**Assignment 12**

**Name**: Bhavin Patil

**Roll No:** 66

**TY-CS-D**

**Implement and demonstrate the use of the following in Solidity:**

**Variable, Operators, Loops, Decision Making, Strings, Arrays, Enums, Structs**

**Variables**

A **Variable** is basically a placeholder for the data which can be manipulated at runtime. Variables allow users to retrieve and change the stored information.

### **Rules For Naming Variables**

* A variable name should not match with reserved keywords.
* Variable names must start with a letter or an underscore (\_), and may contain letters from “a to z” or “A to Z” or digits from “0 to 9” and characters also.
* The name of variables are case sensitive.

### **Types of Variables**

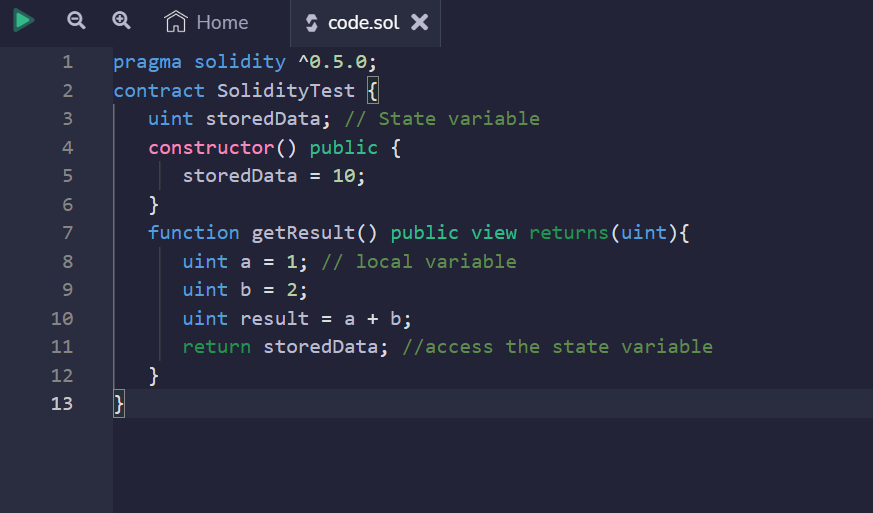
### Solidity supports three types of variables:

**1. State Variables:** Values of these variables are permanently stored in the contract storage. Each function has its own scope, and state variables should always be defined outside of that scope.

**2. Local Variable:** Values of these variables are present till the function executes and it cannot be accessed outside that function. This type of variable is usually used to store temporary values.

**3. Global Variables:**These are some special variables that can be used globally and give information about the transactions and blockChain properties. Some of the global variables are listed below:

**Code**



**Output:**

0: uint256: 10

# Operators

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

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

### **Arithmetic Operators**

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

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

### **Relational Operators**

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

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

### **Logical Operators**

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

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

### **Bitwise Operators**

These operators work at a bit level used to perform bit-level operations. Solidity supports the following arithmetic operators :

| Operator | Denotation | Description |
| --- | --- | --- |
| Bitwise AND | & | Performs boolean AND operation on each bit of integer argument |
| BitWise OR | | | Performs boolean OR operation on each bit of integer argument |
| Bitwise XOR | ^ | Performs boolean exclusive OR operation on each bit of integer argument |
| Bitwise Not | ~ | Performs boolean NOT operation on each bit of integer argument |
| Left Shift | << | Moves all bits of the first operand to the left by the number of places specified by the second operand |
| Right Shift | >> | Moves all bits of the first operand to the right by the number of places specified by the second operand |

### **Assignment Operator**

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

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

### **Conditional Operators**

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

**Syntax:**

if condition true ? then A: else B

**Code**

// Solidity contract to demonstrate

// Arithmetic Operator

pragma solidity ^0.5.0;

// Creating a contract

contract SolidityTest {

    // Initializing variables

    uint16 public a = 20;

    uint16 public b = 10;

    // Initializing a variable

    // with sum

    uint public sum = a + b;

    // Initializing a variable

    // with the difference

    uint public diff = a - b;

    // Initializing a variable

    // with product

    uint public mul = a \* b;

    // Initializing a variable

    // with quotient

    uint public div = a / b;

    // Initializing a variable

    // with modulus

    uint public mod = a % b;

    // Initializing a variable

    // decrement value

    uint public dec = --b;

    // Initializing a variable

    // with increment value

    uint public inc = ++a;

}

**Output**



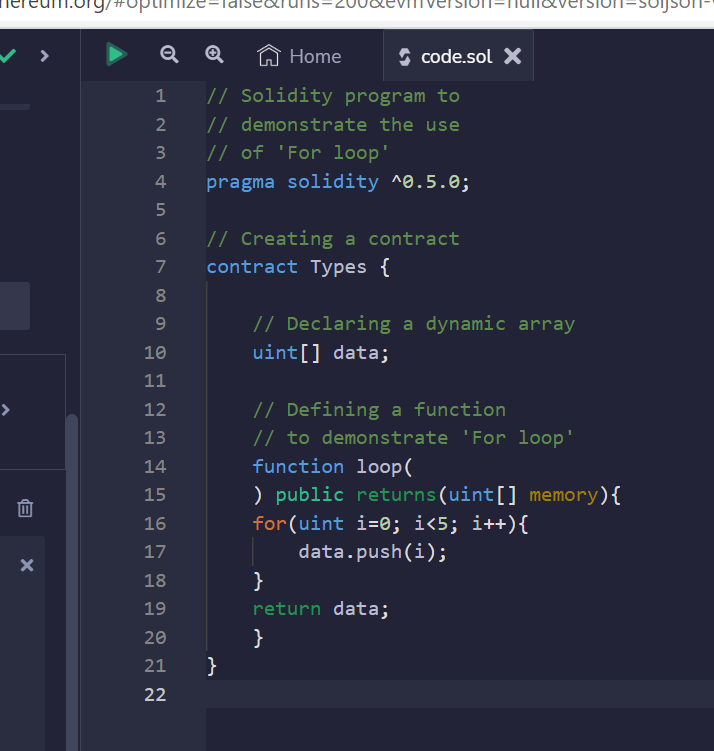
**Loops**

Loops are used when we have to perform an action over and over again. While writing a contract there may be a situation when we have to do some action repeatedly, In this situation, loops are implemented to reduce the number of lines of the statements.

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

|  |  |
| --- | --- |
| **Sr.No** | **Loops & Description** |
| 1 | [While Loop](https://www.tutorialspoint.com/solidity/solidity_while_loops.htm)  The most basic loop in Solidity is the while loop which would be discussed in this chapter. |
| 2 | [do...while Loop](https://www.tutorialspoint.com/solidity/solidity_do_while_loop.htm)  The do...while loop is similar to the while loop except that the condition check happens at the end of the loop. |
| 3 | [For Loop](https://www.tutorialspoint.com/solidity/solidity_for_while_loop.htm)  The for loop is the most compact form of looping. It includes the following three important parts. |
| 4 | [Loop Control](https://www.tutorialspoint.com/solidity/solidity_loop_control.htm)  Solidity provides full control to handle loops and switch statements. |

**Code**



**Output**



**Decision Making**

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

## **if-else**

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

|  |  |
| --- | --- |
| **Sr.No** | **Statements & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/solidity/solidity_if_statement.htm)  The if statement is the fundamental control statement that allows Solidity to make decisions and execute statements conditionally. |
| 2 | [if...else statement](https://www.tutorialspoint.com/solidity/solidity_if_else_statement.htm)  The 'if...else' statement is the next form of control statement that allows Solidity to execute statements in a more controlled way. |
| 3 | [if...else if... statement.](https://www.tutorialspoint.com/solidity/solidity_if_else_if_statement.htm)  The if...else if... statement is an advanced form of if...else that allows Solidity to make a correct decision out of several conditions. |

Code

pragma solidity ^0.5.0;

contract SolidityTest {

   uint storedData; // State variable

   constructor() public {

      storedData = 10;

   }

   function getResult() public view returns(string memory) {

      uint a = 1;

      uint b = 2;

      uint c = 3;

      uint result

      if( a > b && a > c) {   // if else statement

         result = a;

      } else if( b > a && b > c ){

         result = b;

      } else {

         result = c;

      }

      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

   }

}

**Output**

0: string: 3

**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.

## **Escape Characters**

|  |  |
| --- | --- |
| **Sr.No.** | **Character & Description** |
| 1 | **\n**  Starts a new line. |
| 2 | **\\**  Backslash |
| 3 | **\'**  Single Quote |
| 4 | **\"**  Double Quote |
| 5 | **\b**  Backspace |
| 6 | **\f**  Form Feed |
| 7 | **\r**  Carriage Return |
| 8 | **\t**  Tab |

**Code**

pragma solidity ^0.5.0;

contract SolidityTest {

   uint storedData;

   constructor() public {

      storedData = 10;

   }

   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) {   // if statement

         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

   }

}

## **Output**

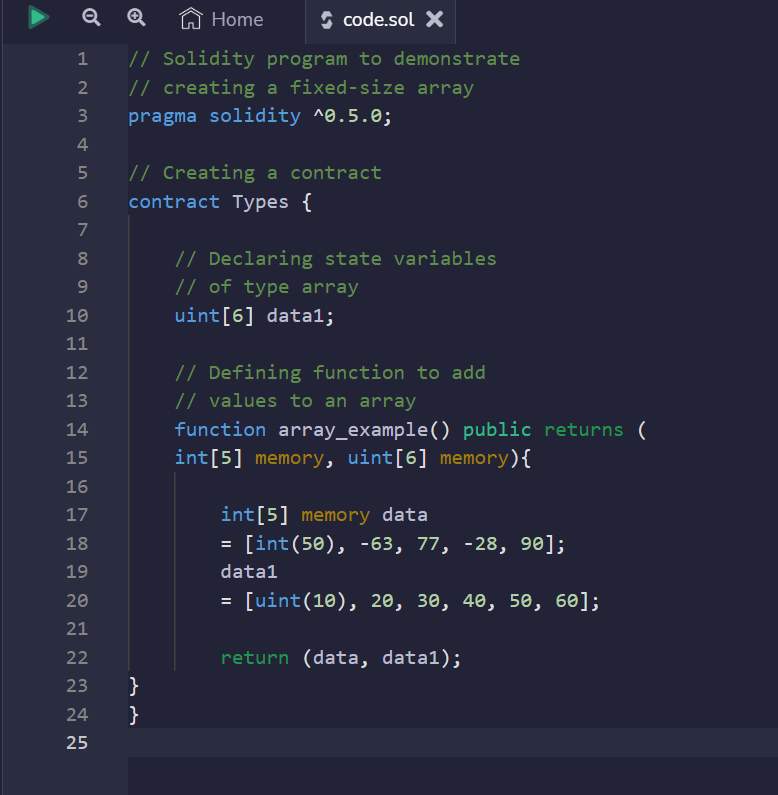
0: string: 3

**Arrays**

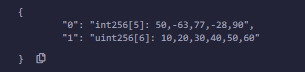
Array is a data structure, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data,

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 cannot 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.



Output



**Enums**

Enums are the way of creating user-defined data types, it is usually used to provide names for integral constants which makes the contract better for maintenance and reading. Enums restrict the variable with one of a few predefined values, these values of the enumerated list are called *enums*. Options are represented with integer values starting from zero, a default value can also be given for the enum. By using enums it is possible to reduce the bugs in the code.

// Solidity program to demonstrate

// how to use 'enumerator'

pragma solidity ^0.5.0;

// Creating a contract

contract Types {

    // Creating an enumerator

    enum week\_days

    {

    Monday,

    Tuesday,

    Wednesday,

    Thursday,

    Friday,

    Saturday,

    Sunday

    }

    // Declaring variables of

    // type enumerator

    week\_days week;

    week\_days choice;

    // Setting a default value

    week\_days constant default\_value

    = week\_days.Sunday;

    // Defining a function to

    // set value of choice

    function set\_value() public {

    choice = week\_days.Thursday;

    }

    // Defining a function to

    // return value of choice

    function get\_choice(

    ) public view returns (week\_days) {

    return choice;

    }

    // Defining function to

    // return default value

    function getdefaultvalue(

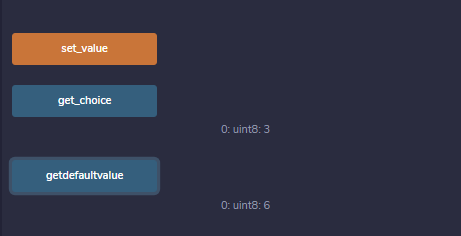
    ) public pure returns(week\_days) {

        return default\_value;

    }

}

**Output**



### **Struct**

Structs in Solidity allows you to create more complicated data types that have multiple properties. You can define your own type by creating a **struct**.

They are useful for grouping together related data.

Structs can be declared outside of a contract and imported in another contract. Generally, it is used to represent a record. To define a structure, *struct* keyword is used, which creates a new data type.

**Code**

// Solidity program to demonstrate

// how to use 'structures'

pragma solidity ^0.5.0;

// Creating a contract

contract test {

// Declaring a structure

struct Book {

    string name;

    string writter;

    uint id;

    bool available;

}

// Declaring a structure object

Book book1;

// Assigning values to the fields

// for the structure object book2

Book book2

    = Book("Building Ethereum DApps",

            "Roberto Infante ",

            2, false);

// Defining a function to set values

// for the fields for structure book1

function set\_book\_detail() public {

    book1 = Book("Introducing Ethereum and Solidity",

                "Chris Dannen",

                    1, true);

}

// Defining function to print

// book2 details

function book\_info(

)public view returns (

    string memory, string memory, uint, bool) {

        return(book2.name, book2.writter,

            book2.id, book2.available);

    }

// Defining function to print

// book1 details

function get\_details(

) public view returns (string memory, uint) {

    return (book1.name, book1.id);

}

}

**Output**

