Solidity Inheritance Quiz

Quiz 1: Understanding Basic Inheritance

Instructions: Neri is developing a contract system to counter Hackana. Which code example correctly demonstrates inheritance in Solidity?

```
pragma solidity ^0.8.0;

// Option A
contract ParentA {
    uint256 public value = 100;
}

contract ChildA extends ParentA {
    function getValue() public view returns (uint256) {
        return value;
    }
}

// Option B
contract ParentB {
    uint256 public value = 100;
}

contract ChildB : ParentB {
    function getValue() public view returns (uint256) {
        return value;
    }
}

// Option C
contract ParentC {
    uint256 public value = 100;
}

contract ChildC inherits ParentC {
    function getValue() public view returns (uint256) {
        return value;
    }
}

// Option D
contract ParentD {
    uint256 public value = 100;
}

contract ChildD {
    ParentD parent;
```

```
    constructor() {
        parent = new ParentD();
    }

    function getValue() public view returns (uint256) {
        return parent.value();
    }
}
```

Which option correctly implements inheritance in Solidity?

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

Answer: A) Option A

Explanation: Option A correctly implements inheritance in Solidity using the `is` keyword. In Solidity, a contract inherits from another by using the syntax: `contract Child is Parent { ... }`.

The other options are incorrect:

- Option B uses a colon (`:`) for inheritance, which is syntax from other languages like C++ but not valid in Solidity
- Option C uses the keyword `inherits`, which is not valid Solidity syntax
- Option D doesn't use inheritance at all - it's using composition (creating an instance of another contract)

With inheritance in Option A, the `ChildA` contract automatically has access to all public and internal members of `ParentA`, including the `value` variable.

Quiz 2: Function Overriding

Instructions: Neri needs to customize behavior in a derived contract. Which code snippet correctly demonstrates function overriding in Solidity?

```
pragma solidity ^0.8.0;

// Option A
contract BaseContractA {
    function getMessage() public pure returns (string memory) {
        return "Base message";
    }
}

contract DerivedContractA is BaseContractA {
    function getMessage() public pure returns (string memory) {
        return "Derived message";
    }
```

```
}

// Option B
contract BaseContractB {
    function getMessage() public pure virtual returns (string memory) {
        return "Base message";
    }
}

contract DerivedContractB is BaseContractB {
    function getMessage() public pure override returns (string memory) {
        return "Derived message";
    }
}

// Option C
contract BaseContractC {
    function getMessage() public pure virtual returns (string memory) {
        return "Base message";
    }
}

contract DerivedContractC is BaseContractC {
    function getMessage() public pure returns (string memory) {
        return super.getMessage() + " extended";
    }
}

// Option D
contract BaseContractD {
    function getMessage() public pure returns (string memory) {
        return "Base message";
    }
}

contract DerivedContractD is BaseContractD {
    function getMessage() public pure override returns (string memory) {
        return "Derived message";
    }
}
```

Which option correctly implements function overriding in Solidity (version 0.8 and above)?

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

Answer: B) Option B

Explanation: Option B correctly implements function overriding in Solidity (version 0.8+) by:

1. Using the `virtual` keyword in the base contract to mark the function as overridable
2. Using the `override` keyword in the derived contract to explicitly indicate that the function overrides the parent function

The other options have issues:

- Option A attempts to override without using the `virtual` and `override` keywords, which causes a compilation error in Solidity 0.8+
- Option C attempts to extend the parent function but doesn't use the `override` keyword
- Option D uses the `override` keyword but the base function isn't marked as `virtual`, so it will fail to compile

Since Solidity 0.6.0, the `virtual` and `override` keywords are required to make function overriding explicit and prevent accidental overriding.

Quiz 3: Multiple Inheritance

Instructions: Neri is designing an advanced security system with multiple contract types. Which scenario correctly demonstrates multiple inheritance resolution in Solidity?

```
pragma solidity ^0.8.0;

// Option A
contract SecurityBase {
    function getSecurityLevel() public pure virtual returns (string memory) {
        return "Base";
    }
}

contract DataHandling {
    function getSecurityLevel() public pure virtual returns (string memory) {
        return "Data";
    }
}

contract SecureDataContract is SecurityBase, DataHandling {
    function getSecurityLevel() public pure override(SecurityBase, DataHandling)
returns (string memory) {
        return "Secure Data";
    }
}

// Option B
contract BaseContract {
    function getVersion() public pure returns (string memory) {
        return "v1.0";
    }
}

contract MidContract is BaseContract {
    function getDetails() public pure returns (string memory) {
```

```
        return "Details";
    }
}

contract FinalContract is MidContract {
    function getStatus() public pure returns (string memory) {
        return "Active";
    }
}

// Option C
contract ContractOne {
    function getData() public pure virtual returns (string memory) {
        return "One";
    }
}

contract ContractTwo {
    function getData() public pure virtual returns (string memory) {
        return "Two";
    }
}

contract DerivedContract is ContractTwo, ContractOne {
    function getData() public pure override(ContractOne, ContractTwo) returns
(string memory) {
        return super.getData();
    }
}

// Option D
contract Authorization {
    function checkAccess() public pure virtual returns (bool) {
        return true;
    }
}

contract Validation {
    function validate() public pure virtual returns (bool) {
        return true;
    }
}

contract SecureProcess is Authorization, Validation {
    function processSecurely() public pure returns (string memory) {
        return "Processed";
    }
}
```

In Option C, what will be returned when calling getData() on the DerivedContract?

- A) "One"

- B) "Two"
- C) A compilation error will occur
- D) "OneTwo"

Answer: B) "Two"

Explanation: In Solidity, when using multiple inheritance with `super`, the function call is resolved based on the C3 linearization algorithm, which determines the order in which base contracts are searched for function definitions.

In Option C, when `super.getData()` is called in `DerivedContract`, it follows the inheritance order from right to left:

1. `DerivedContract` inherits from `ContractTwo, ContractOne` (in that order)
2. When resolving `super.getData()`, it will first check `ContractTwo`, find the function, and return "Two"
3. The function in `ContractOne` is never reached

This is a key aspect of Solidity's inheritance system: the order of inheritance matters, with the rightmost contract being the most "base" contract in the hierarchy.

Quiz 4: Inheritance and Constructors

Instructions: Neri is building a contract system with initialization parameters. Which example correctly passes constructor parameters up the inheritance chain?

```
pragma solidity ^0.8.0;

// Option A
contract TokenBase {
    string public name;

    constructor(string memory _name) {
        name = _name;
    }
}

contract CustomToken is TokenBase {
    uint256 public decimals;

    constructor(uint256 _decimals) {
        decimals = _decimals;
    }
}

// Option B
contract TokenBase2 {
    string public name;

    constructor(string memory _name) {
        name = _name;
    }
```

```
}

contract CustomToken2 is TokenBase2 {
    uint256 public decimals;

    constructor(string memory _name, uint256 _decimals) TokenBase2(_name) {
        decimals = _decimals;
    }
}

// Option C
contract TokenBase3 {
    string public name;

    constructor(string memory _name) {
        name = _name;
    }
}

contract CustomToken3 is TokenBase3("DefaultName") {
    uint256 public decimals;

    constructor(uint256 _decimals) {
        decimals = _decimals;
    }
}

// Option D
contract TokenBase4 {
    string public name;

    constructor(string memory _name) {
        name = _name;
    }
}

contract CustomToken4 is TokenBase4 {
    uint256 public decimals;

    constructor(string memory _name, uint256 _decimals) {
        super(_name); // Try to call parent constructor
        decimals = _decimals;
    }
}
```

Which option correctly initializes inherited constructors?

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

Answer: B) Option B

Explanation: Option B correctly passes constructor parameters to the base contract by specifying the base constructor call in the derived contract's constructor signature.

In Solidity, when a contract inherits from another contract that has a constructor with parameters, you must explicitly pass those parameters to the base constructor using the syntax shown in Option B: `constructor(...) BaseContract(baseParams) { ... }`.

The other options have issues:

- Option A doesn't pass any parameters to the base constructor, so it will fail to compile
- Option C provides a fixed value ("DefaultName") directly in the inheritance declaration, which is a valid but limited approach as it can't use constructor parameters
- Option D tries to call the parent constructor using `super(_name)` inside the constructor body, which is invalid syntax in Solidity

Option B is the standard pattern for passing parameters up an inheritance chain in Solidity.