Solidity Libraries Quiz

Quiz 1: Understanding Libraries

Instructions: Neri is creating reusable code to help fight Hackana. Which statement correctly describes libraries in Solidity?

```
pragma solidity ^0.8.0;

library MathLibrary {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        return a + b;
    }

    function subtract(uint256 a, uint256 b) internal pure returns (uint256) {
        require(a >= b, "Subtraction overflow");
        return a - b;
    }
}

contract Calculator {
    using MathLibrary for uint256;

    function calculate(uint256 x, uint256 y) public pure returns (uint256,
uint256) {
        return (x.add(y), x.subtract(y));
    }
}
```

Which statement about libraries in Solidity is correct?

- A) Libraries can have state variables like contracts
- B) Libraries are deployed independently and can be reused by multiple contracts
- C) Libraries can inherit from other contracts
- D) Libraries can receive Ether like regular contracts

Answer: B) Libraries are deployed independently and can be reused by multiple contracts

Explanation: Libraries in Solidity are designed to be reusable code that can be deployed once and used by multiple contracts. This makes them efficient for common operations or utilities.

The other options are incorrect because:

- Libraries cannot have state variables (Option A)
- Libraries cannot inherit from other contracts (Option C)
- Libraries cannot receive Ether as they don't have a fallback or receive function (Option D)

Libraries help reduce code duplication and gas costs when multiple contracts need the same functionality. In the example, the `MathLibrary` provides basic math operations that can be used by any contract.

Quiz 2: Using Libraries

Instructions: Neri is implementing a secure random number generator. Which code correctly demonstrates using a library in Solidity?

```
pragma solidity ^0.8.0;

// Option A
library RandomLibrary {
    function getRandomNumber(uint256 seed) public view returns (uint256) {
        return uint256(keccak256(abi.encodePacked(block.timestamp, seed)));
    }
}

contract RandomGenerator1 {
    function generateRandom(uint256 seed) public view returns (uint256) {
        return RandomLibrary.getRandomNumber(seed);
    }
}

// Option B
library StringLibrary {
    function concat(string memory a, string memory b) internal pure returns
(string memory) {
        return string(abi.encodePacked(a, b));
    }
}

contract StringProcessor {
    using StringLibrary for string;

    function combineStrings(string memory a, string memory b) public pure returns
(string memory) {
        return a.concat(b);
    }
}

// Option C
library ArrayLibrary {
    uint256[] public values;

    function sum(uint256[] memory array) internal pure returns (uint256) {
        uint256 total = 0;
        for (uint i = 0; i < array.length; i++) {
            total += array[i];
        }
        return total;
    }
}
```

```
contract ArrayProcessor {
    function sumArray(uint256[] memory arr) public view returns (uint256) {
        return ArrayLibrary.sum(arr);
    }
}

// Option D
library Calculator {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        return a + b;
    }
}

contract MathContract {
    Calculator calculator = new Calculator();

    function performAdd(uint256 x, uint256 y) public view returns (uint256) {
        return calculator.add(x, y);
    }
}
```

Which option correctly demonstrates using a library in Solidity?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: B) Option B

Explanation: Option B correctly demonstrates using a library in Solidity with these proper patterns:

1. The library functions are marked as `internal`
2. The contract uses the `using LibraryName for Type` syntax to attach library functions to a type
3. The library is called with the syntax `variable.functionName()` rather than `LibraryName.functionName()`

The other options have issues:

- Option A uses a `public` function in a library, which is valid but inefficient (libraries with internal functions are preferred for gas efficiency)
- Option C tries to define state variables in a library (`uint256[] public values`), which isn't allowed
- Option D tries to create a new instance of the library using `new Calculator()`, which isn't allowed as libraries can't be instantiated

Using `using LibraryName for Type` as shown in Option B is the recommended way to use libraries as it allows for a cleaner, more object-oriented syntax.

Quiz 3: Library Deployment

Instructions: Neri is deploying a library for her anti-Hackana system. Which statement correctly describes library deployment in Solidity?

```
pragma solidity ^0.8.0;

library VerificationLibrary {
    function verify(bytes32 hash, bytes memory signature) internal pure returns
(address) {
        // Complex verification logic
        return address(0);
    }
}

contract DocumentVerifier {
    using VerificationLibrary for bytes32;

    function verifyDocument(bytes32 documentHash, bytes memory signature) public
pure returns (address) {
        return documentHash.verify(signature);
    }
}
```

Which statement about deploying this library is correct?

- A) The library code is automatically included in each contract that uses it
- B) The library is deployed separately and the contract bytecode only contains a reference to it
- C) Libraries can't be deployed, they must be included directly in the contract code
- D) Each function call to the library deploys a new instance of the library

Answer: B) The library is deployed separately and the contract bytecode only contains a reference to it

Explanation: For libraries with `internal` functions in Solidity, the correct deployment model is:

For internal functions (like in our example):

- The library code is embedded into the contract during compilation
- No separate deployment is needed for internal libraries
- This is gas-efficient for smaller libraries

For external functions (not in our example):

- The library must be deployed separately
- Contracts using the library only contain a reference to the deployed library
- This is gas-efficient for larger libraries used by multiple contracts

This deployment model is a key feature of Solidity libraries, allowing code reuse without duplicating code across all contracts that use it.

Quiz 4: Library Restrictions

Instructions: Neri is designing a library for secure data handling. Which code demonstrates a valid library implementation in Solidity?

```
pragma solidity ^0.8.0;

// Option A
library SecureDataA {
    uint256 private totalProcessed;

    function processData(uint256 data) internal pure returns (uint256) {
        totalProcessed++;
        return data * 2;
    }
}

// Option B
library SecureDataB {
    function processData(uint256 data) internal pure returns (uint256) {
        return data * 2;
    }

    fallback() external payable {
        // Handle unexpected calls
    }
}

// Option C
library SecureDataC {
    function processData(uint256 data) internal pure returns (uint256) {
        return data * 2;
    }

    function destroy() internal {
        selfdestruct(payable(address(0)));
    }
}

// Option D
library SecureDataD {
    function processData(uint256 data) internal pure returns (uint256) {
        return data * 2;
    }

    function processAndLog(uint256 data) internal returns (uint256) {
        // Logic to process and log
        return data * 2;
    }
}
```

Which option demonstrates a valid library implementation?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: D) Option D

Explanation: Option D demonstrates a valid library implementation that follows all the rules for Solidity libraries:

1. It doesn't contain any state variables
2. It has only functions (no fallback or receive functions)
3. It doesn't use `selfdestruct`
4. It has both `pure` and non-pure functions, which is allowed

The other options violate library rules:

- Option A tries to define a state variable (`uint256 private totalProcessed`), which is not allowed in libraries
- Option B tries to define a `fallback()` function, which is not allowed in libraries
- Option C tries to use `selfdestruct()`, which is not allowed in libraries

Libraries in Solidity are stateless and can't be destroyed, making them reliable utility code that can be safely reused by multiple contracts.