

# L15: Solidity Fundamentals

## Module B: Ethereum & Smart Contracts

Blockchain & Cryptocurrency Course

December 2025

*[COMIC: Developer beaming with pride shows laptop screen with “contract HelloWorld { string public message = ‘Hello!’; }” to friends. One friend asks “So it stores one word... for how much?” Developer’s pride deflates as they mumble “...twenty thousand gas.”]*

## Frame the Concept:

- Your first smart contract feels magical—code that controls real value on a global network
- But Solidity has unique constraints: immutability, gas costs, and security implications

*Every Solidity developer’s journey begins with Hello World and ends with reentrancy nightmares*

By the end of this lesson, you will be able to:

- Describe Solidity's role as a smart contract language
- Declare and use fundamental data types (uint, address, bool, string, bytes)
- Write functions with appropriate visibility and state mutability modifiers
- Implement events for logging and monitoring
- Use mappings and arrays for data storage
- Apply inheritance and interfaces

**Building on L14:** Gas Mechanics

# The Problem: How do we write bug-free financial code?

## The Challenge

Smart contracts handle billions of dollars in value, yet bugs are irreversible once deployed. Unlike traditional software, there's no "update" button—a single logic error can drain funds permanently.

## Why It Matters

- Smart contract bugs are publicly exploitable by anyone monitoring the blockchain
- Historical losses: The DAO (\$60M, 2016), Parity multisig freeze (\$280M, 2017), Poly Network (\$611M, 2021)

## What We Need

- Risk management and mitigation
- Language features that prevent common vulnerability patterns (reentrancy, integer overflow, access control failures)

## The Cryptoeconomics Question

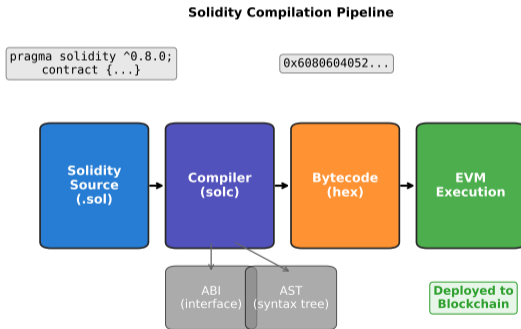
*Managing systemic and idiosyncratic risks*

*Today's lesson: How Solidity Fundamentals addresses this challenge*

# How Does Solidity Enable Smart Contracts?

**Solidity is a statically-typed (variable types fixed at compile time, catching type errors early), contract-oriented programming language:**

- Created specifically for Ethereum smart contracts
- Syntax similar to JavaScript/C++, compiles to EVM bytecode
- Current stable version: 0.8.x (as of 2025)



→ Solidity's static typing catches errors at compile time, before deployment

# What Is the Basic Contract Structure?

## Every Solidity file starts with a version pragma:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HelloWorld {
    string public message;

    constructor(string memory initialMessage) {
        message = initialMessage;
    }

    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

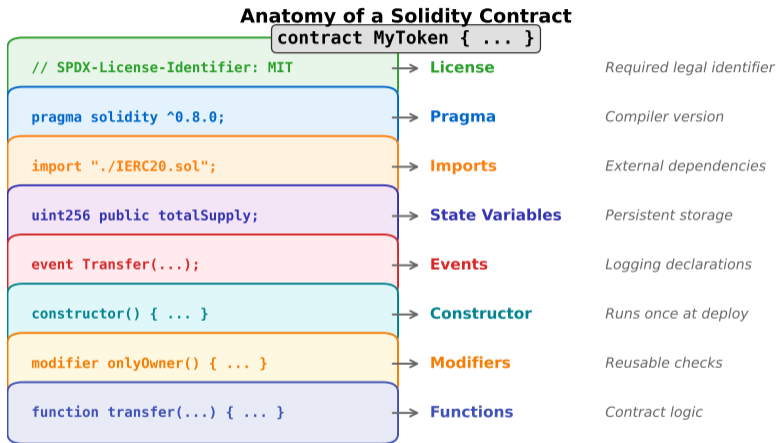
    function getMessage() public view returns (string memory) {
        return message;
    }
}
```

**Key Elements:** License identifier, pragma, contract declaration, state variables, constructor, functions

*Study this code pattern carefully*

# How Is a Solidity Contract Organized?

Structure of a Solidity contract from top to bottom:



**Order matters: License > Pragma > Imports > State > Events > Constructor > Modifiers > Functions**

# What Are SPDX Licenses and Version Pragmas?

**Required at top of every Solidity file:**

**Common Licenses:**

- **MIT:** Permissive open-source license
- **GPL-3.0:** Copyleft license (derivatives must be open-source)
- **UNLICENSED:** Proprietary code

**Version Pragma:**

- `pragma solidity ^0.8.0;` - Compatible with 0.8.0 to 0.8.x
- `pragma solidity >=0.8.0 <0.9.0;` - Range specification
- `pragma solidity 0.8.20;` - Exact version

*Key point: Required at top of every Solidity file*

# What Data Types Does Solidity Support?

Solidity has two categories of data types:

## Solidity Data Types

### Value Types

**uint/int** uint256, int128...

**address** 20-byte Ethereum addr

**bool** true/false

**bytes1-32** Fixed-size bytes

**enum** User-defined states

Copied when assigned  
Stored directly in memory/storage

### Reference Types

**arrays** uint[], string[]

**mappings** mapping(K => V)

**structs** Custom data types

**string** Dynamic UTF-8

**bytes** Dynamic byte array

Passed by reference  
Require data location (storage/memory/calldata)

*What Data Types Does Solidity Support? – visual summary*

# What Are Integer and Boolean Value Types?

## Integers:

- `uint` (unsigned): 0 to  $2^{256} - 1$  (alias for `uint256`)
- `int` (signed):  $-2^{255}$  to  $2^{255} - 1$  (alias for `int256`)
- Sized variants: `uint8`, `uint16`, ..., `uint256`

## Booleans:

- `bool`: `true` or `false`
- Operators: `!` (not), `&&` (and), `||` (or)

```
contract Types {
    uint256 public largeNumber = 1000000000000000000; // 1e18
    uint8 public smallNumber = 255; // Max value for uint8
    int256 public signedNumber = -42;
    bool public isActive = true;
}
```

→ Problem: How do we write bug-free financial code? — Value Types: Integers and Booleans Static typing prevents overflow bugs: Solidity 0.8+ reverts on overflow instead of wrapping

# What Is the Address Type?

## Address type holds 20-byte Ethereum addresses:

- address: Basic address type
- address payable: Can receive Ether via transfer() or send()

## Address Members:

- <address>.balance: Returns Ether balance (in Wei)
- <address payable>.transfer(uint amount): Send Ether, reverts on failure

```
contract AddressExample {
    address public owner;
    address payable public recipient;

    constructor() {
        owner = msg.sender; // Address of contract deployer
        recipient = payable(msg.sender); // Convert to payable
    }

    function checkBalance() public view returns (uint) {
        return owner.balance; // Balance in Wei
    }
}
```

*Study this code pattern carefully*

# How Do Arrays Work in Solidity?

## Fixed-Size Arrays:

```
uint[5] public fixedArray; // Array of 5 uints
```

## Dynamic Arrays:

```
uint[] public dynamicArray;  
string[] public names;  
  
function addElement(uint value) public {  
    dynamicArray.push(value); // Append to array  
}  
  
function getLength() public view returns (uint) {  
    return dynamicArray.length;  
}  
  
function removeLastElement() public {  
    dynamicArray.pop(); // Remove last element  
}
```

*Study this code pattern carefully*

# How Do Mappings Work in Solidity?

## Key-value storage (like hash tables):

```
contract MappingExample {
    // Mapping from address to balance
    mapping(address => uint256) public balances;

    // Nested mapping (address to address to allowance)
    mapping(address => mapping(address => uint256)) public allowances;

    function updateBalance(address account, uint256 amount) public {
        balances[account] = amount;
    }

    function getBalance(address account) public view returns (uint256) {
        return balances[account]; // Returns 0 if key doesn't exist
    }
}
```

**Key Properties:** All keys exist with default value, cannot iterate, only in storage

*Study this code pattern carefully*

# How Do Visibility Modifiers Control Access?

Four visibility levels determine who can call a function:

**Solidity Function Visibility Access**

	External (EO)	Same	Derived
public	Y	Y	Y
external	Y	N	N
internal	N	Y	Y
private	N	Y	N

Tip: Use external for gas efficiency when only called from outside

How Do Visibility Modifiers Control Access? – visual summary

## Recall Our Problem

*How do we write bug-free financial code?*

## What We've Learned So Far

- Static typing catches type mismatches at compile time (before deployment)
- Visibility modifiers (public/private/internal/external) control who can call functions
- Compile-time checks + access control = fewer bugs in immutable financial code

## Still to Address

- How do error handling mechanisms (require/assert/revert) prevent runtime bugs?
- How would YOU design a language for bug-free financial code?

## Think About

- Based on what you've seen, how would *you* solve this problem?
- What trade-offs do you expect?

*Pause and reflect: How does what we've learned so far address "How do we write bug-free financial code?"?*

# What Are State Mutability Modifiers?

## Three state mutability levels:

- 1 **view**: Reads state but doesn't modify it (no gas when called off-chain)
- 2 **pure**: Doesn't read or modify state
- 3 **(none)**: Can read and modify state (always costs gas)

```
contract Mutability {
    uint public value = 10;

    function getValue() public view returns (uint) {
        return value; // Reads state (view)
    }

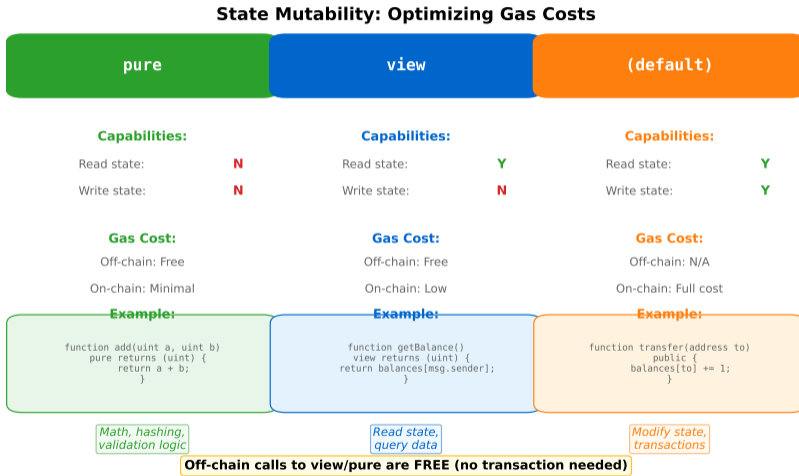
    function add(uint a, uint b) public pure returns (uint) {
        return a + b; // No state access (pure)
    }

    function setValue(uint newValue) public {
        value = newValue; // Modifies state (no modifier)
    }
}
```

→ Problem: How do we write bug-free financial code? — Functions: State Mutability State mutability (view/pure) prevents accidental state changes in read-only functions

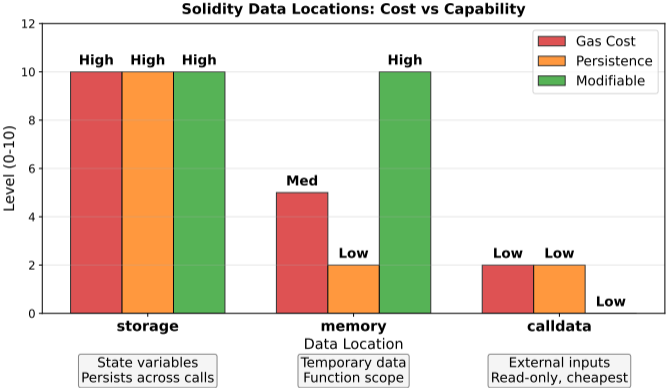
# How Do State Mutability Levels Compare?

Choosing the right mutability saves gas:



# Where Is Data Stored During Execution?

Reference types require explicit data location:



Where Is Data Stored During Execution? – visual summary

# How Do Events Enable Logging?

## Events enable logging for off-chain monitoring:

```
contract EventExample {
    event Transfer(address indexed from, address indexed to, uint256 amount);
    event Approval(address indexed owner, address indexed spender, uint256 amount);

    mapping(address => uint256) public balances;

    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
        emit Transfer(msg.sender, to, amount); // Emit event
    }
}
```

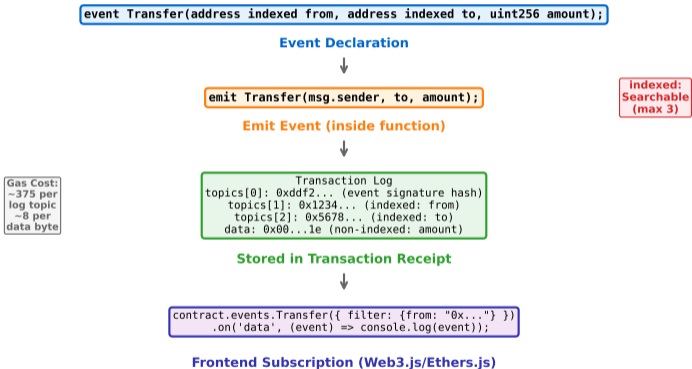
**Indexed Parameters:** Up to 3 parameters can be indexed (searchable in event logs, max 3 per event) for efficient filtering

*Study this code pattern carefully*

# How Do Events Flow from Contract to Frontend?

How events connect on-chain activity to off-chain applications:

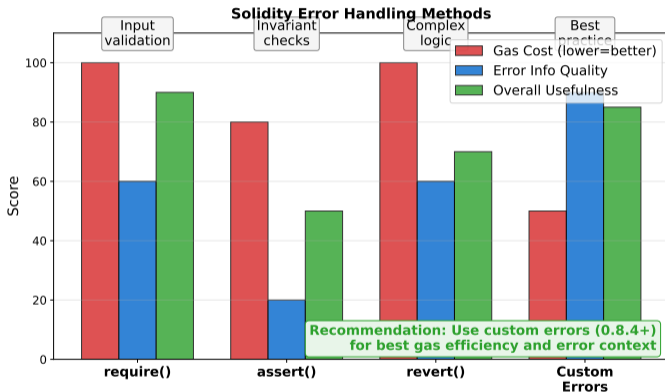
## Events & Logging: From Contract to Frontend



How Do Events Flow from Contract to Frontend? – visual summary

# How Does Solidity Handle Errors?

Solidity provides multiple error handling mechanisms:



*How Does Solidity Handle Errors? – visual summary*

# How Do require, assert, and revert Work?

- 1 **require(condition, message):** Validate inputs/conditions, refunds gas
- 2 **assert(condition):** Check invariants (should never fail)
- 3 **revert(message):** Unconditional revert with message

```
function transfer(address to, uint amount) public {
    require(to != address(0), "Cannot transfer to zero address");
    require(balances[msg.sender] >= amount, "Insufficient balance");

    balances[msg.sender] -= amount;
    balances[to] += amount;

    assert(balances[msg.sender] + balances[to] == totalSupply); // Invariant
}
```

→ Problem: How do we write bug-free financial code? — Error Handling: *require()* validates inputs, *assert()* checks invariants, *revert()* with custom errors saves gas

# How Do Custom Errors Save Gas?

## More gas-efficient than string error messages:

```
contract CustomErrors {
    error InsufficientBalance(uint requested, uint available);
    error Unauthorized(address caller);

    address public owner;

    function withdraw(uint amount) public {
        if (msg.sender != owner) {
            revert Unauthorized(msg.sender);
        }
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(amount, balances[msg.sender]);
        }
        // ... transfer logic
    }
}
```

**Benefits:** Lower gas cost, typed parameters, better error context

*Study this code pattern carefully*

# What Are Modifiers and How Do They Work?

## Reusable code for function preconditions:

```
contract ModifierExample {
    address public owner;
    bool public paused = false;

    constructor() { owner = msg.sender; }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _; // Placeholder for function body
    }

    modifier whenNotPaused() {
        require(!paused, "Contract is paused");
        _;
    }

    function pause() public onlyOwner { paused = true; }
    function unpause() public onlyOwner { paused = false; }

    function criticalFunction() public onlyOwner whenNotPaused {
        // Only owner can call, and only when not paused
    }
}
```

*Study this code pattern carefully*

# How Does Modifier Execution Flow Work?

## How modifiers wrap and control function execution:

### How Modifiers Wrap Function Execution

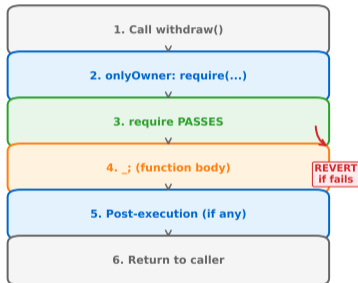
#### Code Definition:

```
modifier onlyOwner() {  
    require(msg.sender == owner);  
    _; // function body here  
    // optional: post-execution  
}
```

```
function withdraw() public onlyOwner {  
    // transfer funds  
}
```

Multiple modifiers:  
func() mod1 mod2  
= mod1 wraps mod2 wraps body

#### Execution Flow:



The underscore ( `_` ) is replaced by the function body at execution time

*How Does Modifier Execution Flow Work? – visual summary*

# How Does Inheritance Work in Solidity?

## Solidity supports multiple inheritance:

```
contract Ownable {
    address public owner;
    constructor() { owner = msg.sender; }
    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }
}

contract Pausable is Ownable {
    bool public paused;
    function pause() public onlyOwner { paused = true; }
}

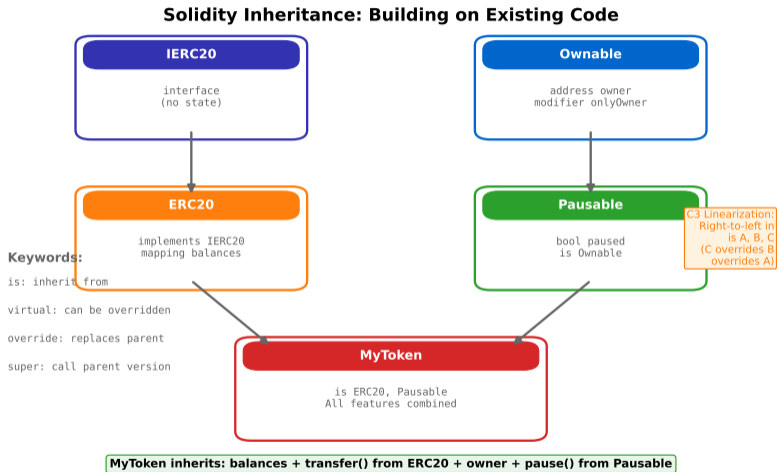
contract MyContract is Pausable {
    function doSomething() public onlyOwner {
        // Inherits owner, onlyOwner, paused, pause()
    }
}
```

**Key:** is for inheritance, virtual/override for polymorphism

*Study this code pattern carefully*

# How Do Inheritance Hierarchies Build Contracts?

Building contracts by combining existing code:



# What Are Interfaces and Why Use Them?

## Define contract structure without implementation:

```
interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
}

contract MyToken is IERC20 {
    mapping(address => uint256) private _balances;
    uint256 private _totalSupply;

    function totalSupply() public view override returns (uint256) {
        return _totalSupply;
    }
    // ... implement other functions
}
```

*Study this code pattern carefully*

## The Original Problem

*How do we write bug-free financial code?*

## How Solidity Fundamentals Solves It

- Static typing catches type mismatches at compile time (e.g., can't send ETH to non-payable address)
- Visibility modifiers enforce access control patterns (onlyOwner prevents unauthorized actions)
- State mutability (view/pure) prevents accidental state changes in read-only functions

## Remaining Limitations

- No formal verification by default—compiler doesn't prove absence of logic bugs
- Tooling maturity lags traditional languages (limited IDE support, debuggers)

## Open Questions

- Will formal verification tools (Certora, K Framework) become standard for high-value contracts?
- Risk: Black swan events (0-day compiler bugs), cascading failures (reentrancy across protocols)

*Solidity Fundamentals partially solves "we write bug-free financial code" but introduces new trade-offs*

## Incentive Structure

- Managing systemic and idiosyncratic risks
- Risk-adjusted returns, insurance mechanisms
- Users bear risk for higher returns

## Economic Security

- Attack cost must exceed potential gain
- Honest behavior = Nash equilibrium

*Cryptoeconomic security: Honest behavior must be the Nash equilibrium*

## Key Economic Question

### Who Pays, Who Earns?

Users bear risk for higher returns

## Design Principle

Attack Cost  $>$  Potential Gain

## Alternatives Considered

- 1 **Chosen Design:** Risk parameters, circuit breakers
- 2 **Alternative:** Traditional risk management approaches

## Trade-offs Made

- Every design optimizes some properties
- ... at the expense of others

## Design Questions

- What would YOU change?
- What's optimized? What's sacrificed?
- Are there other approaches?

## Key Insight

### No Perfect Solution

All blockchain designs involve trade-offs between decentralization, security, and scalability.

*Every design is a trade-off. Understanding alternatives reveals the "why" behind choices.*

## Failure Modes

## Critical Failure Mode

- Black swan events, cascading failures
- Economic incentives misaligned

## Root Cause

- Assumption violated
- Incentive structure broken
- External shock

## Historical Context

- Multiple real-world failures documented
- Patterns repeating across protocols

## Early Warning Signs

- ! Unusual economic behavior
- ! Incentive misalignment
- ! Centralization drift

*Prediction: What could cause this to fail? How would you detect it early?*

*[COMIC: Developer has written 200 lines of Solidity. Hits compile. Screen fills with red: "TypeError: Member 'balances' not found or not visible." "DeclarationError: Identifier already declared." "ParserError: Expected ';' but got '}'." Developer's coffee cup is empty. Clock shows 3 AM.]*

## Key Insight:

- Solidity's strict compiler is your friend—it catches bugs before deployment, not after
- Every compiler error you fix is a potential \$1M exploit you avoided

*Red errors at compile time beat red transactions on mainnet*

- 1 **Solidity Basics:** Statically-typed language compiling to EVM bytecode
- 2 **Data Types:** Value types (uint, address, bool) vs reference types (arrays, mappings)
- 3 **Visibility:** public, external, internal, private determine access
- 4 **State Mutability:** view (read-only), pure (no state), default (modify)
- 5 **Data Locations:** storage (persistent), memory (temp), calldata (cheapest)
- 6 **Error Handling:** Use custom errors (0.8.4+) for gas efficiency

**Next Lesson:** L16 – Lab: Contract Interaction

*Key point: Solidity Basics*

- ① Why is `string` more expensive than `bytes32` for storing short text?
- ② When should you use `external` vs `public` for function visibility?
- ③ Why can't you iterate over a mapping's keys in Solidity?
- ④ What are the tradeoffs between events vs state variables for historical records?
- ⑤ How does the `indexed` keyword in events affect gas costs and queryability?

*Key point: Discussion Questions*

**Continued**

## Coming up next (hands-on lab):

- Introduction to Remix IDE
- Deploying SimpleStorage contract
- Interacting with deployed contracts
- Using MetaMask with test networks
- Deploying to Sepolia testnet

## Preparation:

- Install MetaMask browser extension
- Create Ethereum account and save recovery phrase
- Get Sepolia testnet ETH from faucet

*Key point: Coming up next (hands-on lab)*

## Quiz: Questions 1-5

**Q1. What is the maximum value that can be stored in a uint8 variable?**

- A) 128   B) 255   C) 256   D) 65535

**Answer: B** – uint8 is an 8-bit unsigned integer with range 0 to  $2^8 - 1 = 255$ .

**Q2. Which data type should be used for an Ethereum address that needs to receive Ether?**

- A) address   B) address payable   C) bytes20   D) uint160

**Answer: B** – address payable can receive Ether via transfer() or send().

**Q3. What does the “view” state mutability modifier indicate?**

- A) Function can modify state   B) Function cannot read or modify state  
C) Function reads state but cannot modify it   D) Function is only visible internally

**Answer: C** – view functions read state without modifying it, consuming no gas when called off-chain.

**Q4. Which visibility modifier allows a function to be called ONLY from within the same contract?**

- A) public   B) external   C) internal   D) private

**Answer: D** – private functions can only be called from within the same contract, not from derived contracts.

**Q5. What is the default value for an uninitialized mapping entry?**

- A) null   B) undefined   C) Zero/default value of the value type   D) Throws an error

**Answer: C** – All keys in a mapping exist with their type’s default value (0 for uint, false for bool, etc.).

## Quiz: Questions 6-10

**Q6. Which error handling method should be used to check invariants that should never fail?**

- A) require() B) assert() C) revert() D) throw

**Answer: B** – assert() is for internal errors and invariant checking; it should never fail in correct code.

**Q7. What is the purpose of the constructor in a Solidity contract?**

- A) To destroy the contract B) To initialize state variables when contract is deployed  
C) To create new instances of the contract D) To define contract visibility

**Answer: B** – The constructor runs once during deployment to initialize state variables.

**Q8. Which data location is the cheapest for function parameters in terms of gas?**

- A) storage B) memory C) calldata D) stack

**Answer: C** – calldata is read-only and cheapest because data is not copied into memory.

**Q9. How many parameters in an event can be marked as “indexed” for efficient filtering?**

- A) 1 B) 2 C) 3 D) Unlimited

**Answer: C** – Up to 3 parameters can be indexed in an event for efficient off-chain filtering.

**Q10. What does the underscore (\_) represent in a modifier?**

- A) A comment marker B) A private variable prefix  
C) Placeholder for the function body D) An unused parameter

**Answer: C** – The underscore is replaced by the modified function's body during execution.

**Q11. Which keyword is used to inherit from another contract?**

- A) extends
- B) implements
- C) is
- D) inherits

**Answer: C** – Solidity uses “is” for inheritance (e.g., contract MyToken is ERC20).

**Q12. What is the difference between “pure” and “view” functions?**

- A) pure is faster than view
- B) pure cannot read state, view can read but not modify
- C) pure is for external calls only
- D) No difference, they are aliases

**Answer: B** – pure functions access no state variables; view functions read but don't modify state.

**Q13. Which of the following is TRUE about mappings in Solidity?**

- A) They can be iterated like arrays
- B) They are stored in memory
- C) All possible keys exist with default values
- D) They have a length property

**Answer: C** – Mappings have no concept of undefined keys; all keys map to their type's default value.

**Q14. What is the recommended error handling method in Solidity 0.8.4+ for gas efficiency?**

- A) String-based require()
- B) Custom errors with revert
- C) assert() for all checks
- D) try-catch blocks

**Answer: B** – Custom errors are more gas-efficient than string error messages in require().

**Q15. Which visibility modifier allows external contracts to call a function but NOT internal calls?**

- A) public
- B) external
- C) internal
- D) private

**Answer: B** – external functions can only be called from outside the contract, saving gas for calldata.

Quiz

## Quiz: Questions 16-20

**Q16. What is the size of the address type in Solidity?**

- A) 16 bytes   B) 20 bytes   C) 32 bytes   D) 40 bytes

**Answer: B** – Ethereum addresses are 20 bytes (160 bits) long.

**Q17. Which method is used to add an element to a dynamic array?**

- A) append()   B) add()   C) push()   D) insert()

**Answer: C** – push() appends an element to the end of a dynamic array.

**Q18. What happens when a function marked “view” attempts to modify state?**

- A) Compilation warning   B) Compilation error  
C) Runtime error   D) State modification is ignored

**Answer: B** – Solidity compiler will reject code where view functions attempt state modifications.

**Q19. Which of the following can be declared in an interface?**

- A) State variables   B) Function implementations  
C) Function signatures only   D) Constructors

**Answer: C** – Interfaces can only declare function signatures, events, and errors, not implementations or state.

**Q20. What is the purpose of the “memory” keyword for reference types?**

- A) Optimize for low memory usage   B) Indicate temporary storage during function execution  
C) Enable garbage collection   D) Allow dynamic resizing

**Answer: B** – memory indicates data is stored temporarily during function execution, not persisted to blockchain.