Solidity Calldata Quiz

Quiz 1: Understanding Calldata Basics

Instructions: Neri is optimizing her contracts against Hackana's attacks. Which statement correctly describes calldata in Solidity?

```
pragma solidity ^0.8.0;

contract CalldataExplained {
    // Statement A: Calldata is mutable like memory but persists like storage
    // Statement B: Calldata is immutable, read-only, and used for function
parameters
    // Statement C: Calldata is only usable with internal functions
    // Statement D: Calldata can only store simple value types like uint and bool

    function processData(string calldata input) external pure returns (string
memory) {
        return input;
    }
}
```

Which statement correctly describes calldata?

- A) Calldata is mutable like memory but persists like storage
- B) Calldata is immutable, read-only, and used for function parameters
- C) Calldata is only usable with internal functions
- D) Calldata can only store simple value types like uint and bool

Answer: B) Calldata is immutable, read-only, and used for function parameters

Explanation: Option B correctly describes calldata in Solidity. Calldata is a special data location that:

- Is read-only (immutable), meaning you cannot modify data stored there
- Is specifically used for external function parameters
- Is more gas-efficient than memory for external function parameters
- Can handle complex types like strings and arrays, not just simple types

Using calldata is especially important for optimizing gas costs when dealing with large arrays or strings that don't need to be modified within the function.

Quiz 2: Comparing Calldata vs Memory

Instructions: Neri is reviewing two functions that process transaction data. Which function is more gas-efficient and why?

```
pragma solidity ^0.8.0;

contract TransactionProcessor {
    // Option A
    function processTransactionA(string memory txData) public pure returns (string
memory) {
        return txData;
    }

    // Option B
    function processTransactionB(string calldata txData) external pure returns
(string memory) {
        return txData;
    }
}
```

Which function is more gas-efficient and why?

- A) Function A is more efficient because memory is faster to access than calldata
- B) Function B is more efficient because calldata doesn't require copying the data
- C) Both functions have identical gas costs
- D) Function A is more efficient because public functions have optimized gas costs

Answer: B) Function B is more efficient because calldata doesn't require copying the data

Explanation: Function B using `string calldata` is more gas-efficient because:

- When using `memory` in Function A, the data is copied from calldata to memory, requiring additional gas
- With `calldata` in Function B, the function directly references the original input data without copying it
- This is especially significant for large strings or arrays, where copying can be expensive
- Additionally, Function B is marked as `external` rather than `public`, which is more efficient for functions called from outside the contract

This gas optimization is one of the key benefits of using calldata for external function parameters, especially when dealing with large data structures that don't need modification.

Quiz 3: When to Use Calldata

Instructions: Neri is deciding on data locations for her barangay's data processing contract. For which scenario is calldata the MOST appropriate choice?

```
pragma solidity ^0.8.0;

contract DataProcessor {
    struct CitizenData {
        string name;
        uint256 age;
    }

    // Scenario A
    function updateCitizenAge(string calldata name, uint256 newAge) external {
        // Update age in storage
    }

    // Scenario B
    function addCitizenRecord(CitizenData calldata citizen) external {
        // Store new citizen
    }

    // Scenario C
    function processCitizenData(uint256[] calldata ids) external pure returns
(uint256) {
        uint256 count = 0;
        for (uint i = 0; i < ids.length; i++) {
            count += ids[i];
        }
        return count;
    }

    // Scenario D
    function modifyCitizenData(string[] calldata names) external pure returns
(string[] memory) {
        string[] memory modified = new string[](names.length);
        for (uint i = 0; i < names.length; i++) {
            modified[i] = string(abi.encodePacked("Verified: ", names[i]));
        }
        return modified;
    }
}
```

For which scenario is calldata the MOST beneficial choice in terms of gas optimization?

- A) Scenario A
- B) Scenario B
- C) Scenario C
- D) Scenario D

Answer: C) Scenario C

Explanation: Scenario C benefits most from using calldata because:

- It works with a potentially large array (`uint256[] calldata ids`)
- It only reads from the array without modifying it
- It performs a simple summation operation that doesn't require modifying the input

While all scenarios could use calldata, Scenario C represents the ideal case for calldata optimization:

- Scenario A uses simple value types where the gas savings would be minimal
- Scenario B uses a struct which is beneficial with calldata, but not as much as a large array
- Scenario D creates a new modified array in memory, so while the input benefits from calldata, the function still requires memory allocation for the output

Calldata provides the greatest optimization when working with large arrays or strings that are only read from and not modified.

Quiz 4: Calldata Limitations

Instructions: Neri is fixing a data processing function. Which code snippet containing calldata will compile successfully?

```
pragma solidity ^0.8.0;

contract DataValidator {
    // Option A
    function validateData1(string calldata data) external pure returns (bool) {
        data = "Modified"; // Try to modify calldata
        return data.length > 0;
    }

    // Option B
    function validateData2(string calldata data) external pure returns (bool) {
        return data.length > 0;
    }

    // Option C
    function validateData3(string calldata data) internal pure returns (bool) {
        return data.length > 0;
    }

    // Option D
    function validateData4(string calldata data) private pure returns (bool) {
        return data.length > 0;
    }
}
```

Which function will compile successfully?

- A) validateData1()
- B) validateData2()
- C) validateData3()
- D) validateData4()

Answer: B) validateData2()

Explanation: Function `validateData2()` is the only one that will compile successfully because:

1. It uses `calldata` correctly as a read-only parameter
2. It doesn't attempt to modify the calldata parameter
3. It uses the `external` visibility modifier which is compatible with calldata

The other functions have issues:

- Function A attempts to modify the calldata parameter (`data = "Modified"`), which is not allowed since calldata is immutable
- Function C uses the `internal` visibility with calldata, which is invalid (calldata can only be used with external functions)
- Function D uses the `private` visibility with calldata, which is also invalid (same reason as C)

Remember that calldata:

1. Can only be used with `external` functions
2. Cannot be modified (it's read-only)
3. Is more gas-efficient for external functions that don't need to modify their parameters