Solidity Interfaces Quiz

Quiz 1: Understanding Interface Basics

Instructions: Neri is designing a system that needs to interact with different token contracts. Which code correctly defines a Solidity interface?

```
pragma solidity ^0.8.0;

// Option A
interface IToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// Option B
interface IToken {
    uint256 public totalSupply;
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// Option C
interface IToken {
    function transfer(address to, uint256 amount) external returns (bool) {
        // Transfer implementation
        return true;
    }
    function balanceOf(address account) external view returns (uint256);
}

// Option D
interface IToken {
    function transfer(address to, uint256 amount);
    function balanceOf(address account) view returns (uint256);
}
```

Which option correctly defines a Solidity interface?

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

Answer: A) Option A

Explanation: Option A correctly defines a Solidity interface with these key characteristics:

1. It only declares function signatures without implementing them

2. All functions are marked as `external` (interfaces can only declare external functions)
3. It specifies return types for all functions

The other options have issues:

- Option B tries to declare a state variable (`uint256 public totalSupply`), which is not allowed in interfaces
- Option C tries to implement a function body, which is not allowed in interfaces
- Option D doesn't use the `external` visibility modifier, which is required for interface functions

Interfaces in Solidity are like contracts that only define what functions must be implemented, but not how they work. They're essential for standardizing interactions between different contracts.

Quiz 2: Using Interfaces

Instructions: Neri is building a contract that interacts with different token contracts. Which code correctly uses an interface to interact with an existing token contract?

```
pragma solidity ^0.8.0;

interface IERC20 {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// Option A
contract TokenInteractor1 {
    function sendTokens(address tokenAddress, address to, uint256 amount) external
{
        IERC20 token = new IERC20(tokenAddress);
        token.transfer(to, amount);
    }
}

// Option B
contract TokenInteractor2 {
    function sendTokens(address tokenAddress, address to, uint256 amount) external
{
        IERC20 token = IERC20(tokenAddress);
        token.transfer(to, amount);
    }
}

// Option C
contract TokenInteractor3 {
    IERC20 public token;

    constructor(address tokenAddress) {
        token = IERC20(tokenAddress);
    }

    function sendTokens(address to, uint256 amount) external {
        token.transfer(to, amount);
    }
}

// Option D
contract TokenInteractor4 {
    function sendTokens(address to, uint256 amount) external {
        IERC20.transfer(to, amount);
    }
}
```

Which option(s) correctly use(s) an interface to interact with an existing token contract?

- A) Option A
- B) Option B and Option C
- C) Option C only
- D) Option D

Answer: B) Option B and Option C

Explanation: Options B and C both correctly use the IERC20 interface to interact with an existing token contract:

- Option B creates a temporary interface instance pointing to the token address, then calls a function on it
- Option C stores the interface instance as a state variable during construction, then uses it later

The other options have issues:

- Option A tries to create a new instance of the interface with `new IERC20(tokenAddress)`, but interfaces cannot be instantiated with `new`
- Option D tries to call the function directly on the interface type, not on an instance, which is incorrect

When using interfaces in Solidity:

1. You can't create new instances with `new` since interfaces have no implementation
2. You create interface instances by casting an address to the interface type: `InterfaceName(address)`
3. You then call functions on that instance, not on the interface type itself

Quiz 3: Interface Inheritance

Instructions: Neri is extending token functionality with additional features. Which code correctly demonstrates interface inheritance?

```
pragma solidity ^0.8.0;

// Option A
interface IBasicToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

interface IExtendedToken extends IBasicToken {
    function burnTokens(uint256 amount) external returns (bool);
}

// Option B
interface IBasicToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

interface IExtendedToken is IBasicToken {
    function burnTokens(uint256 amount) external returns (bool);
}

// Option C
interface IBasicToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

interface IExtendedToken {
    IBasicToken basicFunctions;
    function burnTokens(uint256 amount) external returns (bool);
}

// Option D
interface IBasicToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

interface IExtendedToken: IBasicToken {
    function burnTokens(uint256 amount) external returns (bool);
}
```

Which option correctly implements interface inheritance?

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

Answer: A) Option A

Explanation: Option A correctly demonstrates interface inheritance using the `extends` keyword, which is the proper syntax for interface inheritance in Solidity.

With this approach, `IExtendedToken` contains all the function declarations from `IBasicToken` plus its own additional functions.

The other options have issues:

- Option B uses the `is` keyword which is for contract inheritance, not interface inheritance
- Option C tries to use composition (storing another interface as a variable), which is not allowed in interfaces
- Option D uses a colon (`:`) for inheritance, which is incorrect syntax in Solidity

Interface inheritance is a powerful way to build upon existing standards while maintaining compatibility with them. A contract implementing `IExtendedToken` would need to implement all functions from both interfaces.

Quiz 4: Interface Implementation

Instructions: Neri is implementing a token standard in her contract. Which code correctly implements an interface in a contract?

```
pragma solidity ^0.8.0;

interface IToken {
    function transfer(address to, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

// Option A
contract MyToken1 is IToken {
    mapping(address => uint256) private _balances;

    function transfer(address to, uint256 amount) public returns (bool) {
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        return true;
    }

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }
}

// Option B
contract MyToken2 is IToken {
    mapping(address => uint256) private _balances;

    function transfer(address to, uint256 amount) external returns (bool) {
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        return true;
    }

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }
}
```

```
// Option C
contract MyToken3 implements IToken {
    mapping(address => uint256) private _balances;

    function transfer(address to, uint256 amount) external returns (bool) {
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        return true;
    }

    function balanceOf(address account) external view returns (uint256) {
        return _balances[account];
    }
}

// Option D
contract MyToken4 is IToken {
    mapping(address => uint256) private _balances;

    function transfer(address to, uint256 amount) external override returns (bool)
{
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        return true;
    }

    function balanceOf(address account) external view override returns (uint256) {
        return _balances[account];
    }
}
```

Which option correctly implements the IToken interface?

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

Answer: D) Option D

Explanation: Option D correctly implements the IToken interface with the following key elements:

1. It uses the `is` keyword to inherit from the interface
2. It maintains the `external` visibility for the functions as defined in the interface
3. It uses the `override` keyword to explicitly indicate that it's implementing functions defined in the interface

The other options have issues:

- Option A changes the function visibility from `external` to `public`, which doesn't match the interface
- Option B inherits from the interface but doesn't use the `override` keyword (required in Solidity 0.8+)
- Option C uses the keyword `implements` which is not valid Solidity syntax (should use `is` instead)

In Solidity 0.8+, the `override` keyword is required when implementing interface functions to explicitly acknowledge that you're providing an implementation for a function declared elsewhere.