Solidity ERC20 Tokens Quiz

Quiz 1: Understanding ERC20 Basics

Instructions: Neri is creating a community token to unite efforts against Hackana. Which statement correctly describes the ERC20 token standard?

```
pragma solidity ^0.8.0;

// Contract to implement token functionality
contract CommunityToken {
    string public name = "San Juan Community Token";
    string public symbol = "SJCT";
    uint8 public decimals = 18;
    uint256 public totalSupply = 1000000 * 10**18;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor() {
        balanceOf[msg.sender] = totalSupply;
    }

    function transfer(address to, uint256 value) public returns (bool) {
        require(balanceOf[msg.sender] >= value, "Insufficient balance");
        balanceOf[msg.sender] -= value;
        balanceOf[to] += value;
        emit Transfer(msg.sender, to, value);
        return true;
    }

    function approve(address spender, uint256 value) public returns (bool) {
        allowance[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        return true;
    }

    function transferFrom(address from, address to, uint256 value) public returns
(bool) {
        require(balanceOf[from] >= value, "Insufficient balance");
        require(allowance[from][msg.sender] >= value, "Insufficient allowance");
        balanceOf[from] -= value;
        balanceOf[to] += value;
        allowance[from][msg.sender] -= value;
        emit Transfer(from, to, value);
        return true;
```

```
    }
}
```

Which statement about the ERC20 token standard is correct?

- A) ERC20 is primarily designed for non-fungible tokens where each token is unique
- B) ERC20 tokens require a minimum total supply of 1 million tokens to be valid
- C) ERC20 defines a standard interface for fungible tokens with functions like transfer, approve, and transferFrom
- D) ERC20 tokens must be deployed with exactly 18 decimals as shown in the example

Answer: C) ERC20 defines a standard interface for fungible tokens with functions like transfer, approve, and transferFrom

Explanation: The ERC20 standard defines a common interface for fungible tokens on Ethereum, where each token is identical to every other token. The key functions and features of ERC20 tokens include:

1. Core functions like `transfer`, `approve`, and `transferFrom` for token movement
2. View functions like `balanceOf` and `allowance` for checking token information
3. Events like `Transfer` and `Approval` for logging token movements
4. Standard properties like `name`, `symbol`, `decimals`, and `totalSupply`

This standardization allows wallets, exchanges, and other applications to interact with any ERC20 token without needing custom code for each token implementation.

Quiz 2: ERC20 Function Understanding

Instructions: Neri is implementing token approval functionality. What is the purpose of the `approve` and `transferFrom` functions in the ERC20 standard?

```
pragma solidity ^0.8.0;

contract TokenExample {
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    // Example function implementations
    function approve(address spender, uint256 amount) public returns (bool) {
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function transferFrom(address sender, address recipient, uint256 amount)
public returns (bool) {
        require(balanceOf[sender] >= amount, "Insufficient balance");
        require(allowance[sender][msg.sender] >= amount, "Insufficient
```

```
allowance");

        balanceOf[sender] -= amount;
        balanceOf[recipient] += amount;
        allowance[sender][msg.sender] -= amount;

        emit Transfer(sender, recipient, amount);
        return true;
    }
}
```

What is the main purpose of the approve and transferFrom functions in ERC20 tokens?

- A) To allow the token owner to burn tokens when they're no longer needed
- B) To enable token holders to delegate token transfers to third-party contracts or addresses
- C) To restrict token transfers to a whitelist of approved addresses only
- D) To split tokens into smaller denominations for microtransactions

Answer: B) To enable token holders to delegate token transfers to third-party contracts or addresses

Explanation: The `approve` and `transferFrom` functions work together to enable a powerful feature of ERC20 tokens: delegated transfers. This mechanism:

1. Allows token holders to authorize another address (like a smart contract) to transfer a specific amount of tokens on their behalf
2. Makes it possible to build complex DeFi protocols like decentralized exchanges, lending platforms, and staking systems
3. Enables better user experiences where users can approve contracts once and then perform multiple operations without signing each transaction

The workflow typically works like this:

- User calls `approve(spenderAddress, amount)` to give permission to the spender
- The spender (often a contract) can then call `transferFrom(userAddress, recipient, amount)` to move tokens from the user to any recipient, up to the approved amount

This delegation model is fundamental to many Ethereum applications and allows for more complex token interactions than simple peer-to-peer transfers.

Quiz 3: ERC20 Implementation with OpenZeppelin

Instructions: Neri is choosing between different ERC20 implementation approaches for the barangay's community token. Which code correctly implements an ERC20 token using OpenZeppelin?

```
pragma solidity ^0.8.0;

// Option A
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract CommunityTokenA is ERC20 {
```

```
    constructor() ERC20("San Juan Community Token", "SJCT") {
        _mint(msg.sender, 1000000 * 10**decimals());
    }
}

// Option B
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract CommunityTokenB is IERC20 {
    string public name = "San Juan Community Token";
    string public symbol = "SJCT";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    constructor() {
        _mint(msg.sender, 1000000 * 10**decimals());
    }

    function _mint(address account, uint256 amount) internal {
        totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);
    }

    // Implement all IERC20 functions...
}

// Option C
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract CommunityTokenC {
    ERC20 private _token;

    constructor() {
        _token = new ERC20("San Juan Community Token", "SJCT");
        _token._mint(msg.sender, 1000000 * 10**18);
    }
}

// Option D
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract CommunityTokenD is ERC20Burnable, Ownable {
    constructor() ERC20("San Juan Community Token", "SJCT") {
        _mint(msg.sender, 1000000 * 10**decimals());
    }

    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
```

```
    }
}
```

Which implementation correctly uses OpenZeppelin to create an ERC20 token?

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

Answer: D) Option D

Explanation: Both Options A and D correctly implement an ERC20 token using OpenZeppelin's libraries, but Option D provides the most complete and feature-rich implementation by:

1. Inheriting from `ERC20Burnable`, which adds token burning functionality (allowing users to destroy their tokens)
2. Inheriting from `Ownable`, which adds access control for privileged functions
3. Including a `mint` function that can only be called by the owner
4. Using the standard pattern of calling `_mint` in the constructor to create the initial supply

Option A is also correct but more basic, without the additional features.

The other options have issues:

- Option B manually implements the ERC20 interface, defeating the purpose of using OpenZeppelin
- Option C incorrectly tries to create a new ERC20 instance inside the contract and call a protected `_mint` function

OpenZeppelin's ERC20 implementation is widely used because it's secure, gas-efficient, and thoroughly tested. By inheriting from it, you get a complete implementation that follows best practices.

Quiz 4: Advanced ERC20 Features

Instructions: Neri wants to add a feature to the barangay token that automatically takes a small fee from each transfer to fund community projects. Which implementation correctly adds this feature?

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// Option A
contract TokenWithFeeA is ERC20 {
    address public feeCollector;
    uint256 public feePercent = 2; // 2% fee

    constructor(address _feeCollector) ERC20("Fee Token", "FEE") {
        feeCollector = _feeCollector;
        _mint(msg.sender, 1000000 * 10**decimals());
    }
```

```
    function transfer(address recipient, uint256 amount) public override returns
(bool) {
        uint256 fee = (amount * feePercent) / 100;
        uint256 remainingAmount = amount - fee;

        super.transfer(feeCollector, fee);
        super.transfer(recipient, remainingAmount);

        return true;
    }
}

// Option B
contract TokenWithFeeB is ERC20 {
    address public feeCollector;
    uint256 public feePercent = 2; // 2% fee

    constructor(address _feeCollector) ERC20("Fee Token", "FEE") {
        feeCollector = _feeCollector;
        _mint(msg.sender, 1000000 * 10**decimals());
    }

    function _transfer(address sender, address recipient, uint256 amount) internal
virtual override {
        uint256 fee = (amount * feePercent) / 100;
        uint256 remainingAmount = amount - fee;

        super._transfer(sender, feeCollector, fee);
        super._transfer(sender, recipient, remainingAmount);
    }
}

// Option C
contract TokenWithFeeC is ERC20 {
    address public feeCollector;
    uint256 public feePercent = 2; // 2% fee

    constructor(address _feeCollector) ERC20("Fee Token", "FEE") {
        feeCollector = _feeCollector;
        _mint(msg.sender, 1000000 * 10**decimals());
    }

    function transfer(address recipient, uint256 amount) public override returns
(bool) {
        uint256 fee = (amount * feePercent) / 100;

        _transfer(msg.sender, feeCollector, fee);
        _transfer(msg.sender, recipient, amount - fee);

        return true;
    }
}
```

```
// Option D
contract TokenWithFeeD is ERC20 {
    address public feeCollector;
    uint256 public feePercent = 2; // 2% fee

    constructor(address _feeCollector) ERC20("Fee Token", "FEE") {
        feeCollector = _feeCollector;
        _mint(msg.sender, 1000000 * 10**decimals());
    }

    function transfer(address recipient, uint256 amount) public override returns
(bool) {
        uint256 fee = (amount * feePercent) / 100;
        uint256 remainingAmount = amount - fee;

        bool success = super.transfer(feeCollector, fee);
        if (success) {
            return super.transfer(recipient, remainingAmount);
        }
        return false;
    }
}
```

Which implementation correctly adds a transfer fee feature to an ERC20 token?

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

Answer: B) Option B

Explanation: Option B correctly implements a transfer fee by overriding the internal `_transfer` function, which is the recommended approach for customizing transfer behavior in OpenZeppelin's ERC20 implementation.

The key advantages of Option B:

1. It overrides the internal `_transfer` function, which affects all token transfers (both direct transfers and transferFrom)
2. It uses `super._transfer` to maintain the core functionality of the parent class
3. It correctly calculates and deducts the fee from the transferred amount

The other options have issues:

- Option A overrides the `transfer` function but not `transferFrom`, allowing the fee to be bypassed
- Option C directly calls `_transfer` which is internal, creating potential confusion in the function call flow
- Option D makes two separate transfers with a check in between, which is less gas-efficient and could lead to partial transfers

When customizing ERC20 behavior, overriding the internal functions (like `_transfer`) is generally the best approach as it ensures the customization applies consistently across all related functions.