Solidity Mappings Quiz

Quiz 1: Mapping Basics

Instructions: Neri is building a system to track user balances securely. Which of these correctly declares a mapping that connects user addresses to their balance amounts?

```
pragma solidity ^0.8.0;

contract UserBalances {
    // Option A
    address[] public userList;
    uint256[] public balanceList;

    // Option B
    mapping(address => uint256) public userBalances;

    // Option C
    mapping(uint256 => address) public userBalances;

    // Option D
    mapping<address, uint256> public userBalances;
}
```

Which option correctly declares a mapping from user addresses to balance amounts?

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

Answer: B) Option B

Explanation: Option B is correct because it creates a mapping that connects user addresses (the key) to their balance amounts (the value). The syntax `mapping(keyType => valueType) variableName` is the right way to declare a mapping in Solidity.

Think of it like a table with two columns: the left column has addresses, and the right column has balances. When you look up an address (the key), you instantly get the balance (the value) for that address.

Option A uses separate arrays, which would be inefficient for lookups. Option C has the relationship backwards (balance to address instead of address to balance). Option D uses incorrect syntax with angle brackets `<>` instead of the correct `=>` arrow.

Quiz 2: Working with Mappings

Instructions: Neri needs to update a user's balance in her tracking system. Which code correctly adds 100 to the user's existing balance?

```
pragma solidity ^0.8.0;

contract BalanceUpdater {
    mapping(address => uint256) public userBalances;

    // Option A
    function addToBalance1(address user) public {
        userBalances[user] = 100;
    }

    // Option B
    function addToBalance2(address user) public {
        userBalances[user] += 100;
    }

    // Option C
    function addToBalance3(address user) public {
        userBalances.push(user, 100);
    }

    // Option D
    function addToBalance4(address user) public {
        userBalances.add(user, 100);
    }
}
```

Which function correctly adds 100 to a user's existing balance?

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

Answer: B) Option B

Explanation: Option B correctly adds 100 to the user's existing balance using the `+=` operator. This takes whatever value is currently stored in the mapping for that user and adds 100 to it.

Option A would replace the user's balance with exactly 100, erasing whatever was there before. Options C and D use incorrect methods - mappings don't have `push` or `add` functions like arrays do. With mappings, you directly access and modify values using the `[key]` syntax.

Quiz 3: Default Values in Mappings

Instructions: Neri is curious what happens when you try to access a mapping key that hasn't been set yet. What will this code return for a brand new address that has never been added to the mapping?

```
pragma solidity ^0.8.0;
```

```
contract DefaultValues {
    mapping(address => uint256) public tokenBalances;
    mapping(address => bool) public isRegistered;
    mapping(address => string) public userNames;

    function checkNewUser(address user) public view returns (uint256, bool, string
memory) {
        // Return values for an address never used before
        return (tokenBalances[user], isRegistered[user], userNames[user]);
    }
}
```

What will the function return for a new address?

- A) `(null, null, null)`
- B) `(0, false, "")`
- C) Error: "Key does not exist"
- D) `(0, false, null)`

Answer: B) `(0, false, "")`

Explanation: In Solidity, mappings automatically provide a default value for any key that hasn't been explicitly set. The default value depends on the data type:

- For `uint256`, the default is `0`
- For `bool`, the default is `false`
- For `string`, the default is an empty string `""`

This means you don't need to check if a key exists before using it - you'll always get a value back. This is different from many other programming languages where accessing a non-existent key might cause an error. In Solidity, every possible key is considered to exist from the start, with its default value.

Quiz 4: Nested Mappings

Instructions: Neri wants to track which users have approved which other users to spend tokens on their behalf. Which of these correctly creates a nested mapping for this purpose?

```
pragma solidity ^0.8.0;

contract TokenApprovals {
    // Option A
    mapping(address => mapping(address => uint256)) public approvals;

    // Option B
    mapping(address, address) public approvals;

    // Option C
    mapping[address][address] public approvals;

    // Option D
```

```
    mapping(address => address => uint256) public approvals;
}
```

Which option correctly creates a nested mapping for token approvals?

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

Answer: A) Option A

Explanation: Option A correctly creates a nested mapping, which is a mapping inside another mapping. This structure allows us to track which users (`first address`) have approved which other users (`second address`) to spend a specific amount (`uint256`) of tokens.

Think of it like a spreadsheet where each cell has another spreadsheet inside it! The first address leads you to an inner mapping, and then the second address leads to the actual approval amount.

For example, if Alice (address1) approves Bob (address2) to spend 100 tokens:

```
approvals[aliceAddress][bobAddress] = 100;
```

Options B and D use incorrect syntax. Option C uses square brackets which is incorrect - Solidity uses parentheses for mapping declarations.