PassQueen
a.k.a dumpsqueen.com

BTA Certified Blockchain Developer - Ethereum

Blockchain CBDE

Version Demo

Total Demo Questions: 14

Total Premium Questions: 141

Buy Premium PDF

https://passqueen.com

support@passqueen.com

QUESTION NO: 1

Having a bug-bounty program early on:

A. can help to engage the community in testing your smart contracts and therefore help to find bugs early.

B. might be a burden as it is an administrative overhead mainly.

C. is completely useless. Who wants to test beta-ware software? It's better to start with the bug-bounty program after the contract is released on the main-net.

ANSWER: A

Explanation:

"can help to engage the community in testing your smart contracts and therefore help to find bugs early." is correct. A bug-bounty program gives independent security researchers a clear, authorized channel to examine a protocol's smart contracts and responsibly report vulnerabilities. Starting this process early—while software is in development, testnet deployment, or a controlled beta phase—broadens the review beyond the internal engineering team and can identify issues that conventional testing, code review, and audits may not uncover. Early reports are particularly valuable because the team can validate and remediate findings before substantial user funds, integrations, or irreversible on-chain state are at risk. An effective program should define scope, disclosure procedures, severity assessment, reward criteria, and safe-harbor terms so researchers know how to test without disrupting live systems. Bug bounties are a complementary security practice rather than a substitute for secure development, automated testing, professional audits, monitoring, and incident response planning. Ethereum's smart-contract security guidance emphasizes testing and review before deployment, while the Ethereum bug-bounty program illustrates the use of coordinated vulnerability disclosure to improve ecosystem security. See Ethereum smart contract security and Ethereum Bug Bounty Program.

QUESTION NO: 2

To Iterate through a mapping, you:

A. can use the length parameter of the mapping.

B. you need an external helper variable.

C. you cannot iterate any mapping to make the overall language design more safe.

ANSWER: B

Explanation:

The correct answer is *you need an external helper variable.* Solidity mappings are deliberately implemented as key-value storage with no stored list of keys and no length metadata. A mapping can return a value for any key of its declared key type, including keys that were never explicitly assigned, because such reads produce the type's default value. Consequently, the contract has no native way to discover which keys have actually been inserted, and it cannot enumerate a mapping directly.

When an application needs iteration, it must maintain the required key information separately as state—for example, by appending each newly used address or identifier to an array, while the mapping remains the efficient lookup structure. The array (or another purpose-built enumerable data structure) provides the helper state that can be traversed by index. Implementations should also consider gas costs and avoid unbounded on-chain loops where the key collection could grow too large; pagination or off-chain indexing may be more suitable for large datasets. Solidity's documentation explicitly states that mappings do not have a length or a concept of a key or value being set, and therefore cannot be iterated without auxiliary key tracking. See the Solidity mappings documentation and the OpenZeppelin enumerable mapping utilities.

QUESTION NO: 3

Proof of Work (PoW) vs. Proof of Stake.

A. PoW is computationally intensive and requires substantial energy. Miners are rewarded for mining a block and incorporating transactions.

B. PoW is better than PoS, because with PoS we increase the amount of energy spent on the network.

C. PoS is mining with specialized new hardware that has to be purchased with a stack of Ether in the network. Hence the Name: Proof of Stake, which derives from Stack.

ANSWER: A

Explanation:

“PoW is computationally intensive and requires substantial energy. Miners are rewarded for mining a block and incorporating transactions.” correctly describes the central operational characteristic of Proof of Work. In a PoW system, miners compete to find a valid block by performing repeated cryptographic hash calculations. Because success depends on computational work, participants typically use significant processing capacity and therefore consume electricity. This expenditure is intentional: it makes rewriting transaction history or attempting to control block production expensive, providing economic resistance to attacks.

When a miner finds a block that satisfies the network’s difficulty requirement and the block is accepted by the network, the protocol can compensate that miner through a block subsidy, transaction-fee revenue, or both, depending on the blockchain’s rules. Thus, rewards give miners an economic incentive to devote computing resources to validating and extending the chain. Ethereum formerly used PoW before its transition to Proof of Stake, and its documentation describes PoW mining as a process involving computational work, energy use, and miner rewards. See the Ethereum Proof-of-Work documentation and the Ethereum consensus mechanisms overview.

QUESTION NO: 4

When defining a new datatype:

A. its best to use a contract with public storage variables, so it can be used like a class.

B. it's best to use a struct, which is cheaper than deploying a new contract.

C. it's not possible to generate new datatypes in Solidity.

ANSWER: B

Explanation:

“it's best to use a struct, which is cheaper than deploying a new contract.” is correct because Solidity structs are language-level composite types designed to group related values into a single custom data model. A struct declaration does not create a separate on-chain contract or require a contract-creation transaction. Instead, it defines the layout that Solidity uses when a struct is declared in storage, memory, or calldata, subject to the applicable language rules. This makes structs the natural choice for representing records such as a user profile, product, order, or asset attributes within an existing contract.

Deploying a contract creates a distinct account with its own bytecode and storage namespace, and contract creation consumes gas for initialization and code deployment. When the goal is merely to define and store a related set of fields under an existing contract’s logic, a struct avoids that deployment overhead while keeping the data model clear and type-safe. Solidity also permits structs to be used in mappings and arrays, enabling contracts to manage collections of application-specific records efficiently. The Solidity documentation describes structs as custom types that can contain multiple variables, and its storage-layout documentation explains how such state data is arranged in contract storage. See Solidity documentation: Structs and Solidity documentation: Storage layout.

QUESTION NO: 5

Finish the sentence: The Library Web3.js is …:

A. useful when developing distributed applications with HTML and JavaScript, because it already implements the abstraction of the JSON-RPC interface of Ethereum Nodes.

B. necessary when developing distributed applications with HTML and JavaScript, because the proprietary JSON-RPC interface of Ethereum Nodes is a closed source.

ANSWER: A

Explanation:

Web3.js is useful when developing distributed applications with HTML and JavaScript because it provides a JavaScript API for communicating with Ethereum-compatible nodes. Ethereum nodes expose functionality through JSON-RPC methods, such as querying accounts and balances, reading contract state, submitting signed transactions, and obtaining block or transaction data. Rather than requiring an application developer to manually construct JSON-RPC request payloads, manage provider connections, and decode common responses for every interaction, Web3.js supplies higher-level modules and convenience methods for those tasks.

In a browser-based decentralized application, a provider can connect Web3.js to a wallet or injected Ethereum provider; in server-side software, it can connect through an HTTP, WebSocket, or IPC endpoint. The library also includes contract abstractions based on an ABI, allowing JavaScript code to call contract functions, estimate gas, and listen for events in a structured way. This abstraction is precisely why the library is useful for HTML- and JavaScript-based distributed application development. See the Web3.js documentation and Ethereum's JSON-RPC API documentation.

QUESTION NO: 6

You need to use _____ to get the address that initiated the transaction.

A. Tx.origin

B. Msg.sender

ANSWER: A

Explanation:

`Tx.origin` is correct because Solidity exposes `tx.origin` as the address of the externally owned account that originally initiated the transaction. This value is retained throughout the full call chain of that transaction. For example, if a wallet account sends a transaction to one contract and that contract calls another contract, `tx.origin` remains the wallet account's address in both contracts. Therefore, it identifies the transaction's original initiator rather than the contract or account making the most immediate call.

This behavior is distinct from the contextual caller information used for normal contract interactions: the original-origin value represents the top-level account responsible for starting execution. Solidity documents `tx.origin` among the global variables that provide transaction properties, specifically describing it as the sender of the transaction. See the Solidity language documentation on units and globally available variables and the Ethereum execution-layer specification's definition of the transaction origin field at EIP-2929.

QUESTION NO: 7

A Solidity function marked `payable`:

A. can accept ETH sent with a call and can access the amount through `msg.value`.

B. can only be called by the account that deployed the contract.

C. automatically forwards all received ETH to the contract owner.

ANSWER: A

Explanation:

A function must be marked `payable` if it is intended to accept a non-zero ETH value as part of a call. Within such a function, the amount sent with the call is available through `msg.value`. A contract can use this value for deposits, purchases, auctions, or other application-specific payment logic.

Calling a non-payable function with a non-zero value causes the call to revert. Marking a function payable does not automatically transfer value to another account or validate a payment amount; the contract code must implement those rules. Solidity documents mutability restrictions and `msg.value` in its state mutability and global variables references.

QUESTION NO: 8

If a User calls contract A and that calls Contract B, then msg.sender in Contract B will contain the address of:

A. the User.

B. contract A

ANSWER: B

Explanation:

contract A is correct. In Solidity, `msg.sender` is the address of the immediate caller for the current external call frame. The user first sends a transaction to contract A, so within contract A, `msg.sender` is the user's address. When contract A subsequently makes an external call to contract B, contract A becomes the immediate caller of contract B. Therefore, while contract B executes that call, `msg.sender` contains contract A's address.

This behavior is fundamental to Ethereum's call model and is especially important for authorization design. A contract cannot infer the original externally owned account merely from `msg.sender` after an intervening contract call. If the original caller must be intentionally propagated, the calling contract must pass it as an explicit parameter, and the receiving contract must apply appropriate trust and validation rules. Solidity documents `msg.sender` as the sender of the message for the current call, which is why its value changes as execution moves through contracts. See the Solidity global variables documentation at https://docs.soliditylang.org/en/latest/units-and-global-variables.html#block-and-transaction-properties and the Solidity contracts documentation at https://docs.soliditylang.org/en/latest/contracts.html.

QUESTION NO: 9

What's the correct scientific notation?

A. 1 Ether = 10^18 wei, 10^9 Gwei, 10^3 Finney

B. 1 Ether = 10^19 wei, 10^13 Gwei, 10^3 Finney

C. 1 Ether = 10^16 wei, 10^13 Gwei, 10^3 Finney

D. 1 Ether = 10^18 wei, 10^6 Gwei, 10^6 Finney

ANSWER: A

Explanation:

1 Ether = 10^18 wei, 10^9 Gwei, 10^3 Finney is correct because wei is Ethereum's smallest native currency denomination. One ether is defined as exactly 1,000,000,000,000,000,000 wei, which is 10^18 wei. Gwei (short for gigawei) equals 10^9 wei; therefore, dividing 10^18 wei by 10^9 wei per Gwei gives 10^9 Gwei in one ether. Finney is a historical ether denomination equal to 10^15 wei, or 0.001 ether. Consequently, one ether contains 1,000 finney, expressed as 10^3 Finney. These unit relationships are important when interpreting smart-contract values, account balances, and gas prices: transaction and contract values are ultimately represented in wei, while gas prices are commonly displayed in Gwei for readability. Ethereum's documentation lists the denomination table and identifies wei as the base unit of ether. See the Ethereum EVM denomination reference and the Ethereum gas documentation.

QUESTION NO: 10

Assert is used:

A. to check internal states that should never happen.

B. to check input arguments from users.

ANSWER: A

Explanation:

`to check internal states that should never happen.` is correct because Solidity's `assert` is intended for conditions that represent internal errors, broken invariants, or impossible program states. An invariant is a property the contract's logic should preserve throughout execution, such as an accounting relationship that must always hold after a state update. If such a condition fails, it indicates a defect in the contract logic rather than an ordinary condition caused by expected external interaction.

When an `assert` condition evaluates to false, Solidity triggers a `Panic(uint256)` error. This behavior communicates that execution reached a state the developer considered impossible and can help identify bugs during testing, auditing, and monitoring. Solidity distinguishes this use from validation of values supplied through normal contract calls: assertions are reserved for checks that demonstrate the contract's own implementation has violated an assumed invariant. Writing assertions for these critical internal properties makes intended safety guarantees explicit and allows tools and tests to verify them. See the Solidity documentation on Panic via assert and the Errors and revert statement reference.

QUESTION NO: 11

Block Timestamp:

A. the timestamp is based on the time zone of the miner, that is why it changes the difficulty continuously to reflect network latency.

B. the timestamp can't be influenced by a miner and is generally considered safe to be used for randomness on the blockchain.

C. the timestamp can be influenced by a miner to a certain degree but it's always independent from the time-zone.

ANSWER: C

Explanation:

The timestamp can be influenced by a miner to a certain degree but it's always independent from the time-zone. is correct because an Ethereum block timestamp is represented as a Unix timestamp: an integer count of seconds since the Unix epoch. Unix timestamps are a universal time representation and do not encode, derive from, or vary with a miner's local time zone. A node interprets the same numeric timestamp identically regardless of where it runs.

In proof-of-work Ethereum, the block producer supplied the timestamp as part of the block header, so it was not an exact, independently trusted wall-clock reading. Protocol validity rules constrained the value, but a producer could choose a permissible time within those bounds. Consequently, timestamp-dependent contract logic should tolerate limited timestamp variation and should not treat a block timestamp as an unpredictable randomness source. Ethereum's security guidance specifically notes that block timestamps can be influenced by validators and should not be relied on for randomness. In Ethereum's current proof-of-stake design, execution-payload timestamps are tied to consensus-slot timing, providing stronger protocol scheduling, while the timestamp itself remains an epoch-based, time-zone-independent value.

See the Solidity documentation on block.timestamp and Ethereum's proof-of-stake consensus documentation.

QUESTION NO: 12

Ethereum Nodes:

A. must implement the Ethereum protocol and external access can only be done via the proprietary Ethereum Libraries like Web3.js.

B. must implement the Ethereum Protocol and a JSON-RPC to talk with clients.

C. must implement Web3.js to interact with Websites.

ANSWER: B

Explanation:

“must implement the Ethereum Protocol and a JSON-RPC to talk with clients.” is correct because an Ethereum node is software that implements the Ethereum execution-layer protocol: it validates and propagates transactions, participates in block and chain validation according to its role and synchronization mode, and maintains the data needed to serve the network. To make node functionality available to wallets, decentralized applications, developer tools, and other client software, Ethereum execution clients commonly expose a JSON-RPC API. This API provides standardized methods for tasks such as querying account balances, retrieving blocks and transaction receipts, submitting signed transactions, estimating gas, and interacting with deployed smart contracts.

JSON-RPC is the conventional interface used by Ethereum client implementations, whether it is made available locally over IPC or HTTP, or remotely through HTTP and WebSocket endpoints. Application libraries can use this interface, but the node’s external interface is not limited to one proprietary library. Ethereum.org describes JSON-RPC APIs as the way applications interact with Ethereum nodes, while the execution-API specification defines the RPC methods and expected behavior across compatible clients. See the Ethereum JSON-RPC API documentation and the Ethereum Execution APIs specification.

QUESTION NO: 13

All low-level functions on the address, so address.send(), address.call.valueQQ, address.callcode and address.delegatecall:

A. are interrupting execution on error, because they throw an exception.

B. continuing execution on error silently, which is the reason why they are so dangerous.

C. returning Booleans to indicate an error during execution.

D. .send() throws an exception, while the other functions are returning Booleans during execution to indicate an error.

ANSWER: C

Explanation:

returning Booleans to indicate an error during execution. is correct in the historical Solidity context used by this question. Solidity’s low-level address operations—including `send`, `call`, `callcode`, and `delegatecall`—report whether the external operation succeeded through a boolean status value rather than automatically propagating the failure as an exception. Consequently, safe contract code must explicitly inspect that result and take appropriate action when it indicates failure. This behavior is especially important for value transfers and arbitrary external calls, where failure can arise from the target reverting, running out of gas, or rejecting a payment.

In current Solidity versions, `send` still returns `bool`; the call-family operations return a tuple whose first item is the boolean success status and whose second item contains returned data. Thus, the core principle tested remains the same: code using low-level calls must handle the success indicator deliberately. Solidity documents the status return from low-level calls in its address type member reference and recommends checking return values when using these operations in its security considerations.

QUESTION NO: 14

Externally Owned Accounts (EoA):

A. are changing their address every time a Transaction is sent because of the nonce.

B. are keeping their address, but on the blockchain a nonce is increased every time they send a transaction to avoid replay attacks.

ANSWER: B

Explanation:

Externally owned accounts keep a stable address derived from their public key; sending a transaction does not create or assign a new address. Each account instead maintains a transaction nonce, which is a counter stored as part of that account's state. For an externally owned account, a valid new transaction must use the account's current nonce. Once the transaction is processed, the account nonce increases by one.

This sequencing provides a unique position for each transaction originating from that account and lets nodes determine the intended order when multiple transactions are submitted. It also prevents a previously accepted signed transaction from being executed repeatedly: after its nonce has been consumed, a replay using that same nonce is no longer valid against the updated account state. The nonce is therefore separate from the account address: the address identifies the sender, while the nonce identifies the next transaction sequence number for that sender. Ethereum's account documentation describes externally owned accounts as accounts controlled by private keys and identifies the nonce as the transaction count, while the execution-layer specification defines transaction validation against the sender's nonce. See Ethereum.org: Accounts and Ethereum Execution Layer Specification.