SOLIDITY CONCEPTS

ABI and Contract Bytecode

Solidity compilation (Source code is compiled to **ABI** and **Contract Bytecode** which is deployed on Ethereum Blockchain), whereas ABI (Application Binary Interface) is used to interact with the contract.

- Contract Bytecode is usually *public* and unencrypted and is executed by every ETH node.
- Opcodes are the *translation* from the Contract Bytecode which are then executed by the ETH nodes.

Solidity Compilation

- solc is the solidity CLI compiler.
- Remix compiler is the compiler from Remix IDE
- solcjs is the solidity Javascript compiler

Contract Deployment

Solidity contract is first compiled using one of the above three methods, and then can be deployed on either the injected Web3 Provider (in Remix IDE) which could be either one of the test net chains or the mainnet chain, or it could be an in memory chain.

Solidity Contract

Basic Concepts

- Each solidity contract starts with **SPDX (Software Package Data Exchange)** Identifier such as MIT/GPL-3.0 etc. The solidity compiler does not validate the license, but includes the license string in the bytecode metadata.
- Each solidity contract starts with `pragma solidity <version>` which specifies the compiler version to use for compiling the solidity contract.
    - A version of `^0.8.2` specifies a compiler version that is `0.8.2` or higher but less than `0.9.0`.
    - A version range can also be specified like `>=0.8.0 <0.9.0`, which specifies a compiler version that is greater than or equal to `0.8.0` but less than `0.9.0`.

Contract Structure

Below is a basic structure of a solidity contract

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

contract Property {
    uint private price;
    address public owner;

    constructor() {
        price = 0;
        owner = msg.sender;
    }

    // Function modifier
    modifier onlyOwner() {
      require(msg.sender = owner);
      _;
    }

    function changeOwner(address _owner) public onlyOwner {
      owner = _owner;
    }

    function setPrice(uint _price) public {
      price = _price;
    }

    function getPrice() view public returns (uint) {
      return price;
    }

    // Event
    event OwnerChanged(address owner);
}
```

EVM Storage

There are three types of EVM storage:

- Storage
- Stack
- Memory

Storage	Stack	Memory
Holds state variables	Holds function local variables (ex. int)	Holds arguments passed to a function and local variables in a function that are reference types: string, array, mapping, struct
Costs gas	Free	Free
Persistent	Ephermeral (changes with function calls)	Ephermeral
Like a HDD	Like a RAM	Like a RAM

Variables

There are three types of variables in solidity:

- **State Variables** - are private and mutable by default. State variables can be declared as constants/immutable.
- **Local Variables** - are declared within a function body.
- **Global Variables** - are pre-existing variables that can be used within a contract/function body.

> **Note**
>
> A constant variable must be initialiazed at the time of declaration. If a variable is meant to be constant in constructor, it should be declared as immutable.

State Variables	Local Variables
Declared at contract level	Declared within a function body
Permanently stored in contract storage	Stored on stack and not on contract storage (do not set). Exceptions are arrays, structs, mappings and strings, where storage location must be specified (memory).
Can be set as constants	Cannot be set as constants
Expensive to use, as they cost gas for storage	Do not cost gas to use and therfore free
Initialized at **declaration**, or using a **constructor**, or using **setters** after contract deployment	If using **memory** keyword and are arrays or structs, initialized at runtime. Illegal to use memory keyword at contract level.

State variables

Below is an example of state variables.

```
// SPDX-License-Identifier: GPL-3.0
// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of state variables
contract Property {
    // Price is a state variable which is initialized to
default value of 0.
    uint private price;

    // Name is a state variable which is initialized to
assigned value of "Rishab"
    string public name = "Rishab";

    // Location is a constant state variable which must be set
    // at the time of declaration.
    string constant public location = "London";

    // Price can also be initialized via a constructor
    constructor(unint _price) {
      price = _price;
    }

    // Price can also be initialized via a setter
    function setPrice(uint _price) public {
      price = _price;
    }
}
```

Local Variables

Below is an example of local variables.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of local variables
contract Property {
    // The function is set as pure because it does not touch
the blockchain - local variables
    // are stored on the stack.
    function localVariableStack() public pure returns (int) {

      // x is local variable stored on the stack (free).
      int x = 5;

      // An operation is performed on the local variable stored
on stack.
      x = x * 5;

      // The modified variable from the stack is returned.
      return x;
    }

    // This shows an example of string/array/mapping/struct
data type which are by default
    // stored in the storage, and where the type of storage -
storage, memory, calldata needs
    // to be specified. The return type of this function is
also `memory` string.
    function localVariableMemory() public pure returns (memory
string) {

      // If memory keyword is ommitted there will be a compiler
error.
      memory string name = "ETH";

      return name;
    }
}
```

Global variables

There exist special variables and functions in solidity which exist in the global namespace and are mainly used to provide information about the blockchain or utility functions.

Below is a list of global variables in Solidity:

- **msg**: contains information about the account that generates the transaction and also about the transaction or call.
    - **msg.sender**: account address that generates the transaction.
    - **msg.value**: eth value (represented in wei) sent to this contract.
    - **msg.data**: data field from the transaction or call that executed this function
- **this**: the current contract, explicitly convertible to Address. For example, **address(this).balance** returns the balance of the contract.
- **gasleft()**: remaining gas for the transaction
- **block.timestamp**: alias for **now**, returns the current block timestamp since Unix epoch.
- **block.number**: current block number.
- **block.difficulty**: current block difficulty.
- **block.gaslimit**: current block gas limit.
- **block.coinbase**: current block miner's address.
- **tx.gasprice**: gas price of the transaction.
- **tx.origin**: sender of the transaction (full call chain).

Below is an example contract showing usage of global variables

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of Global variables.
contract GlobalVariables {
  // State variable storing the address of the owner of the
contract.
  address public owner;
  // State variable storing the ETH amount that is sent.
  uint public sentValue;
  // Constructor demonstrating usage of global variable
'msg.sender' to set
  // the owner of this contract.
  constructor() {
    owner = msg.sender;
  }

  // A function demonstrating usage of global variable to
update the state
  // of the state variable.
  function changeOwner() {
    owner = msg.sender;
  }

  // This function demonstrates the usage of 'msg.value' to
extract the information
  // of the amount of ETH that is being sent.
  // The function is declared payable because any function that
sends ETH must be
  // declared payable.
  function sendEther() public payable {
    sentValue = msg.value;
  }

  // A contract can have its own balance as well.
  // This function returns the balance of the contract by
converting 'this'
  // into an address, and returning of the balance of the
address.
  function getBalance() public view returns(uint) {
    return address(this).balance;
  }

  // This function demonstrates the usage of 'gasleft()'
function by
  // performing a mathematical operation and calculating the
gas consumed
  // by the mathematical operation.
  function howMuchGas() public view returns(uint) {
    uint start = gasleft();
    uint j = 1;
    for (uint i = 0; i<20; i++) {
      j *= i;
    }
    uint end = gasleft();
    return start - end;
  }
}
```

Constructor

- A constructor can be a no args constructor, but in case any variables need to initialized the arguments for them are accepted into the constructor to initialize their corresponding values.
- A constructor is by default public.
- A constructor is called at the time of contract deployment, and the arguments for the constructor need to be passed at the time of deployment.
- It is normal to register the creator of the contract in the constructor of the contract.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of Constructor.
contract Property {
    // Price is a private state/global variable set to default
value of 0.
    uint price;

    // Name is public state/global variable, set to default
value of empty string.
    string public name;

    // Owner is an immutable public variable, the owner of this
contract.
    address immutable public owner;

    // The constructor takes two params: _price and _name.
    // _price is used to initialize the price state variable.
    // _name is used to initialize the name state variable.
    constructor(uint _price, string memory _name) {
      price = _price;
      name = _name;

      // This is the owner of the contract.
      // msg.sender is the address that sends the transaction
for deploying the contract,
      // who then becomes the owner of the contract.
      owner = msg.sender;
    }
}
```

Function

Definition of a function has the following components:

- The `function` keyword
- The function name
- Parentheses () which contains comma separated list of arguments in the format `<datatype> _variableName`. For example, `unint _name`.
- The function type - **public**, **private**, **internal**, and **external**.
    - A public function can be called both internally within the contract and externally from another contract.

Below are a few examples of function definitions:

1. The below function sets the name, and can be called both internally and externally.

    `funtion setName(string memory _name) public`

2. The below function gets the price and is set is `public` and `view` which means it does not modify the blockchain.

    `function getPrice() public view returns (uint)`

Function Modifiers

Function modifiers modify the behavior of the function on which they are applied. They test a condition prior to calling that function, and the function is called only when the condition in the modifier evaluates to true.

Some of the traits of function modifiers are below:

- They avoid rewriting duplicate code.
- They are contract properties and are inherited
- They do not return any value
- They only use **require()**
- They are defined using **modifier** keyword.

Below is an example contract demonstrating the usage of function modifiers.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of function modifiers
contract Property {
    // price is a private state/global variable.
    uint private price;

    // owner is public state/global variable.
    address public owner;

    constructor() {
      // owner is initialized to the deployer of the smart
contract
      owner = msg.sender;
    }

    // Function modifier is created just like a normal function
instead using
    // the 'modifier' keyword. This modifier if applied to any
function enforces the check
```

```
    // that the function can only be executed by an owner.
    //
    // The function modifier can only have 'require()'
statements.
    //
    // The '_' ending statement inserts the body of the
function on which this modifier
    // is applied.
    modifier onlyOwner() {
      require(owner == msg.sender);
      _;
    }

    // Only the owner can change the owner, as the function has
'onlyOwner' modifier.
    // Attempting to execute this function from any other
address will throw an error.
    function changeOwner(address _owner) public onlyOwner {
      owner = _owner;
    }

    // Only the owner can change the price, as the function has
'onlyOwner' modifier.
    // Attempting to execute this function from any other
address will throw an error.
    function changePrice(uint _price) public onlyOwner {
      price = _price;
    }
}
```

Getter and Setter Functions

A getter function returns the initialized value of the state/global variable, whereas a setter function updates the previously initialized value of the state/global variable to the newly provided value. A getter function is only needed if the state variable is declared as private. If the state variable were declared public, the corresponding getter function for it is already created.

A getter function does not cost any gas, as it is simply a read only operation on the blockchain.

A setter function costs the gas as it updates the state of the previously initialized variable.

Below is an example contract of writing idiomatic getter and setter function.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of getter and setter functions
contract Property {
    // Price is a private state/global variable only accessible
within the contract.
    // It is initialized to a default value of 0.
    uint private price;

    // Name is public state/global variable, and as a result
the corresponding getter
    // function for it is automatically created. Only a setter
function is needed.
    string public name;

    /// A getter function for the private state variable price.
    /// @notice This function does not cost any gas.
    /// @return Returns the currently set value of the price.
    function getPrice() public view returns (uint) {
      return price;
    }

    /// A setter function for the private state variable price.
    ///
    /// @notice This function costs gas.
    function setPrice(uint _price) public {
      price = _price;
    }

    /// A setter function for the public state variable name.
    ///
    /// @dev A getter function is not needed, as the variable
is public.
    function setName(string memory _name) public {
      name = _name;
    }
}
```

Events

Each Ethereum transaction has a receipt attached to it which contains one or more log entries. These log entries are referred to as events in solidity and logs in EVM and Yellow pages.

Events have the following properties/features:

- Events are not accessible from within contracts, not even within the contracts that emit them.
- Events are only accessible from external actors such as JS
- Events allow JavaScript callback functions that listen to them to update the UI.
- Events are inheritable members of a contract, so if an event is declared in an interface or base contract, it is accessible in all derived contracts.

An event is declared using the below syntax in solidity.

```
// declare an Event
// By convention Events in solidity are always declared with
Capitalized first letter.
event Transfer(address _to, uint _value);
```

Events are emitted in Solidity using the following syntax in setter functions.

```
// emit an Event
emit Transfer(_to, msg.value);
```

Data Types

The following are the data types in Solidity:

- int8 to int256 in steps of 8 (int is alias for int256) - default value 0.
- uint8 to uint256 in steps of 8 (uint is alias for uint256) - default value 0
- bool - default value false
- address
- bytes
- string
- array
- struct
- enum
- mapping

> **Caution**
>
> There is no float or double support in solidity.

> **Note**
>
> If using solidity version less than 0.8.0 use SafeMath library to avoid buffer overflow and underflow.

Address

There are two types of addresses in ETH blockchain

- **Plain address** - is an address that cannot receive ETH.
- **Payable address** - is an address that can receive ETH.

An address has the following members:

- **balance** - only for payable addresses.
- **transfer()** - only for payable addresses.
- **send()** - only for payable addresses.
- **call()**, **callcode()**, **delegatecall()**, **staticcall()** - for both plain and payable addresses.

Contract Address

A solidity contract has its own address, which is a 20 byte address string and is generated from:

- the account, and
- the nonce that is used to deploy contract.

A contract address cannot be computed in advance.

> **Caution**
>
> A contract can receive ETH and have ETH balance only if it has a payable function defined.

Receiving ETH

A contract can receive ETH in two possible ways:

- **receive() and fallback()** - if the contract has receive or fallback function declared as external without function keyword, an EOA can simply send ETH to the contract.
- **payable function** - if a contract has a function defined as payable by calling that function, one can send ETH to the contract.

> **Tip**

> if neither of the two - receive/fallback or payable function is present, an attempt to send ETH to the contract will fail.

> **Caution**
>
> The ETH balance of a contract is in possession of anyone who can call **transfer()** function on the contract.

Below is an example contract demonstrating on how to receive ETH in a contract.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing how to receive ETH in a contract
contract Deposit {
  // The receive function does not need a body.
  // It is declared without a function keyword and as external
and payable.
  receive() external payable {}

  // The fallback function just like receive function does not
have a function keyword
  // nor does require a function body. It is declared as both
external and payable.
  //
  // The fallback function gets triggered if neither
'receive()' function
  // or any payable function is found in the contract.
  fallback() external payable {}

  // This function gets the current balance of the contract.
  function getBalance() public view returns(uint) {
    return address(this).balance;
  }

  // This function sends ETH to the contract. The caller can
call this function to send ETH
  // to this contract.
  function sendEther() public payable { }
}
```

Transferring ETH

To enable transferring ETH from a contract, a contract should have a function that declares an address as a payable parameter, and the amount to transfer.

There are two possible ways to transfer ETH from an address to another address:

- **transfer()** - the transfer function is transactional, and if it fails the entire transaction is reverted.
- **send()** - send is a low level function, that returns the success state of the transaction. The onus is on the user to test the return value of the send() function call. The send() function, if invoked, without testing the return value, will always succeed, whether the ETH transfer was successful or not.

Below is an example contract that demonstrates sending ETH to an address from the contract balance.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing how to transfer ETH from a contract to a
recepient.
contract Deposit {
  // owner is a state variable that stores the address of the
contract owner.
  address public owner;

  // The constructor initializes the owner of the contract.
  constructor() {
    owner = msg.sender;
  }
  /// This function demontrates the usage of transfer()
function on address.
  ///
  /// @notice This function takes in two arguments. The
recepient address '_recepient' and
  ///         the amount to transfer '_amount'. The function
checks whether the contract
  ///         has sufficient ETH to transfer to the recepient.
If not, it returs false, else
  ///         returns true. Furthermore, this operation is only
allowed for the owner of the
  ///         contract.
  function transferETH(address payable _recepient, uint
_amount) public returns(bool) {
    require(owner == msg.sender, "You are not the owner");
    if (address(this).balance < _amount) {
      return false;
    }
    _recepient.transfer(_amount);
    return true;
  }

  /// This function demonstrates the usage of send() function
on address.
  ///
  /// @notice This function takes in two arguments. The
recepient address '_recepient' and
  ///         the amount to transfer '_amount'. The function
checks whether the contract
  ///         has sufficient ETH to transfer to the recepient.
If not, it returs false, else
  ///         returns true. Furthermore, this operation is only
allowed for the owner of the
  ///         contract.
  function sendETH(address payable _recepient, uint _amount)
public returns(bool) {
    require(owner == msg.sender, "You are not the owner");
      if (address(this).balance < _amount) {
        return false;
      }
      bool success = _recepient.send(_amount);
      require(success, "Transfer was unsuccessful");
      return true;
    }
}
```

Arrays - Fixed and Dynamic

	Fixed Size Array	**Dynamic Array**
Size	Fixed at compile time	Bytes
Example	bytes1, bytes2, ..., bytes32, uint(3)	string (UTF-8 encoded)
Data Type	Any	Any
Properties	length	length, push and pop
Gas	Cheaper	Expensive

Below is an example of **fixed sized array**

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of fixed size array
contract FixedSizeArray {
    // Numbers is a fixed size array of size 3.
    // The fixed sized array may or may not be initialized.
    uint[3] public numbers = [ 2, 3 ,4 ];

    // Bytes fixed sized array.
    // bytes1 is length 1 byte/8 bits.
    bytes1 public b1;
    // bytes2 is length 2 bytes/16 bits.
    bytes2 public b2;
    // bytes3 is length 3 bytes/24 bits.
    bytes3 public b3;
    // ... upto bytes32

    /// @notice If the below function is called with an index 3
or higher, the
```

```
    ///         transaction will fail. Alternatively, an if
condition can be set
    ///         to check for index greater than or equal to the
length of the array.
    function setElement(uint _index, uint _value) public {
      numbers[_index] = _value;
    }

    // Returns the length of the numbers array.
    function getNumbersLength() public view returns(uint) {
      return numbers.length;
    }

    // Sets the values of fixed size arrays of bytes1, bytes2,
and bytes3
    function setBytesFixedArray() public {
      b1 = 'a';
      b2 = 'ab';
      b3 = 'abc';
    }

    // Function is declared as pure because it neither modifies
the blockchain nor
    // reads from the storage.
    function inMemoryFixedArray() public pure {
      uint[3] memory x = { 1, 2, 3 };
    }
}
```

Below is an example of **dynamically sized array**

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of dynamically sized array
contract DynamicSizeArray {
    // Numbers is a dynamically sized array.
    // Its size changes as elements are pushed and popped.
    uint[] public numbers;
    /// Adds an element to dynamically sized array numbers.
    /// @notice Cost of the gas is constant as the
    ///         storage array is 0 initialized.
    function addElement(uint _value) public {
      numbers.push(_value);
    }

    // Returns the element at the provided index of the numbers
array.
    function getElement(uint _index) public view returns(uint)
{
      if (_index < numbers.length) {
        return numbers[_index];
      }
      return 0;
    }

    /// Removes the last element from the dynamic sized array
numbers.
    /// @notice The gas cost for removal is non constant and
depends on the
    ///         size of the element being removed.
    function popElement() public {
      numbers.pop();
    }

    // Returns the current length of the numbers array.
    // As elements are pushed, the length will increase, and
    // as elements are popped, length will decrease.
    function getNumbersLength() public view returns(uint) {
      return numbers.length;
    }

    // Function is declared as pure because it neither modifies
the blockchain nor
    // reads from the storage. It is not possible to access
push and pop functions
    // in in-memory dynamic sized arrays.
    function inMemoryDynamicArray() public pure {
      uint[] memory x = new uint[](3);
      x[0] = 1;
      x[1] = 2;
      x[2] = 3;
    }
}
```

Bytes and String

The main difference between bytes and strings is that **bytes have a length and can be converted into strings**. You cannot convert every string into bytes and get the length because strings are represented in UTF8 while bytes are not. Additionally, you can access individual elements of the bytes array using indexing. Furthermore, you can push and pop elements from bytes array.

> Tip
>
> Use bytes for arbitrary length raw byte data.

> Tip
>
> Use string for arbitrary length UTF-8 data.

> Note
>
> bytes and string are reference types and not value types.

Below is an example contract of bytes and string

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// Contract showing usage of dynamically sized array
contract BytesAndStrings {
    // b1 and s1 are used below to store the ASCII data.
    // When accessing the b1 to read data from blockchain, you
will see hex data 0x616263
    // When accessing the s1 to read data from blockchain, you
will see 'abc'
```

```
    bytes public b1 = 'abc';
    string public s1 = 'abc';

    // It is only possible to push elements to bytes array.
    function addElement() public {
      b1.push('d');
    }

    // It is only possible to index elements in bytes array.
    function getElement(uint _index) public view {
      return b1[_index];
    }

    // It is only possible to get length of a bytes array.
    function getLength() public view {
      return b1.length;
    }
}
```

Struct

A struct has the following features:

- It is a complex data type that contains a collection of key value pairs of hetrogenous elementary data types.
- It is used to represent a singular thing that has many properties such as a car, address.
- It is stored by default in storage, and if declared within a fucntion it references storage by default.
- It can be declared within a contract or outside a contract. Declaring a struct outside a contract allows it to be shared between multiple contracts.

Below is an example of struct Car

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
```

```
pragma solidity >=0.6.0 <0.9.0;

// Instructor is a struct that is compose of age, name, and
address properties.
struct Instructor {
    uint age;
    string name;
    address addr;
}

// Academy is a contract that makes use of the structure
Instructor
contract Academy {
    // academyInstructor is a variable that declared of
Instructor type
    Instructor public academyInstructor;

    // A constructor that takes in age and name as arguments
and initializes
    // the storage referenced by the academyInstructor
variable.
    constructor(uint _age, string memory _name) {
        academyInstructor.age = _age;
        academyInstructor.name = _name;
        academyInstructor.addr = msg.sender;
    }

    // changeInstructor is a function that takes in three
arguments - age, name, and address
    // and updates the currently set instructor.
    function changeInstructor(uint _age, string memory _name,
address _addr) public {
        // Here we create an in memory instance of Instructor
type.
        // Notice the syntax is <struct_name> ({ key1: value1,
...})
        Instructor memory newInstructor = Instructor ({
              age: _age,
              name: _name,
              addr: _addr
        });
        // The academyInstructor is updated from the newly
created newInstructor in memory.
```

```
        academyInstructor = newInstructor;
    }
}

// School contract making use of the Instructor struct.
contract School{
    Instructor public schoolInstructor;
}
```

Enum

A solidity enum has the following properties:

- It is a user defined type that contains a list of constants that the enum can take. At any given time, a variable of enum type can take only one of those values.
- It is convertible to and from an Integer.
- It is useful in representing the current state of flow in a smart contract.
- Enums can be declared within a contract or outside a contract. When declared outisde a contract it is accessible from multiple contracts.

```
// SPDX-License-Identifier: GPL-3.0

// pragma solidity
pragma solidity >=0.6.0 <0.9.0;

// State is an enum that can take one of three values - Open,
Closed, Unknown.
// Since it is declared outside the contract it is accessible
from multiple contracts.
enum State {
  Open,
  Closed,
  Unknown
}
```

```
// Academy is a contract that makes use of the enum State to
represent
// the current state of the academy.
contract Academy {
    // The variable academyState is initialized to State.Open
to represent
    // that the academy is open.
    State public academyState = State.Open;
}

// School is a contract that makes use of the enum State to
represent
// the current state of the school.
contract School {
    // The variable schoolState is initialized to State.Closed
to represent
    // that the school is closed.
    State public schoolState = State.Closed;
}
```

Mapping

A solidity mapping has the following properties:

- It is a collection of key value pairs of homogeneous types, that is all keys are of same type, and all values are of same type.
- It has a constant look up time in comparison to arrays which have linear look up time.
- It is not iterable.
- It is stored in storage regardless of whether the variable is declared at contract level or within a function.
- The keys are not stored in a mapping, only their hash values are stored.
- If the value of a non existing key is fetched, default value of the corresponding value type is returned.

The syntax for declaring a mapping is as follows:

```
mapping(key_type => value_type)
```

For example, `mapping(address => string)` is a mapping of keys of address type to values of string type.

Below is an example contract showing usage of mapping type.

```
//SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.5.0 <0.9.0;

contract Auction{

    // declaring a variable of type mapping
    // keys are of type address and values of type uint
    mapping(address => uint) public bids;

    // initializing the mapping variable
    // the key is the address of the account that calles the
function
    // and the value the value of wei sent when calling the
function
    function bid() payable public{
        bids[msg.sender] = msg.value;
    }
}
```

Visibility Modifiers

There are 4 visibility modifiers in solidity:

- **public** - it is the default modifier for functions. A getter is automatically created for public state variables
- **private**
- **internal** - it is the default modifier for state variables
- **external**

Below is a comparison of the visibility modifiers:

	Public	Private	Internal	External
Same Contract	✓	✓	✓	✗
Derived Contract	✓	✗	✓	✗
External Contract/DApp	✓	✗	✗	✓

Below is an example contract showing usage of public, private, internal, and external modifiers

```
//SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.5.0 <0.9.0;

contract A {
    int public x = 10;
    int y = 20; // internal by default

    function get_y() public view returns(int){
        return y;
    }

    function f1() private view returns(int){
        return x;
    }

    function f2() public view returns(int){
        int a;
        a = f1();
        return a;
    }

    function f3() internal view returns(int){
        return x;
    }
    function f4() external view returns(int){
        return x;
    }

    function f5() public pure returns(int){
        int b;
        // b = f4(); //f4() is external
        // Even though f4() wasn’t external it wouldn’t be
possible to be called
        // from within f5() because f5() is pure (it can not
read nor write to the blockchain)

        return b;
    }
}

// The contract B is a derived contract from contract A.
// The keyword 'is' is similar to languages like Java where a
class subclasses another
// class by using the keyword 'extends'.
contract B is A {
    int public xx = f3();
    // int public yy = f1(); -> f1() is private and cannot be
called from derived contracts
}

// This is a separate contract from both A and B.
contract C {
    A public contract_a = new A();
    int public xx = contract_a.f4();
    // int public y = contract_a.f1();
    // int public yy = contract_a.f3();
}
```

Error Handling

There are three ways to do exception handling in Solidity - require, assert, and try/catch.

Require

A require statement asserts the trueness of a test condition with an optional error message. On failure of the test condition, the transaction is aborted and all the remaining gas is returned to the user. Require is used for validating user inputs.

Below is an example showing error handling with require.

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

pragma solidity 0.6.12;

contract ExceptionExample {

    mapping(address => uint8) public balanceReceived;
    mapping(address => uint) public balanceReceived;

    function receiveMoney() public payable {
        balanceReceived[msg.sender] += msg.value;
    }

    function withdrawMoney(address payable _to, uint _amount)
public {
        require(_amount <= balanceReceived[msg.sender], "Not
Enough Funds, aborting");

        balanceReceived[msg.sender] -= _amount;
        _to.transfer(_amount);
    }
}
```

Assert

An assert statement also tests the trueness of a condition, with no error message. However, all the gas that is sent will be consumed if the test condition fails. Use assert for validating internal contract correctness to ensure that the contract never reaches that state, for example, divide by 0, modulo 0, integer overflow etc.

Below is an example showing usage of assert.

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

pragma solidity 0.6.12;

contract ExceptionExample {

    mapping(address => uint8) public balanceReceived;

    function receiveMoney() public payable {
        assert(msg.value == uint8(msg.value));
        balanceReceived[msg.sender] += uint8(msg.value);
        assert(balanceReceived[msg.sender] >=
uint8(msg.value));
    }

    function withdrawMoney(address payable _to, uint8 _amount)
public {
        require(_amount <= balanceReceived[msg.sender], "Not
Enough Funds, aborting");
        assert(balanceReceived[msg.sender] >=
balanceReceived[msg.sender] - _amount);
        balanceReceived[msg.sender] -= _amount;
        _to.transfer(_amount);
    }
}
```

Try/Catch

try/catch blocks are rarely used in solidity. The test condition that is expected to fail is enclosed in the try block. Based on the type of error that is being catched there are three possible flavors:

- An error is thrown by require condition failing. In this case the error is caught with the signature `catch Error(string memory reason)`, because require throws Error which has string message.

  Below is an example showing how to catch require error with try/catch blocks.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.14;

contract WillThrow {
    // This function when invoked would trigger Error with
string message
    function aRequireFunction() public pure {
        require(false, "Error message");
    }
}

contract ErrorHandling {

    event ErrorLogging(string reason);

    function catchError() public {
        WillThrow will = new WillThrow();
        try will.aRequireFunction() {
            //here we could do something if it works
        }  catch Error(string memory reason) {
            emit ErrorLogging(reason);
        }
    }
}
```

- An Panic is caused by assert condition failing. In this the error is caught with the signature `catch Panic(uint errorCode)`.

Below is an example showing how to catch Panics from assert with try/catch blocks.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.14;

contract WillThrow {
    // This function when invoked would trigger Panic.
    function aAssertFunction() public pure {
        assert(false);
    }
}
```

```
contract ErrorHandling {

    event ErrorLogging(uint errorCode);

    function catchError() public {
        WillThrow will = new WillThrow();
        try will.aAssertFunction() {
            //here we could do something if it works
        }  catch Panic(uint errorCode) {
            emit ErrorLogging(errorCode);
        }
    }
}
```

- A custom error is thrown from a callee function. In this case, it is more cumbersome, and the error is caught with signature

  Below is an example of how to catch custom errors with try/catch blocks.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.14;

contract WillThrow {
    error NotAllowedError(string);

    // This function when invoked would trigger Error with
string message
    function aCustomErrorFunction() public pure {
        revert NotAllowedError("You are not allowed here");
    }
}

contract ErrorHandling {

    event ErrorLogging(bytes reason);

    function catchError() public {
        WillThrow will = new WillThrow();
        try will.aCustomErrorFunction() {
            //here we could do something if it works
        }  catch (bytes memory reason) {
            emit ErrorLogging(reason);
        }
    }
}
```

Inheritance

- In solidity a contract is like a class. A contract can inherit from another contract known as a base contract.
- Solidity supports multiple inheritance, that is a contract can inherit from multiple contracts. On the blockchain, through, only one contract is created and all state variables, functions from all the base contracts gets copied to the resulting contract.
- Solidity supports polymorphism.
- All function calls are virtual, that is the function from the most derived contract (the contract at the bottom of inheritance chain) is called, except when the exact name is specified using the contract name.
- Contracts inherit from other contracts using the **is** keyword.

Abstract Contract

A contract is an abstract contract if one of the conditions is satisfied:

- It contains atleast one unimplemented function.
- All unimplemented function must be marked `virtual`.
    - The derived contract implementing the virtual function must use the keyword `override`.
- It has `abstract` keyword used in front of the contract name.

Note

An abstract contract cannot be deployed on the blockchain. Only a contract that implements all the functions of the abstract contract can be deployed.

Below is an example of base contract.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.5.0 <0.9.0;

// A fully implemented contract with just the keyword abstract
abstract contract A {
  int public x = 5;
  address public owner;

  constructor() {
    owner = msg.sender;
  }

  function setX(int _x) public {
    x = _x;
  }
}

// Contract B is declared as abstract because it contains an
// unimplemented function - setter function for X.
abstract contract B {
  int public x = 7;
  address public owner;

  constructor() {
    owner = msg.sender;
  }

  // An unimplemented function must always be marked virtual.
  function setX(int _x) public virtual;
}

// The contract C can be deployed on the chain as it is not
abstract
```

```
contract C is A {
  int public y = 3;
}

// The contract D inherits from contract B and implements the
virtual
// function declared in contract B.
contract D is B {
  function setX(int _x) public override {
    x = _x;
  }
}
```

Interface

An interface in solidity has the following features:

- All functions in interface must be unimplemented.
- An Interface can inherit other interfaces.
- Interface is declared with `interface` keyword.
- An Interface has other restrictions:
  - All functions must be declared `external`.
  - No state variables are allowed.
  - No constructor is allowed.
  - Inherting other contracts is not allowed.

Below is an example of interface.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.5.0 <0.9.0;

// Interface is declared with 'interface' keyword.
interface BaseInterface {
  // Function is declared unimplemented and as 'external'
```

```
    function setX(uint _x) external;
  }

  // Contract A implements the functions declared in
  'BaseInterface'.
  contract A is BaseInterface {
    uint public x;

    // The function declared in the interface is implemented.
    function setX(uint _x) override public {
      x = _x;
    }
  }
```

ERC20 Token Standard

A token represents anything of value. It can be a discount voucher, or a share. It can represent any **fungible good**.

ERC stands for **Ethereum Request for Comments**. ERC is a form of proposal and its purpose is to define standards and practices.

EIP stands for **Ethereum Improvement Proposal** and makes changes to the actual code Ethereum. ERC on the other hand is a guidance on how to use various features of Ethereum.

ERC20 is a proposal that intends to standardize how a token contract should be created, how we interact with such a token, and how these contracts interact with each other.

ERC20 is a standard interface used by applications such as wallets, decentralzied exchanges, and so on to interact with tokens.

ERC20 standard enables interoperability.

ERC20 tokens can be interacted with in the same way a standard wallet is used to send, receive, and storing Ethereum.

ERC20 contract keeps track of who owns how many tokens the same way Ethereum network keeps track of who owns how much ETH.

Tokens can be partially or fully ERC-20 complaint. A fully compliant ERC20 token must implement 6 functions and 2 events.

Below is the ERC20 interface for full compliance

```
// ----------------------------------------------------------------
----------------
// EIP-20: ERC-20 Token Standard
// https://eips.ethereum.org/EIPS/eip-20
// -----------------------------------------

interface ERC20Interface {
    function totalSupply() external view returns (uint);
    function balanceOf(address tokenOwner) external view
returns (uint balance);
    function transfer(address to, uint tokens) external returns
(bool success);

    function allowance(address tokenOwner, address spender)
external view returns (uint remaining);
    function approve(address spender, uint tokens) external
returns (bool success);
    function transferFrom(address from, address to, uint
tokens) external returns (bool success);

    event Transfer(address indexed from, address indexed to,
uint tokens);
    event Approval(address indexed tokenOwner, address indexed
spender, uint tokens);
}
```

In addition the implementing contract of ERC20 Interface should declare the following state variables with the exact same variable names:

```
// The name of the token
string public name = "ERCToken";

// The ticket of the token
string public symbol = "ERCT";

// The number of decimals to which the token is divisible.
uint public decimals = 18; //18 is very common
```

Communication between Smart Contracts