Solidity Global Variables Quiz

Quiz 1: Understanding msg.sender

Instructions: Neri is creating a contract to track barangay donations. Which code correctly uses `msg.sender` to identify the donor?

```
pragma solidity ^0.8.0;

contract BarangayDonations {
    mapping(address => uint256) public donations;

    // Option A
    function donate() public payable {
        donations[msg.value] += msg.value;
    }

    // Option B
    function donate2() public payable {
        donations[msg.sender] += msg.value;
    }

    // Option C
    function donate3() public payable {
        donations[address(this)] += msg.value;
    }

    // Option D
    function donate4() public payable {
        donations[block.timestamp] += msg.value;
    }
}
```

Which function correctly uses `msg.sender` to identify the donor?

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

Answer: B) Option B

Explanation: Option B correctly uses `msg.sender` to track the donor's address. `msg.sender` is a global variable that gives you the address of whoever called the function - in this case, the person making the donation.

The other options incorrectly use:

- Option A: Uses `msg.value` (the amount of Ether sent) as an address, which doesn't make sense

- Option C: Uses `address(this)` which is the contract's own address, not the donor's
- Option D: Uses `block.timestamp` (current block time) as an address, which doesn't make sense

Think of `msg.sender` like an automatic caller ID - it tells your contract exactly who's calling it. This is essential for tracking who did what in your contract.

Quiz 2: Understanding msg.value

Instructions: Neri is building a donation tracker for barangay projects. Which statement about `msg.value` is correct?

```
pragma solidity ^0.8.0;

contract DonationTracker {
    mapping(address => uint256) public donations;

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

    function withdrawDonation() public {
        uint256 donatedAmount = donations[msg.sender];
        require(donatedAmount > 0, "No donations to withdraw");
        donations[msg.sender] = 0;
        payable(msg.sender).transfer(donatedAmount);
    }
}
```

Which statement about `msg.value` is correct?

- A) `msg.value` gives us the address of who called the function
- B) `msg.value` is the amount of Ether sent with the function call in wei
- C) `msg.value` shows how many times a function has been called
- D) `msg.value` automatically transfers Ether from the caller to the contract

Answer: B) `msg.value` is the amount of Ether sent with the function call in wei

Explanation: `msg.value` gives us the amount of Ether (in wei units) that was sent along with the function call. For `msg.value` to work, the function must be marked with the `payable` keyword as shown in the `donate()` function.

Wei is the smallest unit of Ether (1 Ether = 10^18 wei), similar to how centavos are the smallest unit of the Philippine peso.

Other clarifications:

- The caller's address is given by `msg.sender`, not `msg.value`
- `msg.value` doesn't count function calls
- The transfer of Ether happens automatically because of the `payable` keyword, but `msg.value` just tells you how much was sent

This is important because it allows your contract to accept and track payments from users.

Quiz 3: Understanding block.timestamp

Instructions: Neri is creating a time-sensitive contract for a barangay event. What does `block.timestamp` represent in this code?

```
pragma solidity ^0.8.0;

contract EventTickets {
    uint256 public ticketPrice = 0.01 ether;
    uint256 public eventTime;
    mapping(address => bool) public ticketHolders;

    constructor() {
        // Event happens 7 days from contract creation
        eventTime = block.timestamp + 7 days;
    }

    function buyTicket() public payable {
        require(msg.value >= ticketPrice, "Not enough ETH sent");
        require(block.timestamp < eventTime, "Event has already started");

        ticketHolders[msg.sender] = true;
    }

    function checkIfEventStarted() public view returns (bool) {
        return block.timestamp >= eventTime;
    }
}
```

What does `block.timestamp` represent?

- A) The time when the contract was created
- B) The current time when the function is called (in seconds since Unix epoch)
- C) The block number of the current block
- D) The time when the event starts

Answer: B) The current time when the function is called (in seconds since Unix epoch)

Explanation: `block.timestamp` gives you the current timestamp of the block in which your transaction is being processed. It represents the current time (in seconds since January 1, 1970 - known as the Unix epoch).

In this contract:

1. When the contract is created, it sets `eventTime` to be 7 days after the current time
2. The `buyTicket` function checks if the current time is before the event time
3. The `checkIfEventStarted` function compares the current time with the event time

Important notes about `block.timestamp`:

- It's provided by the miners, so it can't be 100% trusted for exact time-critical applications
- It only updates when a new block is created
- It's useful for general time-based logic that doesn't require precision to the second

This global variable is useful for creating time-dependent features like deadlines, time locks, or scheduled events.

Quiz 4: Understanding block and tx Global Variables

Instructions: Neri is creating a contract for tracking barangay project funding. Which code snippet correctly uses global variables?

```
pragma solidity ^0.8.0;

contract ProjectFunding {
    address public projectOwner;

    struct Donation {
        address donor;
        uint256 amount;
        uint256 blockNumber;
        uint256 timestamp;
        bytes32 transactionHash;
    }

    Donation[] public donations;

    constructor() {
        projectOwner = msg.sender;
    }

    // Option A
    function donate1() public payable {
        donations.push(Donation(
            msg.sender,
            msg.value,
            block.number,
            block.timestamp,
            blockhash(block.number)
        ));
    }

    // Option B
    function donate2() public payable {
        donations.push(Donation(
            msg.sender,
            msg.value,
            block.number,
            block.timestamp,
            tx.origin
        ));
    }
```

```
    // Option C
    function donate3() public payable {
        donations.push(Donation(
            msg.sender,
            msg.value,
            block.number,
            block.timestamp,
            bytes32(block.difficulty)
        ));
    }

    // Option D
    function donate4() public payable {
        donations.push(Donation(
            msg.sender,
            msg.value,
            block.number,
            block.timestamp,
            blockhash(block.number - 1)
        ));
    }
}
```

Which function correctly uses global variables without errors?

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

Answer: D) Option D

Explanation: Option D correctly uses global variables without errors. Here's why:

1. It correctly stores:
    - The caller's address (`msg.sender`)
    - The amount of Ether sent (`msg.value`)
    - The current block number (`block.number`)
    - The current timestamp (`block.timestamp`)
    - And importantly, it uses `blockhash(block.number - 1)` to get the hash of the previous block

The other options have issues:

- Option A tries to get `blockhash(block.number)` for the current block, which isn't available yet
- Option B tries to store `tx.origin` (an address) in a `bytes32` field, which causes a type error
- Option C tries to convert `block.difficulty` to `bytes32`, which may cause issues (and also note that `block.difficulty` was deprecated in the London hard fork)

The global variables in Solidity provide important context information about the blockchain and the current transaction, which can be used for many purposes including logging, access control, and time-sensitive

operations.