Solidity File Imports Quiz

Quiz 1: Basic File Imports

Instructions: Neri is organizing her code into multiple files to better manage the fight against Hackana. Which syntax correctly imports a local file in Solidity?

```
pragma solidity ^0.8.0;

// Option A
import MathLibrary.sol;

// Option B
import "./MathLibrary.sol";

// Option C
include "./MathLibrary.sol";

// Option D
require "./MathLibrary.sol";
```

Which import statement is correct for importing a local file?

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

Answer: B) Option B

Explanation: Option B uses the correct syntax for importing a local Solidity file. In Solidity:

- The `import` keyword is used (not `include` or `require`)
- Relative paths use the `./` prefix to indicate the current directory
- The full filename including the `.sol` extension is specified

This import pattern allows you to organize related code into separate files and build modular, maintainable smart contracts. The imported file's contents will be included in the compiled contract.

Quiz 2: Import Variations

Instructions: Neri is working with different import techniques. Which import statement is NOT valid in Solidity?

```
pragma solidity ^0.8.0;

// Option A
```

```
import * as MathUtils from "./MathLibrary.sol";

// Option B
import { add, subtract } from "./MathLibrary.sol";

// Option C
import "./MathLibrary.sol" as MathUtils;

// Option D
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Which import statement is NOT valid in Solidity?

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

Answer: C) Option C

Explanation: Option C shows an invalid import syntax in Solidity.

The valid import patterns in Solidity are:

- Direct import: `import "./FileName.sol";` (as in the basic example)
- Specific symbol import: `import { Symbol1, Symbol2 } from "./FileName.sol";` (Option B)
- Global symbol import: `import * as Namespace from "./FileName.sol";` (Option A)
- Package import: `import "@package-name/path/to/File.sol";` (Option D for OpenZeppelin)

Option C tries to use the `as` keyword after the filename, which is not valid syntax in Solidity. The correct way to create an alias would be like Option A, with the `as` keyword between `*` and the namespace name.

Quiz 3: Import Best Practices

Instructions: Neri is importing code from the OpenZeppelin library. Which code example demonstrates the best practice for importing third-party libraries?

```
pragma solidity ^0.8.0;

// Option A
import "../node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken1 is ERC20 {
    constructor() ERC20("Token", "TKN") {}
}

// Option B
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken2 is ERC20 {
    constructor() ERC20("Token", "TKN") {}
```

```
}

// Option C
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken3 is ERC20 {
    constructor() ERC20("Token", "TKN") {}
}

// Option D
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken4 {
    ERC20 public token;

    constructor() {
        token = new ERC20("Token", "TKN");
    }
}
```

Which option demonstrates the best practice for importing and using third-party libraries?

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

Answer: C) Option C

Explanation: Option C demonstrates the best practice for importing and using third-party libraries by using specific symbol imports.

The benefits of this approach:

- It explicitly imports only what's needed (`ERC20`) using the `{ Symbol }` syntax
- It uses the package name directly without hardcoding node_modules paths
- It correctly extends the imported contract using inheritance

The other options have issues:

- Option A uses a hardcoded path to node_modules, which is brittle and may not work in all environments
- Option B does a global import which works but is less explicit about what's being used
- Option D doesn't use inheritance (which is the proper way to use ERC20) and instead creates a new instance

Using specific symbol imports as shown in Option C leads to more maintainable and clearer code by making the dependencies explicit.

Quiz 4: File Organization and Imports

Instructions: Neri is organizing a complex contract system with multiple files. Which approach correctly uses imports for a well-structured project?

```
// File: contracts/interfaces/IStorage.sol
pragma solidity ^0.8.0;

interface IStorage {
    function setValue(uint256 value) external;
    function getValue() external view returns (uint256);
}

// File: contracts/libraries/MathUtils.sol
pragma solidity ^0.8.0;

library MathUtils {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        return a + b;
    }
}

// File: contracts/implementation/Storage.sol
pragma solidity ^0.8.0;

// Import statements for Storage.sol - Select the best option
// Option A: No imports

// Option B:
import "../interfaces/IStorage.sol";

// Option C:
import "../libraries/MathUtils.sol";
import "../interfaces/IStorage.sol";

// Option D:
import { IStorage } from "../interfaces/IStorage.sol";
import { MathUtils } from "../libraries/MathUtils.sol";

contract Storage is IStorage {
    using MathUtils for uint256;
    uint256 private _value;

    function setValue(uint256 value) external override {
        _value = value;
    }

    function getValue() external view override returns (uint256) {
        return _value;
    }

    function incrementValue(uint256 amount) external {
        _value = _value.add(amount);
```

```
    }
}
```

Which import option is best for the Storage.sol contract?

- A) Option A: No imports
- B) Option B: Only import the interface
- C) Option C: Import both files with general imports
- D) Option D: Import both files with specific symbol imports

Answer: D) Option D: Import both files with specific symbol imports

Explanation: Option D demonstrates the best practices for import organization in a Solidity project:

1. It imports both necessary dependencies (the interface and library)
2. It uses specific symbol imports with the `{ Symbol }` syntax
3. It uses relative paths with `../` notation to navigate up one directory level
4. It clearly identifies which symbols are being imported from each file

This approach offers several advantages:

- Clear dependency identification
- Better IDE support for code navigation
- Avoids namespace pollution
- Makes it obvious which external code the contract depends on

The Storage contract needs both the `IStorage` interface (since it's implementing it with `is IStorage`) and the `MathUtils` library (since it's using it with `using MathUtils for uint256`). Option D imports both with the clearest and most maintainable syntax.