Solidity - Quick Guide

Solidity - Overview

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

Solidity is statically typed, supports inheritance, libraries and complex user-defined types programming language.

You can use Solidity to create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

What is 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.

The 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.

What is 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.

It is fine if you do not understand Smart Contract right now, we will go into more detail later.

Solidity - Environment Setup

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.

Method 1 - npm / Node.js

This is the fastest way to install Solidity compiler on your CentoS Machine. We have following steps to install Solidity Compiler −

Install Node.js

First make sure you have node.js available on your CentOS machine. If it is not available then install it using the following commands −

```
# First install epel-release
$sudo yum install epel-release

# Now install nodejs
$sudo yum install nodejs

# Next install npm (Nodejs Package Manager )
$sudo yum install npm

# Finally verify installation
$npm --version
```

If everything has been installed then you will see an output something like this −

```
3.10.10
```

Install solc

Once you have Node.js package manager installed then you can proceed to install Solidity compiler as below −

```
$sudonpm install -g solc
```

The above command will install solcjs program and will make it available globally through out the system. Now you can test your Solidity compiler by issuing following command −

```
$solcjs-version
```

If everything goes fine, then this will print something as follows −

```
0.5.2+commit.1df8f40c.Emscripten.clang
```

Now you are ready to use solcjs which has fewer features than the standard Solidity compiler but it will give you a good starting point.

Method 2 - Docker Image

You can pull a Docker image and start using it to start with Solidity programming. Following are the simple steps. Following is the command to pull a Solidity Docker Image.

```
$docker pull ethereum/solc:stable
```

Once a docker image is downloaded we can verify it using the following command.

```
$docker run ethereum/solc:stable-version
```

This will print something as follows −

```
$ docker run ethereum/solc:stable -version

solc, the solidity compiler commandlineinterfaceVersion: 0.5.2+commit.1df8f40c.Linux.g
```

Method 3: Binary Packages Installation

If you are willing to install full fledged compiler on your Linux machine, then please check official website Installing the Solidity Compiler.

Solidity - Basic Syntax

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

Let's start with a simple source file of Solidity. 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 up to, but not including, version 0.6.0.

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.

Importing Files

Though above example does not have an import statement but Solidity supports import statements that are very similar to those available in JavaScript.

The following statement imports all global symbols from "filename".

```
import "filename";
```

The following example creates a new global symbol symbolName whose members are all the global symbols from "filename".

```
import * as symbolName from "filename";
```

To import a file x from the same directory as the current file, use import "./x" as x;. If you use import "x" as x; instead, a different file could be referenced in a global "include directory".

Reserved Keywords

Following are the reserved keywords in Solidity −

Solidity - First Application

We're using Remix IDE to Compile and Run our Solidity Code base.

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

Example

```
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 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.

Output

```
0: uint256: 3
```

Solidity - Comments

Solidity supports both C-style and C++-style comments, Thus −

- Any text between a // and the end of a line is treated as a comment and is ignored by Solidity Compiler.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.

Example

The following example shows how to use comments in Solidity.

```
function getResult() public view returns(uint){
   // This is a comment. It is similar to comments in C++

   /*
      * This is a multi-line comment in solidity
      * It is very similar to comments in C Programming
   */
   uint a = 1;
   uint b = 2;
   uint result = a + b;
   return result;
}
```

Solidity - 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.

Value Types

Solidity offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

Note: You can also represent the signed and unsigned fixed-point numbers as fixedMxN/ufixedMxN where M represents the number of bits taken by type and N represents the decimal points. M should be divisible by 8 and goes from 8 to 256. N can be from 0 to 80.

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 = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Solidity - Variables

Solidity supports three types of variables.

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

- **Local Variables** − Variables whose values are present till function is executing.
- **Global Variables** − Special variables exists in the global namespace used to get information about the blockchain.

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

State Variable

Variables whose values are permanently stored in a contract storage.

```
pragma solidity ^0.5.0;
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 ^0.5.0;
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
   }
}
```

Example

```
pragma solidity ^0.5.0;
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 storedData; //access the state variable
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: 10
```

Global Variables

These are special variables which exist in global workspace and provide information about the blockchain and transaction properties.

Solidity Variable Names

While naming your variables in Solidity, keep the following rules in mind.

- You should not use any of the Solidity reserved keywords as a variable name. These keywords are mentioned in the next section. For example, break or boolean variable names are not valid.
- Solidity variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, 123test is an invalid variable name but _123test is a valid one.
- Solidity variable names are case-sensitive. For example, Name and name are two different variables.

Solidity - Variable Scope

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.

Example

```
pragma solidity ^0.5.0;
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
   }
}
```

Solidity - Operators

What is an Operator?

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. Solidity supports the following types of operators.

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

Lets have a look on all operators one by one.

Arithmetic Operators

Solidity supports the following arithmetic operators −

Assume variable A holds 10 and variable B holds 20, then −

Show Example

Comparison Operators

Solidity supports the following comparison operators −

Assume variable A holds 10 and variable B holds 20, then −

Show Example

Logical Operators

Solidity supports the following logical operators −

Assume variable A holds 10 and variable B holds 20, then −

Show Example

Bitwise Operators

Solidity supports the following bitwise operators −

Assume variable A holds 2 and variable B holds 3, then −

Show Example

Assignment Operators

Solidity supports the following assignment operators −

Show Example

Note − Same logic applies to Bitwise operators so they will become like <<=, >>=, >>=, &=, |= and ^=.

Conditional Operator (? :)

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

Show Example

Solidity - Loops

While writing a contract, you may encounter a situation where you need to perform an action over and over again. In such situations, you would need to write loop statements to reduce the number of lines.

Solidity supports all the necessary loops to ease down the pressure of programming.

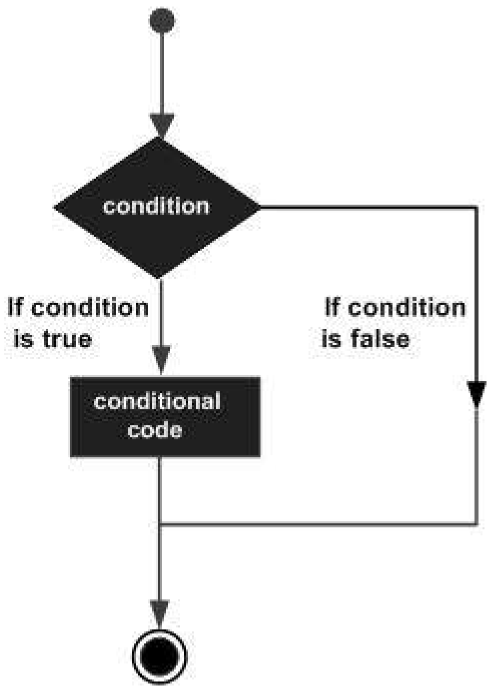
Solidity - Decision Making

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions.

Solidity supports conditional statements which are used to perform different actions based on different conditions. Here we will explain the **if..else** statement.

Flow Chart of if-else

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



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

Solidity - Strings

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 ^0.5.0;

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 ^0.5.0;

contract SolidityTest {
   bytes32 data = "test";
}
```

Escape Characters

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);
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: string: 3
```

Solidity - Arrays

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

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

```
pragma solidity ^0.5.0;

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);

      assert(a.length == 7);
      assert(b.length == len);

      //access array variable
      a[6] = 8;

      //test array variable
      assert(a[6] == 8);

      //static array
      uint[3] memory c = [uint(1) , 2, 3];
      assert(c.length == 3);
   }
}
```

Solidity - Enums

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

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

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

Example

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

```
pragma solidity ^0.5.0;

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

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

Run the above program using steps provided in Solidity First Application chapter.

First Click **setLarge** Button to set the value as LARGE then click **getChoice** to get the selected choice.

Output

```
uint8: 2
```

Click **getDefaultChoice** Button to get the default choice.

Output

```
uint256: 1
```

Solidity - Structs

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

- Title
- Author
- Subject
- Book ID

Defining a Struct

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

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

Example

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

Accessing 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

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

```
pragma solidity ^0.5.0;

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

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

Run the above program using steps provided in Solidity First Application chapter.

First Click **setBook** Button to set the value as LARGE then click **getBookId** to get the selected book id.

Output

```
uint256: 1
```

Solidity - Mapping

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

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

Where

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

Considerations

- Mapping can only have type of **storage** and are generally used for state variables.
- Mapping can be marked public. Solidity automatically create getter for it.

Example

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

```
pragma solidity ^0.5.0;
```

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

   function updateBalance(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}
contract Updater {
   function updateBalance() public returns (uint) {
      LedgerBalance ledgerBalance = new LedgerBalance();
      ledgerBalance.updateBalance(10);
      return ledgerBalance.balances(address(this));
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

First Click **updateBalance** Button to set the value as 10 then look into the logs which will show the decoded output as −

Output

```
{
  "0": "uint256: 10"
}
```

Solidity - Conversions

Solidity allows implicit as well as explicit conversion. Solidity compiler allows implicit conversion between two data types provided no implicit conversion is possible and there is no loss of information. For example uint8 is convertible to uint16 but int8 is convertible to uint256 as int8 can contain negative value not allowed in uint256.

Explicit Conversion

We can explicitly convert a data type to another using constructor syntax.

```
int8 y = -3;
uint x = uint(y);
//Now x = 0xfffff..fd == two complement representation of -3 in 256 bit formc
```

Conversion to smaller type costs higher order bits.

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b = 0x5678
```

Conversion to higher type adds padding bits to the left.

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b = 0x00001234
```

Conversion to smaller byte costs higher order data.

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b = 0x12
```

Conversion to larger byte add padding bits to the right.

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b = 0x12340000
```

Conversion between fixed size bytes and int is only possible when both are of same size.

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b = 0x00001234
uint32 c = uint32(bytes4(a)); // c = 0x12340000
uint8 d = uint8(uint16(a)); // d = 0x34
uint8 e = uint8(bytes1(a)); // e = 0x12
```

Hexadecimal numbers can be assigned to any integer type if no truncation is needed.

```
uint8 a = 12; // no error
uint32 b = 1234; // no error
uint16 c = 0x123456; // error, as truncation required to 0x3456
```

Solidity - Ether Units

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

1×10^{12}.

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

Time Units

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

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

Solidity - Special Variables

Special variables are globally available variables and provides information about the blockchain. Following is the list of special variables −

Example

Try the following code to see the use of msg, a special variable to get the sender address in Solidity.

```
pragma solidity ^0.5.0;

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

   function updateBalance(uint newBalance) public {
      balances[msg.sender] = newBalance;
   }
}
contract Updater {
   function updateBalance() public returns (uint) {
      LedgerBalance ledgerBalance = new LedgerBalance();
      ledgerBalance.updateBalance(10);
      return ledgerBalance.balances(address(this));
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

First Click **updateBalance** Button to set the value as 10 then look into the logs which will show the decoded output as −

Output

```
{
  "0": "uint256: 10"
}
```

Solidity - Style Guide

Style Guide helps to maintain code layout consistent and make code more readable. Following are the best practices following while writing contracts with Solidity.

Code Layout

- **Indentation** − Use 4 spaces instead of tab to maintain indentation level. Avoid mixing spaces with tabs.
- **Two Blank Lines Rule** − Use 2 Blank lines between two contract definitions.

```
pragma solidity ^0.5.0;

contract LedgerBalance {
   //...
}
contract Updater {
   //...
}
```

- **One Blank Line Rule** − Use 1 Blank line between two functions. In case of only declaration, no need to have blank lines.

```
pragma solidity ^0.5.0;

contract A {
   function balance() public pure;
   function account() public pure;
}
contract B is A {
   function balance() public pure {
      // ...
   }
   function account() public pure {
      // ...
   }
}
```

- **Maximum Line Length** − A single line should not cross 79 characters so that readers can easily parse the code.
- **Wrapping rules** − First argument be in new line without opening parenthesis. Use single indent per argument. Terminating element); should be the last one.

```
function_with_a_long_name(
   longArgument1,
   longArgument2,
   longArgument3
);
variable = function_with_a_long_name(
   longArgument1,
   longArgument2,
   longArgument3
);
event multipleArguments(
   address sender,
   address recipient,
   uint256 publicKey,
   uint256 amount,
   bytes32[] options
);
MultipleArguments(
   sender,
   recipient,
   publicKey,
   amount,
   options
);
```

- **Source Code Encoding** – UTF-8 or ASCII encoding is to be used preferably.
- **Imports** – Import statements should be placed at the top of the file just after pragma declaration.
- **Order of Functions** – Functions should be grouped as per their visibility.

```
pragma solidity ^0.5.0;

contract A {
   constructor() public {
      // ...
   }
   function() external {
      // ...
   }

   // External functions
   // ...

   // External view functions
   // ...

   // External pure functions
   // ...

   // Public functions
   // ...

   // Internal functions
   // ...

   // Private functions
   // ...
}
```

- **Avoid extra whitespaces** − Avoid whitespaces immediately inside parenthesis, brackets or braces.
- **Control structures** − Braces should open on same line as declaration. Close on their own line maintaining the same indentation. Use a space with opening brace.

```
pragma solidity ^0.5.0;

contract Coin {
   struct Bank {
      address owner;
      uint balance;
   }
}
if (x < 3) {
   x += 1;
} else if (x > 7) {
   x -= 1;
} else {
   x = 5;
}
if (x < 3)
   x += 1;
else
   x -= 1;
```

- **Function Declaration** − Use the above rule for braces. Always add a visibility label. Visibility label should come first before any custom modifier.

```
function kill() public onlyowner {
   selfdestruct(owner);
}
```

- **Mappings** − Avoid whitespaces while declaring mapping variables.

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

- **Variable declaration** − Avoid whitespaces while declaring array variables.

```
uint[] x;  // not unit [] x;
```

- **String declaration** − Use double quotes to declare a string instead of single quote.

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

Order of Layout

Elements should be layout in following order.

- Pragma statements
- Import statements
- Interfaces
- Libraries
- Contracts

Within Interfaces, libraries or contracts the order should be as −

- Type declarations
- State variables
- Events
- Functions

Naming conventions

- Contract and Library should be named using CapWords Style. For example, SmartContract, Owner etc.
- Contract and Library name should match their file names.
- In case of multiple contracts/libraries in a file, use name of core contract/library.

Owned.sol

```
pragma solidity ^0.5.0;

// Owned.sol
contract Owned {
   address public owner;
   constructor() public {
      owner = msg.sender;
   }
   modifier onlyOwner {
      //....
   }
   function transferOwnership(address newOwner) public onlyOwner {
      //...
   }
}
```

Congress.sol

```
pragma solidity ^0.5.0;

// Congress.sol
import "./Owned.sol";

contract Congress is Owned, TokenRecipient {
   //...
}
```

- Struct Names
  − Use CapWords Style like SmartCoin.
- Event Names
  − Use CapWords Style like Deposit, AfterTransfer.

- Function Names
  − Use mixedCase Style like initiateSupply.
- Local and State variables
  − Use mixedCase Style like creatorAddress, supply.
- Constants
  − Use all capital letters with underscore to seperate words like MAX_BLOCKS.
- Modifier Names
  − Use mixCase Style like onlyAfter.
- Enum Names
  − Use CapWords Style like TokenGroup.

Solidity - Functions

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Like any other advanced programming language, Solidity also supports all the features necessary to write modular code using functions. This section explains how to write your own functions in Solidity.

Function Definition

Before we use a function, we need to define it. The most common way to define a function in Solidity is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

Syntax

The basic syntax is shown here.

```
function function-name(parameter-list) scope returns() {
   //statements
}
```

Example

Try the following example. It defines a function called getResult that takes no parameters −

```
pragma solidity ^0.5.0;

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

Calling a Function

To invoke a function somewhere later in the Contract, you would simply need to write the name of that function as shown in the following code.

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);//access local variable
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: string: 3
```

Function Parameters

Till now, we have seen functions without parameters. But there is a facility to pass different parameters while calling a function. These passed parameters can be captured inside the function and any manipulation can be done over those parameters. A function can take multiple parameters separated by comma.

Example

Try the following example. We have used a **uint2str** function here. It takes one parameter.

```
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);//access local variable
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: string: 3
```

The return Statement

A Solidity function can have an optional **return** statement. This is required if you want to return a value from a function. This statement should be the last statement in a function.

As in above example, we are using uint2str function to return a string.

In Solidity, a function can return multiple values as well. See the example below −

```
pragma solidity ^0.5.0;

contract Test {
   function getResult() public view returns(uint product, uint sum){
      uint a = 1; // local variable
      uint b = 2;
      product = a * b;
      sum = a + b;

      //alternative return statement to return
      //multiple values
      //return(a*b, a+b);
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: product 2
1: uint256: sum 3
```

Solidity - Function Modifiers

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

First we create a modifier with or without parameter.

```
contract Owner {
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

The function body is inserted where the special symbol "_;" appears in the definition of a modifier. So if condition of modifier is satisfied while calling this function, the function is executed and otherwise, an exception is thrown.

See the example below −

```
pragma solidity ^0.5.0;

contract Owner {
   address owner;
   constructor() public {
      owner = msg.sender;
   }
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
contract Register is Owner {
   mapping (address => bool) registeredAddresses;
   uint price;
   constructor(uint initialPrice) public { price = initialPrice; }

   function register() public payable costs(price) {
      registeredAddresses[msg.sender] = true;
   }
   function changePrice(uint _price) public onlyOwner {
      price = _price;
   }
}
```

Solidity - View Functions

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.
- Using selfdestruct.
- Sending Ether via calls.
- Calling any function which is not marked view or pure.
- Using low-level calls.
- Using inline assembly containing certain opcodes.

Getter method are by default view functions.

See the example below using a view function.

Example

```
pragma solidity ^0.5.0;

contract Test {
   function getResult() public view returns(uint product, uint sum){
      uint a = 1; // local variable
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: product 2
1: uint256: sum 3
```

Solidity - Pure Functions

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.

See the example below using a view function.

Example

```
pragma solidity ^0.5.0;

contract Test {
   function getResult() public pure returns(uint product, uint sum){
      uint a = 1;
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: product 2
1: uint256: sum 3
```

Solidity - Fallback Function

Fallback function is a special function available to a contract. It has following features –

- It is called when a non-existent function is called on the contract.

- It is required to be marked external.
- It has no name.
- It has no arguments
- It can not return any thing.
- It can be defined one per contract.
- If not marked payable, it will throw exception if contract receives plain ether without data.

Following example shows the concept of a fallback function per contract.

Example

```
pragma solidity ^0.5.0;

contract Test {
   uint public x ;
   function() external { x = 1; }
}
contract Sink {
   function() external payable { }
}
contract Caller {
   function callTest(Test test) public returns (bool) {
      (bool success,) = address(test).call(abi.encodeWithSignature("nonExisti
      require(success);
      // test.x is now 1

      address payable testPayable = address(uint160(address(test)));

      // Sending ether to Test contract,
      // the transfer will fail, i.e. this returns false here.
      return (testPayable.send(2 ether));
   }
   function callSink(Sink sink) public returns (bool) {
      address payable sinkPayable = address(sink);
      return (sinkPayable.send(2 ether));
   }
}
```

Solidity - 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.

Following example shows the concept of a function overloading in Solidity.

Example

```
pragma solidity ^0.5.0;

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);
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Click callSumWithTwoArguments button first and then callSumWithThreeArguments button to see the result.

Output

```
0: uint256: 3
0: uint256: 6
```

Solidity - Mathematical Functions

Solidity provides inbuilt mathematical functions as well. Following are heavily used methods −

- **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}.

Following example shows the usage of mathematical functions in Solidity.

Example

```
pragma solidity ^0.5.0;

contract Test {
   function callAddMod() public pure returns(uint){
      return addmod(4, 5, 3);
   }
   function callMulMod() public pure returns(uint){
      return mulmod(4, 5, 3);
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Click callAddMod button first and then callMulMod button to see the result.

Output

```
0: uint256: 0
0: uint256: 2
```

Solidity - Cryptographic Functions

Solidity provides inbuilt cryptographic functions as well. Following are important methods −

- **keccak256(bytes memory) returns (bytes32)** − computes the Keccak-256 hash of the input.
- **ripemd160(bytes memory) returns (bytes20)** − compute RIPEMD-160 hash of the input.
- **sha256(bytes memory) returns (bytes32)** − computes the SHA-256 hash of the input.
- **ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)** − recover the address associated with the public key from elliptic curve signature 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.

Following example shows the usage of cryptographic function in Solidity.

Example

```
pragma solidity ^0.5.0;

contract Test {
   function callKeccak256() public pure returns(bytes32 result){
      return keccak256("ABC");
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: bytes32: result 0xe1629b9dda060bb30c7908346f6af189c16773fa148d3366701fbaa35
```

Solidity - Withdrawal Pattern

Withdrawal pattern ensures that direct transfer call is not made which poses a security threat. Following contract is showing the insecure use of transfer call to send ether.

```
pragma solidity ^0.5.0;

contract Test {
   address payable public richest;
   uint public mostSent;

   constructor() public payable {
      richest = msg.sender;
      mostSent = msg.value;
   }
   function becomeRichest() public payable returns (bool) {
      if (msg.value > mostSent) {
         // Insecure practice
         richest.transfer(msg.value);
         richest = msg.sender;
         mostSent = msg.value;
         return true;
      } else {
         return false;
      }
   }
}
```

Above contract can be rendered in unusable state by causing the richest to be a contract of failing fallback function. When fallback function fails, becomeRichest() function also fails and contract will stuck forever. To mitigate this problem, we can use Withdrawal Pattern.

In withdrawal pattern, we'll reset the pending amount before each transfer. It will ensure that only caller contract fails.

```
pragma solidity ^0.5.0;

contract Test {
   address public richest;
   uint public mostSent;

   mapping (address => uint) pendingWithdrawals;

   constructor() public payable {
      richest = msg.sender;
      mostSent = msg.value;
   }
   function becomeRichest() public payable returns (bool) {
      if (msg.value > mostSent) {
         pendingWithdrawals[richest] += msg.value;
         richest = msg.sender;
         mostSent = msg.value;
         return true;
      } else {
         return false;
      }
   }
   function withdraw() public {
      uint amount = pendingWithdrawals[msg.sender];
      pendingWithdrawals[msg.sender] = 0;
      msg.sender.transfer(amount);
   }
}
```

Solidity - Restricted Access

Restricted Access to a Contract is a common practice. By Default, a contract state is read-only unless it is specified as public.

We can restrict who can modify the contract's state or call a contract's functions using modifiers. We will create and use multiple modifiers as explained below −

- **onlyBy** − once used on a function then only the mentioned caller can call this function.
- **onlyAfter** − once used on a function then that function can be called after certain time period.
- **costs** − once used on a function then caller can call this function only if certain value is provided.

Example

```
pragma solidity ^0.5.0;

contract Test {
   address public owner = msg.sender;
   uint public creationTime = now;
   modifier onlyBy(address _account) {
      require(
         msg.sender == _account,
         "Sender not authorized."
      );
      _;
   }
   function changeOwner(address _newOwner) public onlyBy(owner) {
      owner = _newOwner;
   }
   modifier onlyAfter(uint _time) {
      require(
         now >= _time,
         "Function called too early."
      );
      _;
   }
   function disown() public onlyBy(owner) onlyAfter(creationTime + 6 weeks) {
      delete owner;
   }
   modifier costs(uint _amount) {
      require(
         msg.value >= _amount,
         "Not enough Ether provided."
      );
      _;
      if (msg.value > _amount)
         msg.sender.transfer(msg.value - _amount);
   }
   function forceOwnerChange(address _newOwner) public payable costs(200 ethe
      owner = _newOwner;
      if (uint(owner) & 0 == 1) return;
   }
}
```

Solidity - Contracts

Contract in Solidity is similar to a Class in C++. A Contract have following properties.

- **Constructor** − A special function declared with constructor keyword which will be executed once per contract and is invoked when a contract is created.

- **State Variables** − Variables per Contract to store the state of the contract.
- **Functions** − Functions per Contract which can modify the state variables to alter the state of a contract.

Visibility Quantifiers

Following are various visibility quantifiers for functions/state variables of a contract.

- **external** − External functions are meant to be called by other contracts. They cannot be used for internal call. To call external function within contract this.function_name() call is required. State variables cannot be marked as external.
- **public** − Public functions/ Variables can be used both externally and internally. For public state variable, Solidity automatically creates a getter function.
- **internal** − Internal functions/ Variables can only be used internally or by derived contracts.
- **private** − Private functions/ Variables can only be used internally and not even by derived contracts.

Example

```
pragma solidity ^0.5.0;

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

   //public state variable
   uint public info;

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

   //public function
   function updateData(uint a) public { data = a; }
   function getData() public view returns(uint) { return data; }
   function compute(uint a, uint b) internal pure returns (uint) { return a +
}
//External Contract
contract D {
   function readData() public returns(uint) {
      C c = new C();
      c.updateData(7);
      return c.getData();
   }
}
//Derived Contract
contract E is C {
   uint private result;
   C private c;

   constructor() public {
      c = new C();
   }
   function getComputedResult() public {
      result = compute(3, 5);
   }
   function getResult() public view returns(uint) { return result; }
   function getData() public view returns(uint) { return c.info(); }
}
```

Run the above program using steps provided in Solidity First Application chapter. Run various method of Contracts. For E.getComputedResult() followed by E.getResult() shows −

Output

```
0: uint256: 8
```

Solidity - Inheritance

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

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

Example

```
pragma solidity ^0.5.0;

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

   //public state variable
   uint public info;

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

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

Run the above program using steps provided in Solidity First Application chapter. Run various method of Contracts. For E.getComputedResult() followed by E.getResult() shows −

Output

```
0: uint256: 8
```

Solidity - Constructors

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

- A contract can have only one constructor.
- A constructor code is executed once when a contract is created and it is used to initialize contract state.
- After a constructor code executed, the final code is deployed to blockchain. This code include public functions and code reachable through public functions. Constructor code or any internal method used only by constructor are not included in final code.
- A constructor can be either public or internal.
- A internal constructor marks the contract as abstract.
- In case, no constructor is defined, a default constructor is present in the contract.

```
pragma solidity ^0.5.0;

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

- In case, base contract have constructor with arguments, each derived contract have to pass them.
- Base constructor can be initialized directly using following way −

```
pragma solidity ^0.5.0;

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

- Base constructor can be initialized indirectly using following way −

```
pragma solidity ^0.5.0;

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

- Direct and Indirect ways of initializing base contract constructor is not allowed.
- If derived contract is not passing argument(s) to base contract constructor then derived contract will become abstract.

Solidity - Abstract Contracts

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

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

Example

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

```
pragma solidity ^0.5.0;

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

Run the above program using steps provided in Solidity First Application chapter.

Output

```
0: uint256: 3
```

Solidity - Interfaces

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

- Interface can not have any function with implementation.
- Functions of an interface can be only of type external.

- Interface can not have constructor.
- Interface can not have state variables.
- Interface can have enum, structs which can be accessed using interface name dot notation.

Example

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

```
pragma solidity ^0.5.0;

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

Run the above program using steps provided in Solidity First Application chapter.

Note − Select Test from dropdown before clicking the deploy button.

Output

```
0: uint256: 3
```

Solidity - Libraries

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

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

Example

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

```
pragma solidity ^0.5.0;

library Search {
   function indexOf(uint[] storage self, uint value) public view returns (uir
      for (uint i = 0; i < self.length; i++) if (self[i] == value) return i;
      return uint(-1);
   }
}
contract Test {
   uint[] data;
   constructor() public {
      data.push(1);
      data.push(2);
      data.push(3);
      data.push(4);
      data.push(5);
   }
   function isValuePresent() external view returns(uint){
      uint value = 4;

      //search if value is present in the array using Library function
      uint index = Search.indexOf(data, value);
      return index;
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Note − Select Test from dropdown before clicking the deploy button.

Output

```
0: uint256: 3
```

Using For

The directive **using A for B;** can be used to attach library functions of library A to a given type B. These functions will used the caller type as their first parameter (identified using self).

Example

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

```
pragma solidity ^0.5.0;

library Search {
   function indexOf(uint[] storage self, uint value) public view returns (uir
      for (uint i = 0; i < self.length; i++)if (self[i] == value) return i;
      return uint(-1);
   }
}
contract Test {
   using Search for uint[];
   uint[] data;
   constructor() public {
      data.push(1);
      data.push(2);
      data.push(3);
      data.push(4);
      data.push(5);
   }
   function isValuePresent() external view returns(uint){
      uint value = 4;

      //Now data is representing the Library
      uint index = data.indexOf(value);
      return index;
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Note − Select Test from dropdown before clicking the deploy button.

Output

```
0: uint256: 3
```

Solidity - Assembly

Solidity provides an option to use assembly language to write inline assembly within Solidity source code. We can also write a standalone assembly code which then be converted to bytecode. Standalone Assembly is an intermediate language for a Solidity compiler and it converts the Solidity code into a Standalone Assembly and then to byte code. We can used the same language used in Inline Assembly to write code in a Standalone assembly.

Inline Assembly

Inline assembly code can be interleaved within Solidity code base to have more fine-grain control over EVM and is used especially while writing the library functions.

An assembly code is written under **assembly { ... }** block.

Example

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

```
pragma solidity ^0.5.0;

library Sum {
   function sumUsingInlineAssembly(uint[] memory _data) public pure returns (
      for (uint i = 0; i < _data.length; ++i) {
         assembly {
            o_sum := add(o_sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
         }
      }
   }
}
contract Test {
   uint[] data;
   constructor() public {
      data.push(1);
      data.push(2);
      data.push(3);
      data.push(4);
      data.push(5);
   }
   function sum() external view returns(uint){
      return Sum.sumUsingInlineAssembly(data);
   }
}
```

Run the above program using steps provided in Solidity First Application chapter.

Note − Select Test from dropdown before clicking the deploy button.

Output

```
0: uint256: 15
```

Solidity - Events

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

An event can be declared using event keyword.

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

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

Example

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

First Create a contract and emit an event.

```
pragma solidity ^0.5.0;

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

Then access the contract's event in JavaScript code.

```
var abi = /* abi as generated using compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceiptContract = ClientReceipt.at("0x1234...ab67" /* address */);

var event = clientReceiptContract.Deposit(function(error, result) {
   if (!error)console.log(result);
});
```

It should print details similar to as following −

Output

```
{
  "returnValues": {
    "_from": "0x1111...FFFFCCCC",
    "_id": "0x50...sd5adb20",
    "_value": "0x420042"
  },
  "raw": {
    "data": "0x7f...91385",
    "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
  }
}
```

Solidity - Error Handling

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

- **assert(bool condition)** − In case condition is not met, this method call causes an invalid opcode and any changes done to state got reverted. This method is to be used for internal errors.
- **require(bool condition)** − In case condition is not met, this method call reverts to original state. - This method is to be used for errors in inputs or external components.
- **require(bool condition, string memory message)** − In case condition is not met, this method call reverts to original state. - This method is to be used for errors in inputs or external components. It provides an option to provide a custom message.
- **revert()** − This method aborts the execution and revert any changes done to the state.
- **revert(string memory reason)** − This method aborts the execution and revert any changes done to the state. It provides an option to provide a custom message.

Example

Try the following code to understand how error handling works in Solidity.

```
pragma solidity ^0.5.0;

contract Vendor {
   address public seller;
   modifier onlySeller() {
      require(
         msg.sender == seller,
         "Only seller can call this."
      );
      _;
   }
   function sell(uint amount) public payable onlySeller {
      if (amount > msg.value / 2 ether)
         revert("Not enough Ether provided.");
      // Perform the sell operation.
   }
}
```

When revert is called, it will return the hexadecimal data as followed.

Output

```
0x08c379a0                     // Function selector for Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Dat
0x000000000000000000000000000000000000000000000000000000000000001a // Str
0x4e6f7420656e6f7567682045746865722070726f76696465642e000000000000 // Strin
```