Solidity Assert and Revert Quiz

Quiz 1: Understanding assert() vs require()

Instructions: Neri is securing her smart contract against Hackana. Look at the code below and determine which statement is correct about `assert()` vs `require()`.

```
pragma solidity ^0.8.0;

contract SecuritySystem {
    uint256 public totalFunds = 100;

    function withdrawSafely(uint256 amount) public {
        require(amount <= totalFunds, "Not enough funds");
        totalFunds -= amount;

        assert(totalFunds >= 0);
    }
}
```

Which statement about the code above is correct?

- A) `require()` and `assert()` do exactly the same thing
- B) `require()` is used for user inputs, while `assert()` is for internal errors
- C) `assert()` is cheaper (uses less gas) than `require()`
- D) The `assert()` statement in this code is unnecessary because `uint256` can't be negative

Answer: D) The `assert()` statement in this code is unnecessary because `uint256` can't be negative

Explanation: In this code, the `assert(totalFunds >= 0)` check is unnecessary because `uint256` (unsigned integer) can never be negative - its range is from 0 to 2^256-1. Even after subtracting, it can't go below zero.

Generally, `require()` is used for checking user inputs or conditions that might legitimately fail, and it returns any remaining gas when it fails. `assert()` is meant for catching internal errors or invariants that should never be false, and it uses up all remaining gas when it fails. In this case, since `totalFunds` can't go below zero due to the data type, the assert is redundant.

Quiz 2: Using revert() Correctly

Instructions: Neri is implementing error handling in her contract. Which code correctly uses the `revert()` function to stop a transaction?

```
pragma solidity ^0.8.0;

contract ReversionExamples {
    address public owner = msg.sender;
```

```
    // Option A
    function example1(uint256 amount) public {
        if (msg.sender != owner) {
            revert("Only owner can call this function");
        }
        // Function continues here if not reverted
    }

    // Option B
    function example2(uint256 amount) public {
        if (msg.sender != owner) {
            revert;
        }
        // Function continues here if not reverted
    }

    // Option C
    function example3(uint256 amount) public {
        revert(msg.sender != owner, "Only owner can call this function");
        // Function continues here if not reverted
    }

    // Option D
    function example4(uint256 amount) public {
        if (msg.sender == owner) {
            revert("Owner cannot call this function");
        }
        // Function continues here if not reverted
    }
}
```

Which option correctly uses `revert()` to prevent non-owners from calling the function?

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

Answer: A) Option A

Explanation: Option A correctly uses the `revert()` function. It checks if the caller is NOT the owner, and if so, it stops the function execution with a clear error message. This is exactly how you should use `revert()` - inside a condition, with a helpful error message explaining why the transaction was stopped.

Option B is using the old syntax without an error message (which isn't recommended). Option C has incorrect syntax - `revert()` doesn't take a condition as its first parameter. Option D has the logic backwards - it would prevent the owner from using the function, which is the opposite of what we want.

Quiz 3: When to Use assert()

Instructions: Neri is building a secure payment system. In which scenario should she use `assert()` instead of `require()`?

```
pragma solidity ^0.8.0;

contract PaymentSystem {
    mapping(address => uint256) public balances;
    uint256 public totalSupply = 1000000;

    function deposit(uint256 amount) public {
        balances[msg.sender] += amount;
        totalSupply += amount;
    }

    function transfer(address to, uint256 amount) public {
        // Check 1: Does sender have enough balance?
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;
        balances[to] += amount;

        // Check 2: Is total supply still the same?
        // Should this use assert or require?
    }
}
```

For Check 2, which statement should Neri use?

- A) `require(totalSupply == 1000000, "Total supply changed");`
- B) `require(balances[to] > 0, "Recipient balance not updated");`
- C) `assert(totalSupply == 1000000 + balances[to]);`
- D) `assert(balances[msg.sender] + balances[to] + otherBalances == totalSupply);`

Answer: D) `assert(balances[msg.sender] + balances[to] + otherBalances == totalSupply);`

Explanation: `assert()` is best used for checking internal invariants - things that should always be true if your code is working correctly. In this case, checking that the total supply equals the sum of all balances is a perfect use for `assert()` because:

1. This should ALWAYS be true regardless of user input
2. If it's not true, there's a serious bug in your contract (funds were created or destroyed)
3. This represents an invariant (unchanging property) of your system

The total supply should match the sum of all balances at all times. If it doesn't, something is fundamentally broken in your contract's logic, which is exactly what `assert()` is designed to catch.

Quiz 4: Understanding Error Gas Usage

Instructions: Neri is optimizing her contract's gas usage. Which statement about gas consumption in error handling is correct?

```
pragma solidity ^0.8.0;

contract GasUsageExample {
    function exampleA(uint256 value) public pure {
        require(value > 10, "Value too small");
        // More code here
    }

    function exampleB(uint256 value) public pure {
        assert(value > 10);
        // More code here
    }

    function exampleC(uint256 value) public pure {
        if (value <= 10) {
            revert("Value too small");
        }
        // More code here
    }
}
```

Which statement about gas consumption is correct?

- A) `require()` and `revert()` refund unused gas, while `assert()` consumes all gas
- B) `assert()` refunds unused gas, while `require()` and `revert()` consume all gas
- C) All three functions (`require()`, `assert()`, and `revert()`) consume the same amount of gas
- D) `require()` uses the least gas, followed by `revert()`, and `assert()` uses the most gas

Answer: A) `require()` and `revert()` refund unused gas, while `assert()` consumes all gas

Explanation: In Solidity, how errors are handled affects how much gas is used:

- `require()` and `revert()` are meant for recoverable errors (like checking user inputs). When they fail, they refund any unused gas back to the user. This is more user-friendly and appropriate for conditions that might legitimately fail.

- `assert()` is meant for catching serious, unexpected bugs that should never happen in your contract. When it fails, it consumes all gas as a way of signaling that something is fundamentally wrong with your contract. This makes it more expensive for users when it fails.

This is why you should use `require()` or `revert()` for validating user inputs or conditions that might legitimately fail, and reserve `assert()` for verifying internal invariants that should never be false if your code is correct.