Solidity Basics – Quiz

Cryptoeconomics

Question 1

What is the primary purpose of the `pragma solidity` directive at the top of a Solidity file?

A. To import external libraries
B. To specify which compiler version(s) should be used
C. To declare the contract name
D. To define the license

Question 1

What is the primary purpose of the `pragma solidity` directive at the top of a Solidity file?

A. To import external libraries

B. To specify which compiler version(s) should be used

C. To declare the contract name

D. To define the license

Answer: B

The `pragma solidity` directive tells the Solidity compiler which version(s) are acceptable. For example, `pragma solidity ^0.8.20;` means "use at least 0.8.20 but not 0.9.0 or above".

Question 2

Which keyword is used to declare a new smart contract in Solidity?

A. class
B. program
C. contract
D. module

Question 2

Which keyword is used to declare a new smart contract in Solidity?

A. class
B. program
C. contract
D. module

Answer: C
In Solidity, the `contract` keyword defines a new contract, analogous to a class in object-oriented languages. Everything inside the curly braces belongs to that contract.

Question 3

What data type would you use in Solidity to store an Ethereum wallet address?

A. string
B. bytes32
C. address
D. uint160

Question 3

What data type would you use in Solidity to store an Ethereum wallet address?

A. string
B. bytes32
C. address
D. uint160

Answer: C
The `address` type is a 20-byte value specifically designed to hold Ethereum addresses. It has built-in methods like `.transfer()` and `.balance`.

Question 4

What does `uint256` represent in Solidity?

A. An unsigned integer with 256 bits (range 0 to $2^{256} - 1$)
B. A signed integer with 256 decimal digits
C. A floating-point number with 256-bit precision
D. An array of 256 unsigned integers

Question 4

What does `uint256` represent in Solidity?

A. An unsigned integer with 256 bits (range 0 to $2^{256} - 1$)

B. A signed integer with 256 decimal digits

C. A floating-point number with 256-bit precision

D. An array of 256 unsigned integers

Answer: A

`uint256` is an unsigned (non-negative) integer stored in 256 bits. It can hold values from 0 to approximately 1.16×10^{77}. It is the most commonly used numeric type in Solidity.

Question 5

Where is a state variable stored in Ethereum?

A. In the user's browser memory
B. On the blockchain (persistent storage)
C. In a temporary off-chain database
D. Only in the transaction calldata

Question 5

Where is a state variable stored in Ethereum?

A. In the user's browser memory
B. On the blockchain (persistent storage)
C. In a temporary off-chain database
D. Only in the transaction calldata

Answer: B
State variables are permanently stored on the Ethereum blockchain. Writing to them costs gas; reading them from outside the contract is free. They persist between function calls.

Question 6

What does the `public` visibility keyword do when applied to a state variable?

A. Prevents the variable from ever being changed
B. Allows only the owner to read the variable
C. Automatically generates a getter function so anyone can read it
D. Makes the variable free to write to

Question 6

What does the `public` visibility keyword do when applied to a state variable?

A. Prevents the variable from ever being changed

B. Allows only the owner to read the variable

C. Automatically generates a getter function so anyone can read it

D. Makes the variable free to write to

Answer: C

When you declare `uint256 public storedNumber;`, Solidity automatically generates a getter function named `storedNumber()` that anyone can call to read the value. No extra code is needed.

Question 7

What keyword marks a function as readable without spending gas (when called externally)?

A. public
B. view
C. pure
D. constant

Question 7

What keyword marks a function as readable without spending gas (when called externally)?

A. public
B. view
C. pure
D. constant

Answer: B
A `view` function reads from state but does not modify it. External calls to `view` functions do not cost gas because no transaction is needed — only a local node query.

Question 8

What distinguishes a `pure` function from a `view` function in Solidity?

A. `pure` functions cost more gas than `view` functions
B. `pure` functions cannot read OR modify state; `view` functions can read state
C. `pure` functions can only be called by the contract owner
D. `pure` functions always return a boolean

Question 8

What distinguishes a `pure` function from a `view` function in Solidity?

A. `pure` functions cost more gas than `view` functions

B. `pure` functions cannot read OR modify state; `view` functions can read state

C. `pure` functions can only be called by the contract owner

D. `pure` functions always return a boolean

Answer: B

`view` functions can read state variables but not modify them. `pure` functions neither read nor modify state — they depend only on their input parameters (e.g., a math helper). Both are free to call externally.

Question 9

What does Solidity compile into before it can run on Ethereum?

A. JavaScript bytecode
B. Machine code for x86 CPUs
C. EVM bytecode
D. WebAssembly (WASM)

Question 9

What does Solidity compile into before it can run on Ethereum?

A. JavaScript bytecode
B. Machine code for x86 CPUs
C. EVM bytecode
D. WebAssembly (WASM)

Answer: C
Solidity source code is compiled by `solc` into EVM (Ethereum Virtual Machine) bytecode. The EVM is a sandboxed virtual machine that runs identically on every Ethereum node.

Question 10

In Solidity, what is a `mapping`?

A. A sorted list of values

B. A key-value store where you look up a value by a key

C. A function that transforms one type to another

D. A loop that iterates over a collection

Question 10

In Solidity, what is a `mapping`?

A. A sorted list of values

B. A key-value store where you look up a value by a key

C. A function that transforms one type to another

D. A loop that iterates over a collection

Answer: B

`mapping(address => uint256)` is a hash map (key-value store). Given an address key, you instantly retrieve the associated uint256 value. Mappings cannot be iterated and do not track which keys exist.

Question 11

In the SimpleStorage contract, what does `storedNumber = _num;` inside a function do?

A. Reads the current value of storedNumber
B. Creates a new local variable
C. Writes the argument value to the persistent state variable
D. Emits an event with the new number

Question 11

In the SimpleStorage contract, what does `storedNumber = _num;` inside a function do?

A. Reads the current value of storedNumber

B. Creates a new local variable

C. Writes the argument value to the persistent state variable

D. Emits an event with the new number

Answer: C

This assignment updates the state variable `storedNumber` on the blockchain with the value passed as `_num`. Every write to a state variable costs gas and is recorded permanently.

Question 12

What special variable holds the address of whoever called the current function?

A. block.sender
B. tx.origin
C. msg.sender
D. address.caller

Question 12

What special variable holds the address of whoever called the current function?

A. block.sender

B. tx.origin

C. msg.sender

D. address.caller

Answer: C

`msg.sender` is a global variable automatically set to the Ethereum address that initiated the current call. It is widely used for access control (e.g., `require(msg.sender == owner)`).

Question 13

What is the role of the `constructor` function in a Solidity contract?

A. It is called every time any function in the contract runs
B. It runs exactly once when the contract is deployed to set up initial state
C. It allows the contract to receive ETH payments
D. It destroys the contract when called

Question 13

What is the role of the `constructor` function in a Solidity contract?

A. It is called every time any function in the contract runs

B. It runs exactly once when the contract is deployed to set up initial state

C. It allows the contract to receive ETH payments

D. It destroys the contract when called

Answer: B

The `constructor` runs exactly once at deployment and never again. It is used to set initial values, such as assigning `totalSupply` tokens to `msg.sender` in an ERC-20 token contract.

Question 14

What does `require(condition, "error message")` do in Solidity?

A. Logs the condition result to an event
B. Reverts the transaction with an error message if the condition is false
C. Pauses the contract until the condition becomes true
D. Sends a refund to the caller if the condition fails

Question 14

What does `require(condition, "error message")` do in Solidity?

A. Logs the condition result to an event

B. Reverts the transaction with an error message if the condition is false

C. Pauses the contract until the condition becomes true

D. Sends a refund to the caller if the condition fails

Answer: B

`require` checks a condition. If the condition is false, the entire transaction is reverted (all state changes undone) and the error message is returned to the caller. Gas spent up to that point is consumed.

Question 15

In the ERC-20 `transfer` function, why must `require(balanceOf[msg.sender] >= _amount)` appear BEFORE updating balances?

A. Solidity requires checks before any state change for syntactic reasons

B. Following the Checks-Effects-Interactions pattern prevents reentrancy attacks

C. The ERC-20 standard mandates this exact order in its specification

D. Placing checks after would cause a compiler error

Question 15

In the ERC-20 `transfer` function, why must `require(balanceOf[msg.sender] >= _amount)` appear BEFORE updating balances?

A. Solidity requires checks before any state change for syntactic reasons

B. Following the Checks-Effects-Interactions pattern prevents reentrancy attacks

C. The ERC-20 standard mandates this exact order in its specification

D. Placing checks after would cause a compiler error

Answer: B

The Checks-Effects-Interactions (CEI) pattern requires: (1) validate inputs/conditions first, (2) update state, (3) call external contracts. Checking balance before updating it prevents reentrancy attacks where a malicious contract could drain funds by re-entering before balances are updated.

Question 16

What is an `event` in Solidity used for?

A. To call external contracts
B. To store data permanently on-chain at low cost for off-chain listeners
C. To replace state variables
D. To trigger automatic payments

Question 16

What is an `event` in Solidity used for?

A. To call external contracts

B. To store data permanently on-chain at low cost for off-chain listeners

C. To replace state variables

D. To trigger automatic payments

Answer: B

Events emit data to the transaction log — a cheap storage area that off-chain apps (like wallets and block explorers) can listen to. Events are not accessible from within smart contracts but are much cheaper than state variable storage.

Question 17

What does the `indexed` keyword on an event parameter do?

A. Stores the parameter in a sorted array for fast lookup
B. Makes the parameter searchable and filterable in event logs
C. Prevents the parameter from being read by external apps
D. Doubles the gas cost of emitting the event

Question 17

What does the `indexed` keyword on an event parameter do?

A. Stores the parameter in a sorted array for fast lookup

B. Makes the parameter searchable and filterable in event logs

C. Prevents the parameter from being read by external apps

D. Doubles the gas cost of emitting the event

Answer: B

Marking an event parameter `indexed` adds it to a special "topic" in the log, allowing Ethereum nodes to efficiently filter events by that value. For example, you can find all `Transfer` events where `from` equals a specific address.

Question 18

In the ERC-20 allowance system, what must a token holder do BEFORE a third party can call `transferFrom` on their behalf?

A. Transfer half their tokens to the third party first

B. Call `approve(spenderAddress, amount)` to authorize the spending

C. Send a signed message to the token contract owner

D. Burn their existing tokens and re-mint them

Question 18

In the ERC-20 allowance system, what must a token holder do BEFORE a third party can call `transferFrom` on their behalf?

A. Transfer half their tokens to the third party first

B. Call `approve(spenderAddress, amount)` to authorize the spending

C. Send a signed message to the token contract owner

D. Burn their existing tokens and re-mint them

Answer: B

The ERC-20 allowance flow is: (1) token holder calls `approve(spender, amount)`, which sets `allowance[holder][spender] = amount`; (2) the spender (e.g., a DEX) can then call `transferFrom(holder, destination, amount)` up to the approved amount.

Question 19

What does `mapping(address => mapping(address => uint256)) public allowance` **represent?**

A. A list of all token holders and their balances

B. A nested map: for each owner, how much each spender is allowed to transfer

C. A two-dimensional array of token prices

D. A mapping from address pairs to timestamps

Question 19

What does `mapping(address => mapping(address => uint256)) public allowance` represent?

A. A list of all token holders and their balances

B. A nested map: for each owner, how much each spender is allowed to transfer

C. A two-dimensional array of token prices

D. A mapping from address pairs to timestamps

Answer: B

The nested mapping `allowance[owner][spender]` stores how many tokens `spender` is authorized to transfer from `owner`'s account. `allowance[alice][dex] = 100` means the DEX contract can move up to 100 tokens from Alice.

Question 20

In the `transferFrom` function, after transferring tokens, what must happen to the allowance?

A. The allowance must be deleted entirely
B. The allowance is automatically reset to zero by Solidity
C. The transferred amount must be subtracted from the allowance
D. The allowance remains unchanged to allow future transfers

Question 20

In the `transferFrom` function, after transferring tokens, what must happen to the allowance?

A. The allowance must be deleted entirely

B. The allowance is automatically reset to zero by Solidity

C. The transferred amount must be subtracted from the allowance

D. The allowance remains unchanged to allow future transfers

Answer: C

After a `transferFrom`, the code must explicitly reduce the allowance: `allowance[_from][msg.sender] -= _amount`. This prevents the spender from transferring more than the approved total. Solidity does not do this automatically.

Question 21

Which of the following is a required function in the ERC-20 token standard?

A. mint()
B. burn()
C. transfer(address to, uint256 amount)
D. freeze(address account)

Question 21

Which of the following is a required function in the ERC-20 token standard?

A. mint()
B. burn()
C. transfer(address to, uint256 amount)
D. freeze(address account)

Answer: C
`transfer()` is one of the six required ERC-20 functions. The others are `transferFrom()`, `approve()`, `allowance()`, `balanceOf()`, and `totalSupply()`. Functions like `mint()` and `burn()` are optional extensions.

Question 22

What is a `struct` in Solidity?

A. A special array type that auto-sorts its elements
B. A user-defined type that groups multiple variables together
C. A function that constructs new contracts
D. A type alias for mapping

Question 22

What is a `struct` in Solidity?

A. A special array type that auto-sorts its elements

B. A user-defined type that groups multiple variables together

C. A function that constructs new contracts

D. A type alias for mapping

Answer: B

A `struct` groups multiple fields under one name. For example, `struct Proposal { string description; uint256 voteCount; }` bundles text and a count together, letting you store complex records in arrays or mappings.

Question 23

What does a `modifier` do in Solidity?

A. It modifies the return type of a function
B. It is reusable code that wraps a function, typically for access control or validation
C. It compresses the function's bytecode
D. It makes a function callable only once

Question 23

What does a `modifier` do in Solidity?

A. It modifies the return type of a function

B. It is reusable code that wraps a function, typically for access control or validation

C. It compresses the function's bytecode

D. It makes a function callable only once

Answer: B

A modifier like `onlyOwner` can be applied to multiple functions. The modifier body runs first; `_;` marks where the original function body executes. This keeps access-control logic DRY (Don't Repeat Yourself).

Question 24

In the `onlyOwner` modifier, what does the `_;` symbol mean?

A. End the function and return immediately
B. A placeholder where the body of the modified function runs
C. Increment a counter variable
D. Skip the rest of the modifier

Question 24

In the `onlyOwner` modifier, what does the `_;` symbol mean?

A. End the function and return immediately

B. A placeholder where the body of the modified function runs

C. Increment a counter variable

D. Skip the rest of the modifier

Answer: B

`_;` is a placeholder in a modifier that tells Solidity: "run the function body here." Code before `_;` executes first (e.g., the `require` check), then the function body runs, then any code after `_;` in the modifier.

Question 25

What is the difference between `memory` and `storage` in Solidity function parameters?

A. `memory` is cheaper but disappears after the function ends; `storage` persists on-chain
B. `memory` stores data in RAM; `storage` stores data on disk
C. `memory` is for integers only; `storage` is for strings
D. `memory` costs more gas than `storage` for reads

Question 25

What is the difference between `memory` and `storage` in Solidity function parameters?

A. `memory` is cheaper but disappears after the function ends; `storage` persists on-chain

B. `memory` stores data in RAM; `storage` stores data on disk

C. `memory` is for integers only; `storage` is for strings

D. `memory` costs more gas than `storage` for reads

Answer: A

`memory` creates a temporary copy that exists only for the duration of the function call. `storage` refers to the persistent blockchain state. Function parameters like strings must be declared `memory` because they are temporary inputs.

Question 26

In the voting contract, why does `mapping(address => bool) public hasVoted` use `bool` as the value type?

A. Because booleans are the only type allowed in mappings

B. To record whether each address has cast a vote (true = voted, false = not voted)

C. To count how many times each address has voted

D. To store the vote choice (true = yes, false = no)

Question 26

In the voting contract, why does `mapping(address => bool) public hasVoted` use `bool` as the value type?

A. Because booleans are the only type allowed in mappings

B. To record whether each address has cast a vote (true = voted, false = not voted)

C. To count how many times each address has voted

D. To store the vote choice (true = yes, false = no)

Answer: B

The `bool` value acts as a flag: `hasVoted[voter] = false` by default (uninitialized mapping values are zero/false), and is set to `true` after the first vote. The `require(!hasVoted[msg.sender])` check then prevents double-voting.

Question 27

What happens if you call a function with `onlyOwner` modifier from an address that is NOT the owner?

A. The function runs but logs a warning event
B. The function runs with reduced privileges
C. The transaction reverts with "Not owner" error
D. The function is queued for later execution by the owner

Question 27

What happens if you call a function with `onlyOwner` modifier from an address that is NOT the owner?

A. The function runs but logs a warning event

B. The function runs with reduced privileges

C. The transaction reverts with "Not owner" error

D. The function is queued for later execution by the owner

Answer: C

The modifier checks `require(msg.sender == owner, "Not owner")` before running the function. If the caller is not the owner, `require` fails and the entire transaction reverts — no state changes occur.

Question 28

How does the `getWinner()` function in the voting contract determine the winning proposal?

A. It returns the proposal that was created first
B. It iterates all proposals and returns the one with the highest voteCount
C. It calls an oracle to determine the outcome
D. It returns the proposal with the most recent vote

Question 28

How does the `getWinner()` function in the voting contract determine the winning proposal?

A. It returns the proposal that was created first

B. It iterates all proposals and returns the one with the highest voteCount

C. It calls an oracle to determine the outcome

D. It returns the proposal with the most recent vote

Answer: B

The function uses a loop to iterate all proposals, tracking the index of the one with the highest `voteCount`. It then returns the winning proposal's data. This is an O(n) operation and would be expensive for very large proposal sets.

Question 29

After deploying a contract to Remix's local blockchain (JavaScript VM), which interface elements let you interact with it?

A. The compiler panel on the left
B. The deployed contract's expanded panel in the "Deploy & Run Transactions" tab
C. The terminal at the bottom only
D. A separate browser window that Remix opens automatically

Question 29

After deploying a contract to Remix's local blockchain (JavaScript VM), which interface elements let you interact with it?

A. The compiler panel on the left
B. The deployed contract's expanded panel in the "Deploy & Run Transactions" tab
C. The terminal at the bottom only
D. A separate browser window that Remix opens automatically

Answer: B
After clicking "Deploy", Remix shows the deployed contract at the bottom of the "Deploy & Run Transactions" tab. Blue buttons are read-only (free calls), orange buttons write state (send transactions with gas).

Question 30

In Remix, which compiler version selector setting is used to match `pragma solidity ^0.8.20;`?

A. Any version starting with 0.7.x
B. Version 0.8.20 or any later 0.8.x version
C. Version 0.8.20 exactly — no other version works
D. The latest 0.9.x version

Question 30

In Remix, which compiler version selector setting is used to match `pragma solidity ^0.8.20;`?

A. Any version starting with 0.7.x

B. Version 0.8.20 or any later 0.8.x version

C. Version 0.8.20 exactly — no other version works

D. The latest 0.9.x version

Answer: B

The caret ^ means "compatible with". `^0.8.20` accepts 0.8.20, 0.8.21, 0.8.25, etc., but not 0.9.0. In Remix, selecting any 0.8.x compiler at version 0.8.20 or higher satisfies this pragma.

Question 31

When deploying to a public testnet like Sepolia from Remix, what must you select in the "Environment" dropdown?

A. JavaScript VM (London)
B. Hardhat Provider
C. Injected Provider - MetaMask
D. Web3 Provider

Question 31

When deploying to a public testnet like Sepolia from Remix, what must you select in the "Environment" dropdown?

A. JavaScript VM (London)
B. Hardhat Provider
C. Injected Provider - MetaMask
D. Web3 Provider

Answer: C
"Injected Provider - MetaMask" connects Remix to your MetaMask wallet. Remix then uses your wallet to sign and broadcast the deployment transaction to whatever network MetaMask is connected to (e.g., Sepolia testnet).

Question 32

What is the gas cost difference between writing to a storage variable and performing a simple addition, roughly?

A. They cost the same: about 3 gas each
B. Addition costs $\sim$3 gas; a storage write costs $\sim$20,000 gas
C. Addition costs $\sim$1,000 gas; a storage write costs $\sim$2,000 gas
D. Storage writes are free if the variable was already initialized

Question 32

What is the gas cost difference between writing to a storage variable and performing a simple addition, roughly?

A. They cost the same: about 3 gas each

B. Addition costs ∼3 gas; a storage write costs ∼20,000 gas

C. Addition costs ∼1,000 gas; a storage write costs ∼2,000 gas

D. Storage writes are free if the variable was already initialized

Answer: B

EVM operation costs are highly asymmetric: an ADD opcode costs 3 gas, while SSTORE (writing a new storage slot) costs 20,000 gas. This is why minimizing storage writes is critical for gas-efficient contract design.

Question 33

Why can't the voting contract's `getWinner()` loop be used in a contract with thousands of proposals on mainnet?

A. Solidity does not support loops longer than 100 iterations
B. The loop would hit the block gas limit and the transaction would revert
C. MetaMask blocks transactions that use loops
D. Remix cannot compile contracts with unbounded loops

Question 33

Why can't the voting contract's `getWinner()` loop be used in a contract with thousands of proposals on mainnet?

A. Solidity does not support loops longer than 100 iterations

B. The loop would hit the block gas limit and the transaction would revert

C. MetaMask blocks transactions that use loops

D. Remix cannot compile contracts with unbounded loops

Answer: B

Each iteration of the loop costs gas. If the proposals array is very large, the total gas cost of a single call could exceed the block gas limit (∼30M gas on Ethereum), causing the transaction to revert. Production voting contracts track the winner incrementally to avoid this.

Question 34

Which step must a student do in Remix BEFORE clicking the “Deploy” button?

A. Configure MetaMask with a testnet
B. Successfully compile the contract (the compile button must show no errors)
C. Export the ABI to a JSON file
D. Set the gas price manually

Question 34

Which step must a student do in Remix BEFORE clicking the "Deploy" button?

A. Configure MetaMask with a testnet
B. Successfully compile the contract (the compile button must show no errors)
C. Export the ABI to a JSON file
D. Set the gas price manually

Answer: B
You must compile first. Remix compiles Solidity source code into bytecode. If there are compiler errors (shown in red), deployment is not possible. Only after a successful compile is the contract available in the "Deploy & Run Transactions" panel.

Question 35

What limitation does the simple voting contract described in the slides have compared to a production system?

A. It cannot emit events
B. It does not support token-weighted voting or vote delegation
C. It can only handle two proposals
D. It requires all voters to be whitelisted before deployment

Question 35

What limitation does the simple voting contract described in the slides have compared to a production system?

A. It cannot emit events

B. It does not support token-weighted voting or vote delegation

C. It can only handle two proposals

D. It requires all voters to be whitelisted before deployment

Answer: B

The simple voting contract gives equal weight to all voters (one address = one vote) and has no delegation mechanism. Production systems like Compound governance use token-weighted voting and allow token holders to delegate their voting power.

Question 36

In the ERC-20 token we built, what happens if Alice tries to call `transfer(bob, 1000)` but her `balanceOf` is only 500?

A. Alice is credited an overdraft and must repay later

B. The transaction reverts because `require(balanceOf[msg.sender] >= _amount)` fails

C. Only 500 tokens are transferred instead of 1000

D. The transfer succeeds but a debt mapping is updated

Question 36

In the ERC-20 token we built, what happens if Alice tries to call `transfer(bob, 1000)` but her `balanceOf` is only 500?

A. Alice is credited an overdraft and must repay later

B. The transaction reverts because `require(balanceOf[msg.sender] >= _amount)` fails

C. Only 500 tokens are transferred instead of 1000

D. The transfer succeeds but a debt mapping is updated

Answer: B

The `require` check enforces that the sender has sufficient balance. If it fails, `revert` is triggered: no tokens move, no state changes persist, and the caller receives the error message "Insufficient balance". Solidity has no overdraft concept.