LAB 21-08-2023

1 Creating Contracts from a Contract

1.1 Introduction

Creating new contracts from within a contract in Solidity is a powerful feature that allows for code modularity and reusability.

With the '**new'** keyword, a contract can instantiate and deploy another contract on the blockchain. This functionality provides flexibility and enables dynamic contract creation based on specific conditions or requirements.

When creating a new contract using the '**new'** keyword, the complete code of the contract to be instantiated must be known at the compile time of the calling contract.

1.2 Syntax

The syntax for creating a new contract instance is straightforward. It involves specifying the contract name, assigning it to a variable, and providing any necessary constructor arguments in parentheses, **if applicable.**

For example:

```
ContractName variableName = new ContractName(arguments);
```

1.3 How does the 'new' keyword work?

Upon using the new keyword, the following actions occur:

- An instance of the new contract is created.
- The new contract is deployed on the blockchain.
- The **state variables** of the new contract are initialized.
- The **constructor** of the new contract is executed, allowing for additional setup.
- The **nonce** value is incremented by one.

It is important to note that creating new contracts from within a contract involves certain risks. The called contract has authority and can execute functions that may impact the calling contract or interact with other contracts.

TASK

Deploy and observe the following smart contracts.

```
contract Student {

    string private stdName;

    uint private stdRollNum;

    // setting values of state variables

    constructor (string memory name,uint rollNum)    {

        stdName = name;

        stdRollNum = rollNum;                         }

}

contract StudentsList {

    // array to store students ( composition )

    Student[] public students;

    constructor () {

        Student newStudent = new Student("Burak", 20);

        students.push(newStudent);                      }

}
```

TASK

Develop a **Hospital Management System** using Solidity. The system should allow the creation of patient records and store them in a list. Each patient record should contain their **'name'**, **'age'** and **'disease'**. The system should provide a way to **initialize** the patient records and **retrieve** the details of the stored patients.

Hint:

1) Implement Single Level Inheritance from **'Person'** to **'Patient'.**
2) Use following statement to initialize new students

```
    Patient newPatient = new Patient("Bob", 20, "Fever" );
```

2 Try/Catch

2.1 Introduction

In Solidity, the '**try/catch'** statement provides a way to **handle exceptions** and react to failed **external** function calls and contract creations.

This feature was introduced in Solidity version **0.6.0**, significantly enhancing the error handling capabilities of the language.

2.2 Syntax

1) Basic Syntax

The basic syntax of try/catch in Solidity is as follows:

```
try externalContract.demo() {
    // if the call succeeds, execute some code
} catch {
    // if the call fails, execute some code
}
```

Here, **'externalContract.demo()'** represents an external function call. The code within the try block will run if the function call succeeds, while the code within the catch block will execute if the function call fails.

You can also use **'this.demo()'** instead of 'externalContract.demo()' to invoke an external call within the same contract.

However, it **cannot be used in the constructor** because the contract has not been created at that time.

2) Return Value

If the called function has a return value, you can declare returns(returnType val) after the try keyword. This allows you to capture the returned value and utilize it within the try block. When creating contracts, the returned value represents the newly created contract instance.

```
      try externalContract.f() returns(returnType val) {
       // if the call succeeds, run some code
      } catch {
      // if the call fails, run some code
      }
```

3) Specific Exception Causes

The catch block also supports catching specific exception causes.

- ➔ You can use the '**Error**(string memory reason)' syntax to catch exceptions thrown by **revert**("reasonString") and **require**(false, "reasonString").
- ➔ Additionally, you can use **'Panic**(uint errorCode)' to catch errors caused by **assertions**, **overflows**, division by zero, and array **access out of bounds**.
- ➔ If none of the above exceptions match, you can use '**catch (bytes memory lowLevelData)'** to catch other revert errors, including custom type errors.

```
try externalContract.f() returns(returnType) {
    // if the call succeeds, run some code
} catch Error(string memory /*reason*/) {
    // catch revert("reasonString") and require(false, "reasonString")
} catch Panic(uint /*errorCode*/) {
    // Catch errors caused by Panic, such as assert failures,
overflows, division by zero, array access out of bounds
} catch (bytes memory /*lowLevelData*/) {
    // If a revert occurs and the above two exception types fail to
match, it will go into this branch
    // such as revert(), require(false), revert a custom type error
}
```

Important

It's important to note that the ‘try/catch’ statement can only handle errors from external function calls and contract creations. It cannot be used for **internal function calls** within the same contract.

It's essential to carefully handle exceptions and consider the potential impact on state changes, as changes within the called function will still be rolled back while those in the calling function won't.

TASK

1) Observe and deploy the following smart contracts.
2) Try **execute(0), execute(1),** and **execute(2)**, and explain the difference in output **(logs)**, if.
3) Try **executeNew(0)**, **executeNew(1)** and **executeNew(2)**, and explain the difference in output **(logs)**, if.
4) Keyword ‘**new’** has been used twice. Highlight them and explain the purpose of using it.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
```

```
contract PrimeChecker{
    constructor(uint a){
        require(a != 0, "Invalid Entry");
        assert(a != 1);
    }
    function isPrime(uint256 b) internal pure returns (bool) {
    if (b <= 1) {
        return false;  }
    if (b == 2) {
        return true;   }
    if (b % 2 == 0) {
        return false;  }
    for (uint256 i = 3; i * i <= b; i += 2) {
        if (b % i == 0) {
            return false;    }
    }
    return true;
}

function onlyPrime(uint256 b) external pure returns (bool success) {
    // revert when a non-prime number is entered
    require(isPrime(b), "Ups! Reverting");
    success = true;                           }
}
contract TryCatch {
    // success event
    event SuccessEvent();
    // failure event
    event CatchEvent(string message);
    event CatchByte(bytes data);

    PrimeChecker even;
    constructor() {
        even = new PrimeChecker(2);
    }

    function execute(uint amount) external returns (bool success) {
        try even.onlyPrime(amount) returns(bool _success){
            emit SuccessEvent();
            return _success;
        } catch Error(string memory reason){
            // if call fails
            emit CatchEvent(reason);
        }
    }
    function executeNew(uint a) external returns (bool success) {
        try new PrimeChecker(a) returns(PrimeChecker _even){
            // if call succeeds
            emit SuccessEvent();
            success = _even.onlyPrime(a);
        } catch Error(string memory reason) {
            emit CatchEvent(reason);
        }
         catch (bytes memory reason) {
            emit CatchByte(reason);
        }
    }
}
```

TASK

Develop a Solidity smart contract that includes two contracts: **SimpleDivision** and **TryCatchExample**. The **SimpleDivision** contract provides a function **Divide2Nums** for dividing two input numbers.

The **TryCatchExample** contract interacts with **SimpleDivision** and implements error handling using the try-catch mechanism.

3 Library

3.1 Introduction

Libraries in Solidity play a crucial role in enhancing code reusability and reducing gas consumption.

Similar to contracts, libraries consist of functions that can be **called by other contracts.**

They are designed to provide a collection of useful functions created by experts or project teams, allowing developers to leverage existing code and build upon it.

3.2 Basic Syntax

```
library <libraryName> {

    // block of code   }
```

3.3 Key Characteristics of Libraries:

1) **Non-Stateful:**

➔ Libraries in Solidity do not allow the declaration of state variables. They are assumed to be **stateless** and are unable to modify the state of a contract.

➔ This restriction ensures that libraries focus on performing basic operations based on inputs and outputs without introducing complexities associated with state management.

2) **No Inheritance:**

➔ Unlike regular contracts, libraries cannot inherit from other contracts or be inherited by other contracts.

➔ This constraint ensures that the functionality of libraries **remains isolated** and independent, promoting modularity and reusability.

3) **No Ether Transactions:**

➔ Libraries cannot receive or send ether.

➔ This limitation is imposed to maintain the purity of their functions and prevent unintended side effects or security vulnerabilities related to handling and transferring funds.

3.4 Advantages and Benefits:

1) **Code Reusability:** Libraries enable developers to reuse existing code by providing a set of well-defined functions. This promotes efficiency and reduces redundancy, as developers can rely on tested and reliable code for common operations.

2) **Gas Optimization:** Deploying common code as libraries reduces the gas cost associated with deploying multiple copies of the same functionality across different contracts. Libraries provide a way to centralize commonly used code, leading to more economical and optimized contract deployments.

3) **Modular Development:** Libraries encourage modular development practices by separating reusable logic into standalone components. This modular approach enhances code maintainability, readability, and scalability, enabling developers to focus on specific aspects of contract functionality.

3.5 Best Practices for Using Libraries:

★ **Use the "using for" Statement:**

The **"using for"** statement allows attaching library functions to a specific type, making them accessible as if they were members of that type.

This provides a convenient and intuitive way to leverage library functions. Remember to pass the relevant variable as the first parameter when calling the library function.

```
<libraryName> for <dataType>
```

★ **Direct Invocation:**

Libraries can be called directly by using the library contract's name followed by the function name.

This method is suitable when using library functions that do not require a specific type context.

```
      return Explibrary.exponential(firstVal, secondVal);
```

TASK: Contract vs. Library

1) There are some errors in the following code, correct them.

2) Deploy and observe the updated version of code.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

library Explibrary {
    function exponential( uint256 a, uint256 b) public pure returns
(uint256) {
        return a ** b ;
    }
}

contract SumContract {
    function sum( uint256 a, uint256 b) public pure returns (uint256) {
        return a + b ;
    }
}
contract LibraryClient {

     function GetExponential(uint256 firstVal, uint256 secondVal)
public pure returns(uint256) {
        return Explibrary.exponential(firstVal, secondVal);
     }
```

```
      function Sum(uint256 firstVal, uint256 secondVal) public pure
returns(uint256) {
        return SumContract.sum(firstVal, secondVal);
     }
}
```

Hint:

```
 SumContract public foo = new SumContract();
```

TASK

1) Make a **library** for finding Sum, Difference, Power functions.
2) Make a separate contract and utilize the functions available in the library.

⇒ Handle possible Errors gently !

4 Import

4.1 Introduction

Importing source code files in Solidity using the **'import'** keyword allows for modular development and reusability of code.

Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on).

At a global level, you can use import statements of the following form:

```
import "filename";
```

The filename part is called an **'import path.'**

4.2 Several Ways

Solidity provides several ways to import contracts and global symbols:

1) **Local Imports**: Solidity supports importing local files relative to the current file's location. You can import files in the same folder or in different folders using the relative path.

   For example:

   ```
   // Import from the same folder
   ```

```
import "./MyContract.sol";

// Import from a different folder

import "../Utils/Helper.sol";
```

2) **Import Through GitHub** (Works only in Remix): Solidity in Remix IDE allows importing contracts directly from GitHub repositories. You can copy the GitHub URL pointing to the contract and use it in the import statement.

   For example:

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/v2.5.0/contracts/math/SafeMath.sol";
```

3) **Import via NPM Directory:** Solidity also supports importing contracts from the NPM directory structure. You can install Solidity contracts using npm and import them using the @ symbol followed by the package name and the contract path.

   For example:

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

This method is commonly used when utilizing external libraries or frameworks.

4) **Import Specific Global Symbols:** Solidity allows importing specific global symbols from a contract instead of importing all global symbols. You can specify the symbols you want to import using curly braces {}.

   For example:

```
import { MySymbol1, MySymbol2 } from "./MyContract.sol";
```

This way, you can selectively import only the required symbols from a contract, reducing gas costs and improving code clarity.

Note:

It's important to note that the import statements should be placed after declaring the Solidity version and before the rest of the code. This ensures that the imported symbols are available in the current **global scope**.

TASK

Observe and deploy the following smart contracts.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Callee {
    string public name = "I'm getting called";
}
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

// import Foo.sol from current directory
import "./Callee.sol";

contract Import {
    Callee public demo = new Callee();

    // Test Callee.sol by getting it's name.
    function getdemoName() public view returns (string memory) {
        return demo.name();
    }
}
```

TASK

In the last task of **Library** , Make separate files for **Explibrary** and **SumContract,** and import in the main contract.

5 ABI Encode

5.1 Introduction

- ABI(**Application Binary Interface**) is the standard for interacting with Ethereum smart contracts. Data is encoded based on its type, and because the encoded result doesn't contain type information, it is necessary to indicate their types when decoding.

- The encoding on how to perform such encoding is defined by the implementation of the Ethereum Virtual Machine (EVM).
- In Solidity, the **abi.encode()** function is a powerful tool that enables the serialization and deserialization of data structures, facilitating data encoding for function calls and other communication between systems.
- ABI encoding ensures consistency and machine-readability of data structures across different systems and programming languages.

5.2 Encoding Parameters

1) **abi.encode():**

★ The abi.encode() function in Solidity allows for the encoding of function parameters or data structures using the Application Binary Interface (ABI) encoding.

★ It takes the input parameters of a function or data structure and returns a byte array representing the encoded data.

★ ABI is designed to interact with smart contracts by filling each parameter with **32 bytes** data and splicing them together.

```
function encode(address _address, uint data)

    public pure returns (bytes memory) {

            return (abi.encode(_address, data));

}
```

2) **abi.encodePacked():**

★ The abi.encodePacked() function concatenates and tightly packs multiple values of different types into a single byte array.

★ It creates a byte array that represents the tightly packed data.

★ This function is useful when you want to combine values without any padding. It is similar to abi.encode, but omits a lot of 0 filled in.

★ You can use abi.encodePacked when you want to save space and don't interact with contracts. For example when computing hash of some data.

```
 function encodePacked(address _address, uint data) public pure
returns(bytes memory result) {
        result = abi.encodePacked(_address, data);

    }
```

3) **abi.encodeWithSelector():**

★ When calling a function in Solidity, the function's selector is required. The selector is a hash of the function's name and input parameter types. The abi.encodeWithSelector() function encodes function calls along with the function selector.

★ It takes the function selector and input parameters as arguments, returning a byte array representing the encoded function call. This is particularly useful when interacting with contracts.

```
   function encodeWithSelector(address _address, uint data)

        public pure returns(bytes memory result) {

        result =
abi.encodeWithSelector(bytes4(keccak256("encodeWithSelector(addre
ss, uint )")),_address, data);

    }
```

4) **abi.encodeWithSignature():**

★ Similar to abi.encodeWithSelector(), the abi.encodeWithSignature() function encodes function calls but uses the function's signature instead of the selector.

★ The signature is a hash of the function's name and input parameter types.

★ It almost equal to encodeWithSelector - use whatever fits best.

```
function encodeWithSignature(address _address, uint data) public
pure returns(bytes memory result) {

        result =
abi.encodeWithSignature(("encodeWithSelector(address, uint
)"),_address, data);

    }
```

Conclusion:

The abi.encode() function in Solidity provides a convenient way to encode and serialize data structures, ensuring consistency and compatibility between different systems.

By understanding how to use abi.encode() and its related functions such as abi.encodePacked(), abi.encodeWithSelector(), and abi.encodeWithSignature(), you can effectively encode function parameters and data structures for various purposes, including function calls, serialization, and efficient data storage.

TASK

1) Codes are given above for 4 Encoding functions. Deploy, observe and explain each.
2) Compare and verify the similarities and differences of four encoding functions

TASK

1) Define a struct **Laptop** that includes multiple data types as its members.
2) Write a Solidity function to encode an array of your defined struct using ABI encoding.

6 ABI Decode

6.1 Introduction:

In Solidity, the **abi.decode()** function plays a vital role in the deserialization process by decoding binary data generated by abi.encode().

6.2 Decoding Binary Data using abi.decode():

The abi.decode() function in Solidity enables the conversion of binary data back into its original parameter types. It is used to restore the parameters that were encoded using abi.encode(). Here's how the abi.decode() function works:

1) **Syntax:** The **abi.decode()** function takes **two** arguments: the binary data to be decoded and the data type specification representing the expected structure of the decoded data.

```
(decodedParams) = abi.decode(binaryData, (dataType1, dataType2, ...));
```

2) **Data Type Specification:** The data type specification defines the expected structure of the decoded data. It consists of the individual data types in the order they were encoded. For example, (uint, address, string, uint[2]) specifies that the decoded data will contain a uint, an address, a string, and an array of two uints.
3) **Decoding Process:** During the decoding process, abi.decode() takes the binary data and applies the specified data type specification to extract the original parameters. The decoded parameters are assigned to variables in the same order as the data type specification.

Conclusion:

The abi.decode() function in Solidity serves as a crucial tool for decoding binary data generated by abi.encode(), allowing the restoration of original parameters.

By understanding how to use abi.decode() and providing the correct data type specification, you can effectively decode binary data and extract the encoded parameters in their original format.

TASK

Decode the **bytes** obtained in previous task from **abi.encode()** and verify the output.

```
 function decode(bytes memory data) public pure returns(address daddr,
uint dnum) {
        (daddr, dnum) = abi.decode(data, (address,  uint));
```

TASK

1) Use ABI encoding techniques to serialize the voting data that takes address, ID, name and date of birth (unit[3]) into a byte array
2) Use ABI decoding techniques to extract the voter ID details from the encoded byte array.
3) Verify the accuracy of the decoded data by comparing it with the original voting data.