Solidity Function Modifiers Quiz

Quiz 1: Understanding Modifiers

Instructions: Neri is building a secure donation fund with function modifiers. What is the main purpose of using modifiers in her smart contract?

```
pragma solidity ^0.8.0;

contract DonationFund {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    function withdrawFunds(uint256 amount) public onlyOwner {
        // Code to withdraw funds
    }
}
```

What is the main purpose of the `onlyOwner` modifier in this contract?

- A) It makes the function run faster
- B) It restricts who can call certain functions
- C) It automatically sends funds to the owner
- D) It prevents the contract from being hacked entirely

Answer: B) It restricts who can call certain functions

Explanation: Function modifiers act like security guards for your functions. The `onlyOwner` modifier checks if the person calling the function is the owner of the contract before allowing the function to run. If someone else tries to call the function, they'll get the error message "Not the owner" and the function won't execute. This is great for protecting sensitive functions like withdrawing funds from unauthorized access.

Quiz 2: The Underscore in Modifiers

Instructions: In the modifier below, what does the underscore symbol (_) represent?

```
pragma solidity ^0.8.0;

contract AccessControl {
```

```
    address public admin;

    constructor() {
        admin = msg.sender;
    }

    modifier onlyAdmin() {
        require(msg.sender == admin, "Access denied");
        _;
        // Some code could go here too
    }

    function changeSettings() public onlyAdmin {
        // Code to change settings
    }
}
```

What does the underscore (_) represent in the modifier?

- A) It's just a decoration with no meaning
- B) It represents where the function's code will be executed
- C) It's a special character that ends the modifier
- D) It represents the admin's address

Answer: B) It represents where the function's code will be executed

Explanation: Think of the underscore (_) as saying "put the rest of the function's code here." When you use a modifier on a function, Solidity replaces the underscore with all the code from that function. In this example, first the `require` statement runs to check if you're the admin. If you are, then the function's code runs where the underscore is. If you put code after the underscore, that would run after the function's code executes.

Quiz 3: Multiple Modifiers

Instructions: Neri has added multiple modifiers to a function in her donation contract. In what order will they be executed?

```
pragma solidity ^0.8.0;

contract EnhancedDonation {
    address public owner;
    bool public paused = false;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }
```

```
    modifier notPaused() {
        require(!paused, "Contract is paused");
        _;
    }

    function emergencyWithdraw() public onlyOwner notPaused {
        // Withdrawal code here
    }
}
```

In what order will the modifiers be executed?

- A) The function runs first, then `onlyOwner`, then `notPaused`
- B) `onlyOwner` runs first, then `notPaused`, then the function code
- C) `notPaused` runs first, then `onlyOwner`, then the function code
- D) Both modifiers run at the same time, then the function code

Answer: B) `onlyOwner` runs first, then `notPaused`, then the function code

Explanation: When you use multiple modifiers, they run in the order they're listed (from left to right). In this case, `onlyOwner` is listed first, so it runs first to check if you're the owner. If that passes, then `notPaused` runs to make sure the contract isn't paused. Only after both checks pass will the actual function code run. This is like going through multiple security checkpoints before reaching a secure area.

Quiz 4: Modifiers with Parameters

Instructions: Neri wants to create a modifier that checks if a donation amount is within an acceptable range. Which option correctly implements a modifier with parameters?

```
pragma solidity ^0.8.0;

contract RangeLimitedDonations {
    // Option A
    modifier validAmount() {
        require(msg.value > 0, "Amount must be positive");
        _;
    }

    // Option B
    modifier validAmount(uint256 amount) {
        require(amount > 0, "Amount must be positive");
        _;
    }

    // Option C
    modifier validAmount(uint256 min, uint256 max) {
        _;
        require(msg.value >= min && msg.value <= max, "Amount out of range");
    }

    // Option D
```

```
    modifier validAmount(uint256 min, uint256 max) {
        require(msg.value >= min && msg.value <= max, "Amount out of range");
        _;
    }
}
```

Which option correctly implements a modifier that checks if the donation amount is within a specified range?

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

Answer: D) Option D

Explanation: Option D is correct because it:

1. Accepts two parameters (`min` and `max`) to define the acceptable range
2. Checks if the value sent (`msg.value`) is within that range
3. Places the underscore (`_`) after the check, so the function only runs if the check passes
4. Provides a clear error message if the check fails

Option A doesn't take parameters. Option B takes a parameter but doesn't use `msg.value` which would be needed for donations. Option C puts the underscore before the check, which means the function would run before checking the condition (which defeats the purpose of the modifier).