Solidity Basics

Notes + Example Code: EVM, Variables, Functions, Arrays, Globals

Introduction

Solidity is the main language used to write Ethereum smart contracts. This handout summarizes key concepts and provides short example code for each topic.

1. What is Solidity?

Solidity is an **object-oriented**, **high-level** language for implementing **smart contracts**. It is designed to target the **Ethereum Virtual Machine (EVM)**.

- Statically typed
- Supports inheritance, libraries, and user-defined types

Exam Check

On which virtual machine does Solidity run? EVM

Example: Minimal contract file (pragma + contract)

```
pragma solidity ^0.8.19;

contract HelloWorld {
    // empty contract
}
```

2. Variables in Solidity (Local / State / Global)

Local variables

- Declared inside a function
- Not stored on-chain

State variables

- Declared outside functions
- Stored on-chain (persistent)

Global variables

- Injected by EVM at runtime
- Examples: `msg.sender`, `block.timestamp`, `block.coinbase`

Scope rule: Scope depends on where a variable is declared (not its value).

Example: Local + State + Global variables

```
pragma solidity ^0.8.19;

contract VarDemo {
    // State variable (stored on-chain)
    uint public counter = 0;

    function inc() public {
        // Local variable (temporary)
        uint beforeVal = counter;

        // Global variables from EVM
        address caller = msg.sender;
        uint time = block.timestamp;

        counter = beforeVal + 1;

        // (caller/time are just examples; not stored unless assigned to state)
    }
}
```

Exam Check

State variables are: declared outside a function and stored on the blockchain.

3. Common Types + Default Values

- **uint** = unsigned integer (alias for `uint256`)
- **int** = signed integer (alias for `int256`)
- **address** = Ethereum address type
- **bool** = boolean

Ranges:

- `uint8`: 0 to $2^8 - 1$
- `uint256`: 0 to $2^{256} - 1$

Defaults (unassigned):

- `bool` → false
- `uint` → 0
- `int` → 0
- `address` → `0x0000000000000000000000000000000000000000`

Example: Types + defaults

```
pragma solidity ^0.8.19;

contract TypesDemo {
    uint8 public u8 = 10;
    uint public u = 1230;      // uint == uint256
    int public i = -123;

    address public addr =
        0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

    bool public b1 = false;

    // Defaults if not assigned:
    bool public b2;       // false
    uint public uDefault; // 0
    int public iDefault;  // 0
    address public aDefault; // 0x000...000
}
```

Quick MCQ Answers

- **uint stands for:** Unsigned Integer
- **Range of uint8:** 0 to $2^8 - 1$
- **Default bool:** false
- **Default address:** `0x0000000000000000000000000000000000000000`

4. Functions + Keywords (public, view)

public

- Callable internally and externally

view

- Cannot change contract state
- Reads only

Example: public + view (set/get pattern)

```
pragma solidity ^0.8.19;

contract SimpleStore {
    uint public num;

    function set(uint _num) public {
        num = _num; // changes state
    }

    function get() public view returns (uint) {
        return num; // reads state only
    }
}
```

Exam Check

- **public:** callable inside and outside the contract
- **view:** cannot modify state

5. Control Flow (If/Else, Loops, break/continue)

Solidity supports standard control flow such as **if/else** and **for loops**. Use loops carefully because large loops can be expensive in gas.

Example: if/else + loop + break/continue

```
pragma solidity ^0.8.19;

contract Conditions {
    function foo(uint x) public pure returns (uint) {
        if (x < 10) return 0;
        else if (x < 20) return 1;
        else return 2;
    }

    function loop() public pure returns (uint sum) {
        for (uint i = 0; i < 10; i++) {
            if (i == 3) continue; // skip i=3
            if (i == 5) break;    // stop at i=5
            sum += i;
        }
    }
}
```

6. Arrays, Strings, Storage vs Memory

Arrays can be fixed-size (`uint[10]`) or dynamic (`uint[]`).
storage vs memory

- **storage** = persistent state on-chain
- **memory** = temporary during execution

Common operations: `push`, `pop`, `length`, `delete`.

Example: Arrays + memory returns + common operations

```
pragma solidity ^0.8.19;

contract ArrayDemo {
    string public greet = "Hello World!";

    uint[] public arr;          // dynamic (storage)
    uint[] public arr2 = [1,2,3];
    uint[10] public fixedArr;   // fixed size

    function get(uint i) public view returns (uint) {
        return arr[i];
    }

    // Return an array in memory
    function getArr(uint[] memory _arr)
        public
        pure
        returns (uint[] memory)
    {
        return _arr;
    }

    function doStuff(uint x) public {
        arr.push(x);      // add
        arr.pop();        // remove last
        uint len = arr.length;
        delete arr2[1];   // resets arr2[1] to 0 (length unchanged)

        uint;
        string memory hi = "hi";

        // silence warnings
        a[0] = len;
        hi = hi;
    }

    function letterC() public pure returns (string memory) {
        return "C";
    }
}
```

Array MCQ Answers

- Correct return signature:
  `function getArr(uint[] memory _arr) public view returns (uint[] memory)`
- Length: `arr.length`
- Add: `arr.push(i)`

7. Global Variables (DYOR / Common Ones)

`msg.sender` = address of the caller

`block.coinbase` = address of the miner/validator who produced the block

Example: msg.sender and block.coinbase

```
pragma solidity ^0.8.19;

contract GlobalsDemo {
    function whoCalled() public view returns (address) {
        return msg.sender;
    }

    function blockProducer() public view returns (address) {
        return block.coinbase;
    }

    function timeNow() public view returns (uint) {
        return block.timestamp;
    }
}
```

Final Checklist (What to remember)

- Solidity runs on the **EVM**
- 3 variable categories: **local, state, global**
- **public** = internal + external access
- **view** = cannot change state
- Arrays: `push`, `pop`, `length`, `delete`
- Globals: `msg.sender`, `block.timestamp`, `block.coinbase`

Tip: Keep real funds separate from development wallets. Never commit private keys to GitHub.