Solidity – Exercises : Functions

Parameters

The following code snippets[1] show the following multiple functions, each with different constructs for parameters and return values.

1. The first function, *singleIncomingParameter*, accepts one parameter named _data of type int and returns a single **return** value that is identified using _output of type int.

   The function signature provides constructs to define both the incoming parameters and return values. The **return** keyword in the function signature helps define the return types from the function.

   In the following code snippet, the **return** keyword within the function code automatically maps to the first return type declared in the function signature.

```
function singleIncomingParameter(int _data) returns (int _output) {
        return _data * 2;
}
```

2. The second function, *multipleIncomingParameter*, accepts two parameters: *_data* and *_data2*, which are both of type int and return a single **return** value identified using *_output* of type int, as follows:

```
function multipleIncomingParameter(int _data, int _data2) returns (int _output) {
        return _data * _data2;
}
```

3. The third function, *multipleOutgoingParameter*, accepts one parameter, _data, of type int and returns two return values identified using square and half, which are both of type int.

   In the following code snippet, returning multiple parameters is something unique to Solidity and is not found in many programming languages:

```
function multipleOutgoingParameter(int _data) returns (int square, int half)
{
        square = _data * _data;
        half = _data /2 ;
}
```

[1] (source: Ritesh Modi, Solidity Programming Essentials, Packt)

4. The fourth function, *multipleOutgoingTuple*, is similar to the third function mentioned previously. However, instead of assigning return values as separate statements and variables, it returns values as a tuple.

   A tuple is a custom data structure consisting of multiple variables, as shown in the following code snippet:

```
function multipleOutgoingTuple(int _data) returns (int square, int half)
{
        (square, half) = (_data * _data,_data /2 );
}
```

   Key in the entire code and **fix** some of the errors. Test it in Remix in the javascript environment.

```
contract  cParameters {

        function singleIncomingParameter(int _data) returns (int _output) {
                return _data * 2;
        }

        function multipleIncomingParameter(int _data, int _data2) returns (int _output)
        {
                return _data * _data2;
        }
        function multipleOutgoingParameter(int _data) returns (int square, int half)
        {
                square = _data * _data;
                half = _data /2 ;
        }
        function multipleOutgoingTuple(int _data) returns (int square, int half)
        {
                (square, half) = (_data * _data,_data /2 );
        }
}
```

   Test the Inputs and Outputs of the function in the above contract. Ensure that you understand what is happening.

Modifiers

Modifiers are special functions that change the behavior of a function. Here, the function code remains the same, but the execution path of a function changes. Modifiers can only be applied to functions.

```
contract  cModifier {
    address owner;
    int public mydata;

    constructor(){
        owner = msg.sender;
    }

  modifier isOwner{
        if(msg.sender==owner){
            _;
        }
  }

  function doDoubleMod(int _data) public isOwner {
        mydata=_data*2;
  }

 function doDouble(int _data) public {
       if(msg.sender==owner){
           mydata=_data*2;
       }
 }
}
```

The contract shown here has: a constructor, two state variables, and a function. It also has an additional special function that is defined using the modifier keyword.

These functions do not use the if condition to check whether the caller of the function is the same as the account that deployed the contract; instead, these functions are decorated with the modifier name in their signature.

Modifiers are defined using the *modifier* keyword and an identifier. The code for modifier is placed within curly brackets. The code within a modifier can validate the incoming value and can conditionally execute the called function after evaluation.

The _ identifier is of special importance here—its purpose is to replace itself with the function code that is invoked by the caller.

When a caller calls the *doDoubleMod* function, which is decorated with the isOwner modifier, the **modifier** takes control of the execution and replaces the _ identifier with the called function code, that is, *doDoubleMod*.
The functionality can be written without the *modifier* as in the next function *doDouble*(). If the same restriction is required we would have to repeat the same code segment. However the same *modifier* can be applied to multiple functions, we can reuse the same modifier.

This helps in writing cleaner, more readable, and more maintainable code. Developers do not have to keep repeating the same code in every function or check for the incoming value when executing a function.

Run and test the above contract. How would you test the above contract to test a non-contract owner ability to run the function?

Require

```
contract requireContract{
      function validInt8(uint _data) pure public returns (uint8){
         require(_data >=0);
         require(_data <=255);
         return uint8(_data);
      }

      function isEven(uint _data) pure public returns(bool){
          require(_data % 2 ==0);
          return true;
      }
}
```

The **require** statement should be used for validating all arguments and values that are incoming to the function. This means that if another function from another contract or function in the same contract is called, the incoming value should also be checked using the **require** function.

The **require** function should be used to check the current state of variables before they are used. If require throws an exception, it should mean that the values passed to the function were not expected by the function and that the caller should modify the value before sending it to a contract.

Run and test the above contract.

Assert

The **assert** statement has a similar syntax to the require statement. If it accepts a statement, that should then evaluate to either a true or false value. Based on that, the execution will either move on to the next statement or throw an exception.

The unused gas is not returned to the caller and instead the entire gas supply is consumed by **assert**. The state is reversed to original. The **assert** function results in invalid opcode, which is responsible for reverting the state and consuming all gas.

While **require** should be used for values coming from the outside, **assert** should be used for validating the current state and condition of the function and contract before execution.

Think of **assert** as working with runtime exceptions that you cannot predict.

The **assert** statement should be used when you think that a current state has become inconsistent and that execution should not continue.

The function below uses an **assert** statement to prevent an overflow exception when two numbers are added.

```
contract requireContract{
      function validInt8(uint _data) pure public returns (uint8){
         require(_data >=0);
         require(_data <=255);

         uint8 value=20;
         assert(value+_data<=255); //prevent unexpected overflow

         return uint8(_data + value);
      }

      function isEven(uint _data) pure public returns(bool){
          require(_data % 2 ==0);
          return true;
      }
}
```

Run and test the above contract.

Revert

The **revert** statement is very similar to the **require** function. However, it does not evaluate any statement and does not depend on any state or statements. Hitting a **revert** statement means an exception is thrown, along with the return of unused gas, and reverts to its original state.

```
contract revertContract{
      function validInt8(int _data) public pure returns (string memory){
        if(_data<0 || _data > 255) {
          revert();
        }
        return "valid";
      }
}
```

Run and test the above contract.

Exercise: Writing Contracts

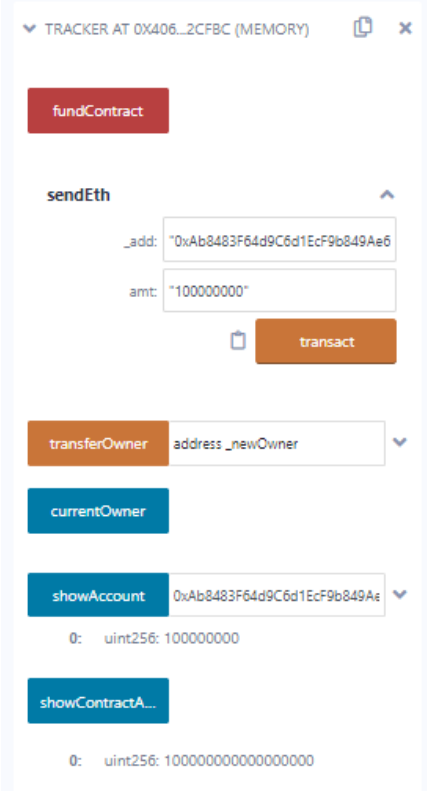
Write a contract that keeps track of the amount that is sent from the contract to various addresses. To keep track of the amount sent you need to create a mapping of address to uint256.

The contract has the following functions:

1. *sendEth*: A function to eth to another address that can only be called by the contract owner. Refer to previous activity on how to transfer Ether: use the solidity transfer function.
   Each amount sent is updated in the state mapping variable (mentioned above).

2. *transferOwner*: A transfer ownership of contract function that can only be called by the contract owner.
3. *currentOwner*: Display the address of the current owner.
4. *showAccount*: Another function that returns the amount last transfer by address.
5. *fundContract*: Before you can send any Eth the contract must have some funds. This function allows the owner to send some Eth to the contract.
6. *showContractAmt*: Display the amount held by the contract.

To check the ownership you should create a modifier that can be used on both of the above function.



Test your contract. You can just the Javascript VM environment.

1. Check that owner restricted functions are indeed restricted to owner only. You need to remember the account you use to deploy the contract - that is the contract owner.
2. Check that you can transfer Eth to the contract using the *fundContract* function by only the owner. Remember to fill the *value* field in Remix under *Deploy and Run* before you transact this function.
3. Check that Eth transfer are done corrected to different accounts.
4. Check that after the transfer the account tracking by the contract is correct.
5. Repeat 2-3.
6. Check the ownership transfer is correct. Then repeat 1-3 checks.