be used to modify or retrieve the value of the variable.

Reserved Keywords

Following are the reserved keywords in Solidity −

abstract	after	alias	apply
auto	case	catch	copyof
default	define	final	immutable
implements	in	inline	let

macro	match	mutable	null
of	override	partial	promise
reference	relocatable	sealed	sizeof
static	supports	switch	try
typedef	typeof	unchecked	

```
pragma solidity ^0.5.0;
contract SolidityTest {
   constructor() public{
   }
   function getResult() public view returns(uint){
      uint a = 1;
      uint b = 2;
      uint result = a + b;
      return result;
   }
}
```

Step 1 − Copy the given code in Remix IDE Code Section.

Step 2 − Under Compile Tab, click **Start to Compile** button.

Step 3 − Under Run Tab, click **Deploy** button.

Step 4 − Under Run Tab, Select **SolidityTest at 0x...** in drop-down.

Step 5 − Click **getResult** Button to display the result.

Data Types

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Solidity offers the programmer built-in as well as user defined data types.

Type	Keyword	Values

Boolean	bool	true/false
Integer	int/uint	Signed and unsigned integers of varying sizes.
Integer	int8 to int256	Signed int from 8 bits to 256 bits. int256 is the same as int.
Integer	uint8 to uint256	Unsigned int from 8 bits to 256 bits. uint256 is the same as uint.

Address:

address holds the 20 byte value representing the size of an Ethereum address. An address can be used to get the balance using .balance method and can be used to transfer balance to another address using .transfer method.

```
address x = 0x212;
address myAddress = msg.sender;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

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".

State Variable

Variables whose values are permanently stored in a contract storage.

```
pragma solidity ^0.8.0;
contract SolidityTest {
   uint storedData;      // State variable
   constructor() public {
      storedData = 10;   // Using State variable
   }
}
```

Local Variable

Variables whose values are available only within a function where it is defined. Function parameters are always local to that function.

```
pragma solidity ^0.8.0;
contract SolidityTest {
   uint storedData; // State variable
   constructor() public {
      storedData = 10;
```

```
  }
  function getResult() public view returns(uint){
    uint a = 1; // local variable
    uint b = 2;
    uint result = a + b;
    return result; //access the local variable
  }
}
```

Scope of local variables is limited to function in which they are defined but State variables can have three types of 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.

```
 pragma solidity ^0.8.0;
contract C {
  uint public data = 30;
  uint internal iData= 10;

  function x() public returns (uint) {
    data = 3; // internal access
    return data;
  }
}
contract Caller {
  C c = new C();
  function f() public view returns (uint) {
    return c.data(); //external access
  }
}
contract D is C {
  function y() public returns (uint) {
    iData = 3; // internal access
    return iData;
  }
  function getResult() public view returns(uint){
    uint a = 1; // local variable
    uint b = 2;
    uint result = a + b;
    return storedData; //access the state variable
  }
}
```

Operator

Solidity supports the following types of operators.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators
- Conditional (or ternary) Operators

String Type

Solidity supports String literal using both double quote (") and single quote ('). It provides string as a data type to declare a variable of type String.

```
pragma solidity ^0.8.0;
contract SolidityTest {
  string data = "test";
}
```

In above example, "test" is a string literal and data is a string variable. More preferred way is to use byte types instead of String as string operation requires more gas as compared to byte operation. Solidity provides inbuilt conversion between bytes to string and vice versa. In Solidity we can assign String literal to a byte32 type variable easily. Solidity considers it as a byte32 literal.

```
pragma solidity ^0.8.0;
contract SolidityTest {
  bytes32 data = "test";
}
```

Bytes to String Conversion

Bytes can be converted to String using string() constructor.

```
bytes memory bstr = new bytes(10);
string message = string(bstr);
```

Example

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

```
pragma solidity ^0.5.0;
contract SolidityTest {
  constructor() public{
  }
  function getResult() public view returns(string memory){
    uint a = 1;
    uint b = 2;
    uint result = a + b;
    return integerToString(result);
  }
  function integerToString(uint _i) internal pure
    returns (string memory) {

    if (_i == 0) {
      return "0";
    }
    uint j = _i;
    uint len;

    while (j != 0) {
      len++;
      j /= 10;
    }
    bytes memory bstr = new bytes(len);
    uint k = len - 1;

    while (_i != 0) {
      bstr[k--] = byte(uint8(48 + _i % 10));
      _i /= 10;
    }
    return string(bstr);
  }
}
```

Array

Array is a data structure, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

In Solidity, an array can be of compile-time fixed size or of dynamic size. For storage array, it can have different types of elements as well. In case of memory array, element type can not be mapping and in case it is to be used as function parameter then element type should be an ABI type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

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 arrayName [arraySize];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid Solidity data type. For example, to declare a 10-element array called balance of type uint, use this statement –

uint balance[10];

To declare an array of dynamic size in Solidity, the programmer specifies the type of the elements as follows –

type[] arrayName;

Initializing Arrays

You can initialize Solidity array elements either one by one or using a single statement as follows −

uint balance[3] = [1, 2, 3];

The number of values between braces [ ] can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

uint balance[] = [1, 2, 3];

You will create exactly the same array as you did in the previous example.

balance[2] = 5;

The above statement assigns element number 3rd in the array a value of 5.

Creating dynamic memory arrays

Dynamic memory arrays are created using new keyword.

uint size = 3;
uint balance[] = new uint[](size);

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

uint salary = balance[2];

The above statement will take 3rd element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

Members

- **length** − length returns the size of the array. length can be used to change the size of dynamic array be setting it.
- **push** − push allows to append an element to a dynamic storage array at the end. It returns the new length of the array.

Example

Example 1:

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.18;
contract array1{
uint[6] public arr;//=[10,20,30,40,50,60];
function setter(uint index,uint value) public{
arr[index]=value;
}
function getter(uint index) public view returns(uint x){
```

```
return (arr[index]);
}
function length() public view returns(uint){
return arr.length;
}

```

2.

```
uint[] public arr;

function pushele(uint item) public{
   arr.push(item);

}

function popele() public {

arr.pop();

}

function length() public view returns(uint){

return arr.length;
}
}
```

3.

```
contract test {
  function testArray() public pure{
     uint len = 7;
//dynamic array
     uint[] memory a = new uint[](7);

     //bytes is same as byte[]
     bytes memory b = new bytes(len);
```

ENUM

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

```
pragma solidity ^0.8.18;
```

```
contract Types {
    enum week_days
    {
      Monday,
      Tuesday,
      Wednesday,
      Thursday,
      Friday,
      Saturday,
      Sunday
     }

    week_days choice;
    week_days constant default_value = week_days.Sunday;
    function set_value() public {
      choice = week_days.Thursday;
    }
    function get_choice() public view returns (week_days) {
      return choice;
    }
    function getdefaultvalue() public pure returns(week_days) {
        return default_value;
    }
}
```

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 −

```
//SPDX-License-Identifier:MIT

pragma solidity ^0.8.18;

contract test {

   struct Book {

      string name;

      string writter;

      uint id;
```

```
    bool available;

  }

  Book book1;

  Book book2 = Book("Building Ethereum DApps","Roberto Infante ",2, false);

 function set_book_detail() public {book1 = Book("Introducing Ethereum and Solidity","Chris Dannen",1,
true);

  }

 function book_info()public view returns (string memory, string memory, uint, bool) {

      return(book2.name, book2.writter,book2.id, book2.available);

   }

function get_details() public view returns (string memory, uint) {

    return (book1.name, book1.id);

  }

}
```

Access a Struct and its variable

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the struct to define variables of structure type. The following example shows how to use a structure in a program.

Example

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.18;
contract test {
  struct Book {
    string name;
    string writter;
    uint id;
    bool available;
  }
  Book book1;
  Book book2 = Book("Building Ethereum DApps","Roberto Infante ",2, false);
 function set_book_detail() public {book1 = Book("Introducing Ethereum and Solidity","Chris Dannen",1,
true);
  }
 function book_info()public view returns (string memory, string memory, uint, bool) {
      return(book2.name, book2.writter,book2.id, book2.available);
   }
function get_details() public view returns (string memory, uint) {
    return (book1.name, book1.id);
  }
}
```

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.

```
pragma solidity ^0.8.0;

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

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

Constructors

The constructor is a special function that is called when the contract is deployed to the blockchain. It is used to initialize the contract and set its initial state. The constructor typically takes any necessary arguments, such as the initial values for the contract's variables.
For example:

```
constructor(string memory _name) public {
    name = _name;
}
```

Events

These are used to emit log messages that can be used to track the contract's execution and state. Events are typically used to provide a record of important actions that have taken place within the contract, such as the completion of a transaction. They can be used to trigger external functions or to provide information to users of the contract.
For example:

```
event Deposit(address indexed _from, uint256 _value);
event Withdrawal(address indexed _from, uint256 _value);

function deposit(uint256 _amount) public {
    balance = balance.add(_amount);
    emit Deposit(msg.sender, _amount);
}
```

```
function withdraw(uint256 _amount) public {
    require(balance >= _amount, "Insufficient balance");
    balance = balance.sub(_amount);
    msg.sender.
```

Modifiers

These are used to modify the behavior of functions. They can be used to check for certain conditions before allowing a function to execute. For example, a modifier might be used to check that a certain requirement has been met before allowing a function to be called, or to ensure that a function can only be called by certain users.
For example:

```
modifier onlyOwner {
    require(msg.sender == owner, "Only the owner can perform this action");
    _;
}

function withdraw(uint256 _amount) public onlyOwner {
    require(balance >= _amount, "Insufficient balance");
    balance = balance.sub(_amount);
    msg.sender.transfer(_amount);
}
```

Example2:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract AccessControl {
    address public owner;

    // Modifier to check if the caller is the owner
    modifier onlyOwner() {
        require(msg.sender == owner, "Not the contract owner");
        _;
    }

    constructor() {
        owner = msg.sender; // Set the deployer as the owner
    }

    // Function restricted to the owner
    function restrictedFunction() public onlyOwner {
        // Function logic here
    }
}
```

8. The fallback function

This is a special function that is called when the contract receives an external call to an undefined function. It is used to handle situations where an external party tries to call a function that does not exist in the contract. The fallback function can be used to provide a default behavior for the contract, such as returning an error message or ignoring the call. For example:

```
function() external {
    revert("This function does not exist");
}
```

In conclusion, it is important to design and write smart contracts carefully to ensure that they accurately reflect the terms of the agreement and function as intended. And for that it is crucial to know what components make up the structure of a contract so that you fully understand and work with them.

Solidity – Inheritance

Inheritance is one of the most important features of the object-oriented programming language. It is a way of extending the functionality of a program, used to separate the code, reduces the dependency, and increases the re-usability of the existing code. Solidity supports inheritance between smart contracts, where multiple contracts can be inherited into a single contract. The contract from which other contracts inherit features is known as a base contract, while the contract which inherits the features is called a derived contract. Simply, they are referred to as parent-child contracts. The scope of inheritance in Solidity is limited to public and internal modifiers only. Some of the key highlights of Solidity are:

- A derived contract can access all non-private members including state variables and internal methods. But using this is not allowed.
- Function overriding is allowed provided function signature remains the same. In case of the difference of output parameters, the compilation will fail.
- We can call a super contract's function using a super keyword or using a super contract name.
- In the case of multiple inheritances, function calls using super gives preference to most derived contracts.

Solidity provides different types of inheritance.

1. Single Inheritance

In Single or single level inheritance the functions and variables of one base contract are inherited to only one derived contract.

Example:

1.

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.18;
//Single
contract parent{
    uint internal sum;
    function setValue() external {
        uint a = 10;
        uint b = 20;
        sum = a + b;
    }
}
```

```
contract child is parent{
    function getValue() external view returns(uint) {
        return sum;
    }
}
contract caller {
    child cc = new child();
    function testInheritance() public returns (uint) {
        cc.setValue();
        return cc.getValue();
    }
}
```

2. Multi-level Inheritance

It is very similar to single inheritance, but the difference is that it has levels of the relationship between the parent and the child. The child contract derived from a parent also acts as a parent for the contract which is derived from it.

```
Example: 2. //SPDX-License-Identifier:MIT

pragma solidity ^0.8.18;
//Multilevel
contract A {
    string internal x;
    string a = "Allthebest" ;
    string b = "For exams";
    function getA() external{
    x = string(abi.encodePacked(a,b));
    }
}
contract B is A {
    string public y;
    string c = "Allthebest";
    function getB() external payable returns(string memory){
        y = string(abi.encodePacked(x, c));
    }
}
contract C is B {
    function getC() external view returns(string memory){
        return y;
    }
}
contract caller {
    C cc = new C();
    function testInheritance() public returns (string memory) {
        cc.getA();
        cc.getB();
        return cc.getC();
    }
}
```

3. Hierarchical Inheritance

In Hierarchical inheritance, a parent contract has more than one child contracts. It is mostly used when a common functionality is to be used in different places.

Example:

```
3. //SPDX-License-Identifier:MIT
pragma solidity ^0.8.18;
//heirarchical
contract A {
    string internal x;
    function setX() external {
        x = "Allthebest";
    }
    uint internal sum;
    function setA() external {
        uint a = 10;
        uint b = 20;
        sum = a + b;
    }
}
contract B is A {
    function getAstr() external view returns(string memory){
        return x;
    }
}
contract C is A {
    function getAValue() external view returns(uint){
            return sum;
    }
}
contract caller {
    B contractB = new B();
    C contractC = new C();
    function testInheritance() public returns (string memory, uint) {
        contractB.setX();
        contractC.setA();
        return (contractB.getAstr(), contractC.getAValue());
    }
}
```

4. Multiple Inheritance

In Multiple Inheritance, a single contract can be inherited from many contracts. A parent contract can have more than one child while a child contract can have more than one parent.

Example:

```
4. //SPDX-License-Identifier:MIT

pragma solidity ^0.8.18;
// Multiple
contract A {
    string internal x;
    function setA() external {
        x = "Allthebest";
    }
}
contract B {
    uint internal pow;
    function setB() external {
        uint a = 2;
        uint b = 20;
        pow = a ** b;

    }
}
contract C is A, B {
function getStr() external view returns(string memory) {
        return x;
    }
    function getPow() external view returns(uint) {
        return pow;
    }
}
contract caller {
    C contractC = new C();
    function testInheritance() public returns(string memory, uint) {
        contractC.setA();
        contractC.setB();
        return (contractC.getStr(), contractC.getPow());
    }
}
```

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.

```
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;
  }
}
```

Function Categories:

View, Pure, Fallback, Modifier, Overloading, Mathematical, Cryptographic

View

View functions ensure that they will not modify the state. A function can be declared as **view**. The following statements if present in the function are considered modifying the state and compiler will throw warning in such cases.

- Modifying state variables.
- Emitting events.
- Creating other contracts.
- Sending Ether via calls.
- Calling any function which is not marked view or pure.
- Using low-level calls.
- Using inline assembly containing certain opcodes.

Pure functions ensure that they not read or modify the state. A function can be declared as **pure**. The following statements if present in the function are considered reading the state and compiler will throw warning in such cases.

- Reading state variables.
- Accessing address(this).balance or <address>.balance.
- Accessing any of the special variable of block, tx, msg (msg.sig and msg.data can be read).
- Calling any function not marked pure.
- Using inline assembly that contains certain opcodes.

Pure functions can use the revert() and require() functions to revert potential state changes if an error occurs.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

```
contract Test {
  function getSum(uint a, uint b) public pure returns(uint){
    return a + b;
  }
  function getSum(uint a, uint b, uint c) public pure returns(uint){
    return a + b + c;
  }
  function callSumWithTwoArguments() public pure returns(uint){
    return getSum(1,2);
  }
  function callSumWithThreeArguments() public pure returns(uint){
    return getSum(1,2,3);
  }
}
```

inbuilt mathematical functions

- **addmod(uint x, uint y, uint k) returns (uint)** − computes (x + y) % k where the addition is performed with arbitrary precision and does not wrap around at 2^{256}.
- **mulmod(uint x, uint y, uint k) returns (uint)** − computes (x * y) % k where the addition is performed with arbitrary precision and does not wrap around at 2^{256}.

```
contract Test {
  function getSum(uint a, uint b) public pure returns(uint){
    return a + b;
  }
  function getSum(uint a, uint b, uint c) public pure returns(uint){
    return a + b + c;
  }
  function callSumWithTwoArguments() public pure returns(uint){
    return getSum(1,2);
  }
  function callSumWithThreeArguments() public pure returns(uint){
    return getSum(1,2,3);
  }
}
```

Cryptographic functions as well. Following are important methods

- **keccak256(bytes memory) returns (bytes32)** − computes the Keccak-256 hash of the input.
  Keccak256 is a cryptographic hash function that takes an input of an arbitrary length and produces a fixed-length output of 256 bits. It is the function used to compute the hashes of Ethereum addresses, transaction IDs, and other important values in the Ethereum ecosystem
- **ripemd160(bytes memory) returns (bytes20)** − compute RIPEMD-160 hash of the input.
- RIPEMD-160 is a 160-bit cryptographic hash function. It is intended for use as a replacement for the 128-bit hash functions MD4, MD5, and RIPEMD. RIPEMD was developed in the framework of the EU project RIPE (RACE Integrity Primitives Evaluation, 1988-1992).
- **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 or return zero on error. The function parameters correspond to ECDSA values of the signature: r - first 32 bytes of signature; s: second 32 bytes of signature; v: final 1 byte of signature. This method returns an address. ecrecover() is a very useful Solidity function that allows the smart contract to validate that incoming data is properly signed by an expected party.

Libraries

Libraries in solidity are similar to contracts that contain reusable codes. A library has functions that can be called by other contracts.

Deploying a common code by creating a library reduces the gas cost. Functions of the library can be called directly when they do not modify the state variables i.e. only pure and view functions can be called from outside of the library.

It cannot be destroyed because it is assumed as stateless.

The library does not have state variables, it cannot inherit any element and cannot be inherited.

Creating Library

A library contract is defined by using the library keyword instead of a general contract. Libraries do not have any storage thus it cannot hold state variables, fallback or payable functions also cannot be created inside the library as it cannot store ethers.

Libraries are not for changing the state of the contract, it can only be used for performing basic operations based on inputs and outputs. However it can implement some data types like struct and enums which are user-defined, and constant variables that are stored in a stack of Ethereum, not in storage.

Syntax:

```
library <libraryName> {
    // block of code
}
```

Deploying Library Using ‘For’ Keyword

A library can be defined on the same contract as well as it can be imported from outside by using the import statements.

Example:
import <libraryName> from “./library-file.sol”;

A single file can contain multiple libraries that can be specified using curly braces in the import statement separated by a comma. A library can be accessed within the smart contract by using ‘for’ keyword.

Syntax:
<libraryName> for <dataType>

The above statement can be used to attach library functions to any type. *libraryName* is the name of the desired library to import, *dataType* is the variable type for which we want to access the library. All members of the library can also be used by the wildcard operator(*).
Example:**e:**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
library LibExample {
   function pow(uint a, uint b) public pure returns (uint) {
        return (a ** b);
    }
}
contract LibraryExample {

    using LibExample for uint;

function getPow(uint num1, uint num2) public pure returns ( uint) {
        return num1.pow(num2);
    }
}
```

Solidity – Error Handling

Solidity has many functions for error handling. Errors can occur at compile time or runtime. Solidity is compiled to byte code and there a syntax error check happens at compile-time, while runtime errors are difficult to catch and occurs mainly while executing the contracts. Some of the runtime errors are out-of-gas error, data type overflow error, divide by zero error, array-out-of-index error, etc. Until version 4.10 a single throw statement was there in solidity to handle errors, so to handle errors multiple if…else statements, one has to implement for checking the values and throw errors which consume more gas. After version 4.10 new error handling construct **assert, require, revert** statements were introduced and the throw was made absolute.

Require Statements

The 'require' statements declare prerequisites for running the function i.e. it declares the constraints which should be satisfied before executing the code. It accepts a single argument and returns a boolean value after evaluation, it also has a custom string message option. If false then exception is raised and execution is terminated. The unused gas is returned back to the caller and the state is reversed to its original state. Following are some cases when require type of exception is triggered:

- When require() is called with the arguments which result as false.
- When a function called by a message does not end properly.
- When a contract is created using the new keyword and the process does not end properly.
- When a codeless contract is targeted to an external function.
- When ethers are sent to the contract using the public getter method.
- When .transfer() method fails.
    - When an assert is called with a condition that results in false.
    - When a zero-initialized variable of a function is called.
    - When a large or a negative value is converted to an enum.
    - When a value is divided or modulo by zero.
    - When accessing an array in an index which is too big or negative.

Example: In the below example, the *contract requireStatement* demonstrates how to use the 'require statement'.

```
// Solidity program to demonstrate require statement
pragma solidity ^0.5.0;


// Creating a contract
contract requireStatement {


    // Defining function to check input
    function checkInput(uint _input) public view returns(string memory){
        require(_input >= 0, "invalid");
        require(_input <= 255, "invalid");


        return "Input is invalid";
    }


    // Defining function to use require statement
    function Odd(uint _input) public view returns(bool){
        require(_input % 2 != 0);
        return true;
    }
}
```

Assert Statement

Its syntax is similar to the require statement. It returns a boolean value after the evaluation of the condition. Based on the return value either the program will continue its execution or it will throw an exception. Instead of returning the unused gas, the assert statement consumes the entire gas supply and the state is then reversed to the original state. Assert is used to check the current state and function conditions before the execution of the contract. Below are some cases with assert type exceptions:

- When an assert is called with a condition that results in false.
- When a zero-initialized variable of a function is called.
- When a large or a negative value is converted to an enum.
- When a value is divided or modulo by zero.
- When accessing an array in an index which is too big or negative.

Example: In the below example, the *contract assert Statement* demonstrates how to use an ‘assert statement’.

```
// Solidity program to demonstrate assert statement
pragma solidity ^0.5.0;

// Creating a contract
contract assertStatement {
     // Defining a state variable
    bool result;


    // Defining a function to check condition


    function checkOverflow(uint _num1, uint _num2) public {
        uint sum = _num1 + _num2;
        assert(sum<=255);
        result = true;
    }

    // Defining a function to print result of assert statement
    function getResult() public view returns(string memory){
         if(result == true){
            return "No Overflow";
         }
         else{
            return "Overflow exist";
         }
    }
}
```

Revert Statement

This statement is similar to the require statement. It does not evaluate any condition and does not depends on any state or statement. It is used to generate exceptions, display errors, and revert the function call. This statement contains a string message which indicates the issue related to the information of the exception. Calling a revert statement implies an exception is thrown, the unused gas is returned and the state reverts to its original state. Revert is used to handle the same exception types as require handles, but with little bit more complex logic.

Example: In the below example, the *contract revertStatement* demonstrates the 'revert statement'.

```
// Solidity program to demonstrate revert statement
pragma solidity ^0.5.0;

// Creating a contract
contract revertStatement {

   // Defining a function to check condition
   function checkOverflow(uint _num1, uint _num2) public view returns(string memory,
uint){
          uint sum = _num1 + _num2;
           if(sum < 0 || sum > 255){
               revert(" Overflow Exist");
           }
           else{
               return ("No Overflow", sum);
           }
     }
}
}
```

Sample Smart Contract

1. Simple Amount Transfer

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.18;

contract Coin {

    address public minter;
    mapping (address => uint) public balances;
```

```
   event Sent(address from, address to, uint amount);


    function Coins() public{
        minter = msg.sender;
        }


    function mint(address receiver, uint amount) public  {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
       emit Sent(msg.sender, receiver, amount);
    }
}
```

2. Lottery:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;
contract Lottery{
    address public manager;
    address payable[] public participants;
    constructor()
    {
        manager=msg.sender;
    }

    receive() external payable{
        require(msg.value==1 ether);
        participants.push(payable (msg.sender));
    }
    function getBalance() public view returns(uint)
    {
        require(msg.sender==manager);
        return address(this).balance;
    }
    function random() public view returns(uint){
       return
uint(keccak256(abi.encodePacked(block.prevrandao,block.timestamp,participants.
length)));
    }
    function selectWinner() public {
        require(msg.sender==manager);
        require(participants.length>=3);
        uint r=random();
        address payable winner;
        uint index =r%participants.length;
        winner=participants[index];
        winner.transfer(getBalance());
        participants=new address payable[](0);
    }

}
```

3: Function call in another contract

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.18;
contract Callee{
    uint public x;
    uint public value;

    function SetX(uint _x) public returns (uint){
        x=_x;
        return x;
    }

}
contract Caller{
    function setX(Callee _callee, uint _x) public{
        uint x=_callee.SetX(_x);
    }
    function setXfromAddress(address _add, uint _x) public{
        Callee call=Callee(_add);
        call.SetX(_x);
    }

}
```

4. Voting Based System:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Voting {

    address public owner;
    string[] public candidates;
    mapping(address => bool) public voters;
    mapping(string => uint256) public votes;
    mapping(address=> bool) public voted;
    event Voted(address voter, string candidate);

    constructor(string[] memory _candidates) {
        owner = msg.sender;
        candidates = _candidates;
    }

    function register() public {
        require(msg.sender != owner);
        require(!voters[msg.sender]);
        voters[msg.sender] = true;
        voted[msg.sender] =false;
    }

    function vote(uint256 _candidateIndex) public {
        require(voters[msg.sender]);
        require(voted[msg.sender]==false);
        require(_candidateIndex < candidates.length);

        string memory selectedCandidate = candidates[_candidateIndex];
        votes[selectedCandidate]++;
        emit Voted(msg.sender, selectedCandidate);
        voted[msg.sender]=true;
    }

    function getTotalVotes(string memory _candidate) public view returns
(uint256)
    {
        return votes[_candidate];
    }
}
```

5. Auction Smart Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Auction {
    address public auctioneer;
```

```solidity
    address public highestBidder;
    uint public highestBid;

    mapping(address => uint) public bids;

    bool public ended;

    modifier onlyAuctioneer() {
        require(msg.sender == auctioneer, "Only auctioneer can perform this action");
        _;
    }

    modifier notAuctioneer() {
        require(msg.sender != auctioneer, "Auctioneer cannot bid in the auction");
        _;
    }

    event NewHighestBid(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    constructor() {
        auctioneer = msg.sender;
        ended = false;
    }

    function bid() public payable notAuctioneer {
        require(!ended, "Auction has ended");

        require(msg.value > highestBid, "Bid amount is lower than the current highest bid");

        if (highestBid != 0) {
            bids[highestBidder] += highestBid;
        }

        highestBidder = msg.sender;
        highestBid = msg.value;

        emit NewHighestBid(msg.sender, msg.value);
    }
function endAuction() public onlyAuctioneer {
        require(!ended, "Auction already ended");
        ended = true;

        emit AuctionEnded(highestBidder, highestBid);

        payable(auctioneer).transfer(highestBid);
    }

    function withdraw() public {
        uint amount = bids[msg.sender];
        require(amount > 0, "No funds to withdraw");

        bids[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}

```

7. Crowdsourcing Smart Contract

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.9;

contract CrowdFunding1 {
    struct Campaign{
    address owner;
    string title;
    string description;
    uint256 target;
    uint256 deadline;
    uint256 amountcollected;
    string image;
    address[] donators;
    uint256[] donations;
}

mapping(uint256=>Campaign) public campaigns;

uint256 public numbeOfCampaigns=0;

function createCampaign(address _owner,string memory _title, string memory
_description, uint256 _target, uint256 _deadline, string memory _image) public
returns (uint256){
    Campaign storage campaign=campaigns[numbeOfCampaigns];

    require(campaign.deadline<block.timestamp, "The deadline should be a date
in the future");
```

```
    campaign.owner= _owner;
    campaign.title= _title;
    campaign.deadline=_deadline;
    campaign.description=_description;
    campaign.target=_target;
    campaign.image=_image;

    numbeOfCampaigns++;

    return numbeOfCampaigns -1;

}
function donateToCampaign(uint256 _id) public payable{
    uint256 amount=msg.value;
    Campaign storage campaign=campaigns[_id];
    campaign.donators.push(msg.sender);
    campaign.donations.push(amount);

    (bool sent,) = payable(campaign.owner).call{value: amount}("");

    if(sent){
        campaign.amountcollected=campaign.amountcollected+amount;
    }

}
function getDonators(uint256 _id) view public returns (address[] memory,
uint256[] memory){
return (campaigns[_id].donators, campaigns[_id].donations);
}
function getCampaigns() public view returns (Campaign[] memory){
    Campaign[] memory allCampaigns=new Campaign[](numbeOfCampaigns);
    for(uint i=0;i<numbeOfCampaigns;i++){
        Campaign storage item =campaigns[i];
        allCampaigns[i]=item;
    }
    return allCampaigns;
}
}
```