Solidity Data Types Quizzes

Quiz 1: Basic Data Types

Instructions: Examine the incomplete code snippet below and select the correct option to complete it properly.

```
pragma solidity ^0.8.0;

contract PalengkeInventory {
    _____ isStallOpen;
    _____ bananaPrice;
    _____ vendorName;
    _____ vendorWallet;
}
```

Choose the correct option to fill in the blanks in order:

- A) `uint256`, `string`, `bool`, `address`
- B) `bool`, `uint256`, `string`, `address`
- C) `bool`, `int256`, `bytes`, `uint256`
- D) `boolean`, `uint`, `string`, `wallet`

Answer: B) `bool`, `uint256`, `string`, `address`

Explanation: The correct data types are:

- `isStallOpen` should be a `bool` (boolean) since it represents a true/false state
- `bananaPrice` should be a `uint256` (unsigned integer) since prices are positive numbers
- `vendorName` should be a `string` to store text data
- `vendorWallet` should be an `address` to store an Ethereum address

Quiz 2: Understanding Integer Types

Instructions: Which of the following statements about integer types in Solidity is incorrect?

```
pragma solidity ^0.8.0;

contract IntegerExamples {
    uint8 small = 255;
    uint16 medium = 65535;
    uint256 large =
115792089237316195423570985008687907853269984665640564039457584007913129639 9;
    int256 negative = -100;
}
```

Choose the incorrect statement:

- A) `uint` is an alias for `uint256`
- B) `int8` can store values from -128 to 127
- C) `uint8` can overflow if assigned a value greater than 255
- D) `uint` can store negative values in special cases

Answer: D) `uint` can store negative values in special cases

Explanation: `uint` (unsigned integer) can never store negative values under any circumstances. That's why it's called "unsigned" - it doesn't have a sign bit. For negative values, you must use `int` (signed integer) types. In Solidity 0.8.0 and above, overflow checking is built-in and will revert transactions that would cause overflow.

Quiz 3: Working with Arrays

Instructions: Complete the code snippet to create an array of prices for fruits in a market stall.

```
pragma solidity ^0.8.0;

contract FruitMarket {
    // Array to store the prices of 5 different fruits
    _____ fruitPrices = [50, 30, 25, 60, 15];

    function updatePrice(uint256 index, uint256 newPrice) public {
        fruitPrices[index] = newPrice;
    }
}
```

Choose the correct option to fill in the blank:

- A) `uint256[]`
- B) `uint256[5]`
- C) `array<uint256>`
- D) `uint256[5] public`

Answer: D) `uint256[5] public`

Explanation: The correct declaration is `uint256[5] public`. This creates a fixed-size array of 5 elements, all of type `uint256`. The `public` visibility modifier automatically creates a getter function for the array. Fixed-size arrays in Solidity require the size to be specified at compile time, and we need it to be public so the `fruitPrices` array can be accessed externally.

Quiz 4: Understanding Mappings

Instructions: What is incorrect about the following code that uses a mapping to track vendor balances?

```
pragma solidity ^0.8.0;

contract PalengkeAccounting {
```

```
    mapping(address => uint256) public vendorBalances;

    function depositFunds() public payable {
        vendorBalances[msg.sender] += msg.value;
    }

    function _____ {
        vendorBalances[0] = 1000;
    }
}
```

Choose the correct option to identify what's incorrect:

- A) The mapping declaration syntax is wrong
- B) You cannot use integer keys in mappings, only address types
- C) Mappings don't have a length or size property and can't be iterated through
- D) Using `0` as a key in a mapping is invalid; you need a valid address

Answer: D) Using `0` as a key in a mapping is invalid; you need a valid address

Explanation: In this code, the error is trying to use `0` as an address key in the mapping. When using a mapping with an address key type, you must provide a valid Ethereum address (20 bytes). The value `0` is not a valid address format. The correct approach would be to use a proper address, such as `address(0)` if you want to refer to the zero address, or a specific address like `0x123...`.

Quiz 5: Converting Between Data Types

Instructions: Which of the following type conversions in Solidity is safe and will not result in data loss?

```
pragma solidity ^0.8.0;

contract TypeConversions {
    function convert() public pure {
        // Option A
        uint8 a = 100;
        uint256 b = a;

        // Option B
        uint256 c = 300;
        uint8 d = uint8(c);

        // Option C
        int8 e = 100;
        uint8 f = uint8(e);

        // Option D
        address addr = 0x123;
        uint256 g = uint256(uint160(addr));
    }
}
```

Choose the correct option:

- A) Option A: Converting uint8 to uint256
- B) Option B: Converting uint256 to uint8
- C) Option C: Converting int8 to uint8
- D) All conversions are safe

Answer: A) Option A: Converting uint8 to uint256

Explanation: Converting from a smaller integer type to a larger one (like uint8 to uint256) is always safe because the larger type can represent all values of the smaller type. Option B can lose data if the uint256 value is greater than 255 (the maximum for uint8). Option C can cause issues if the int8 contains a negative value. Option D (address to uint) is technically possible but requires proper understanding of how addresses are stored.

Quiz 6: String and Bytes

Instructions: Complete the code to properly declare a bytes variable to store a vendor ID efficiently.

```
pragma solidity ^0.8.0;

contract VendorRegistry {
    _____ public vendorId;

    function setVendorId(bytes memory _id) public {
        vendorId = _id;
    }
}
```

Choose the correct option to fill in the blank:

- A) `string`
- B) `bytes`
- C) `bytes32`
- D) `byte[]`

Answer: B) `bytes`

Explanation: For a variable-length byte array, the correct type is `bytes`. This is more gas-efficient than `string` when dealing with raw binary data. The `bytes32` type would limit the storage to exactly 32 bytes (fixed-size), which might not be flexible enough for variable-length IDs. `byte[]` is a dynamic array of individual bytes, which is less gas-efficient than using the `bytes` type for similar purposes.