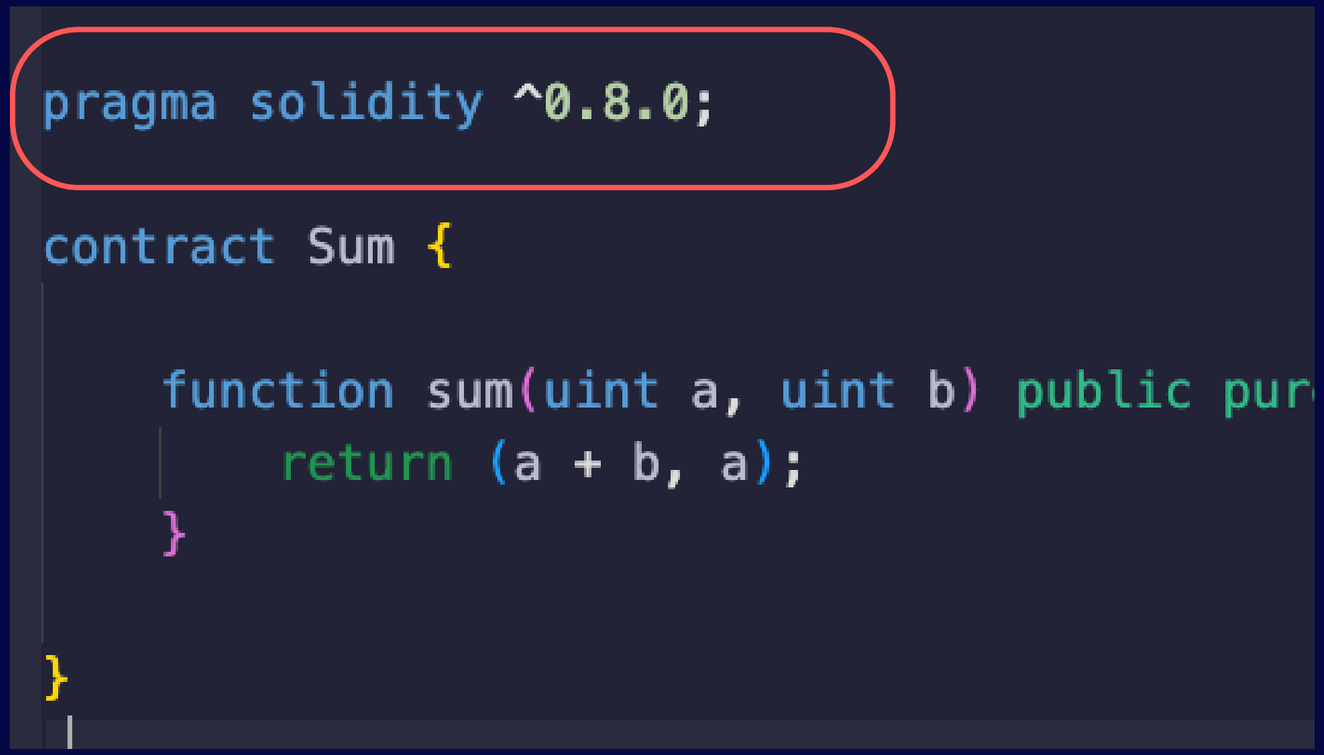

SOLIDITY

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Solidity version

- The version statement at the top of a Solidity contract specifies the version of the Solidity compiler that should be used to compile the contract



Solidity version

- The version statement at the top of a Solidity contract specifies the version of the Solidity compiler that should be used to compile the contract

```
pragma solidity ^0.8.0;

contract Sum {

    function sum(uint a, uint b) public pur
        return (a + b, a);
    }

}
```

version 0.8.0 or higher, but not 0.9.0 or above

Solidity version

- The version statement at the top of a Solidity contract specifies the version of the Solidity compiler that should be used to compile the contract

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Sum {

    function sum(uint a, uint b) public pure retu
        return (a + b, a);
    }

}
```

Any version betweek 0.7.0 to 0.9.0

Solidity version

- The version statement at the top of a Solidity contract specifies the version of the Solidity compiler that should be used to compile the contract

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Sum {

    function sum(uint a, uint b) public pure retu
        return (a + b, a);
    }

}
```

Any version betweek 0.7.0 to 0.9.0

Solidity version

- The version statement at the top of a Solidity contract specifies the version of the Solidity compiler that should be used to compile the contract

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {

    function sum(uint a, uint b) public pure
        return (a + b, a);
    }

}
```

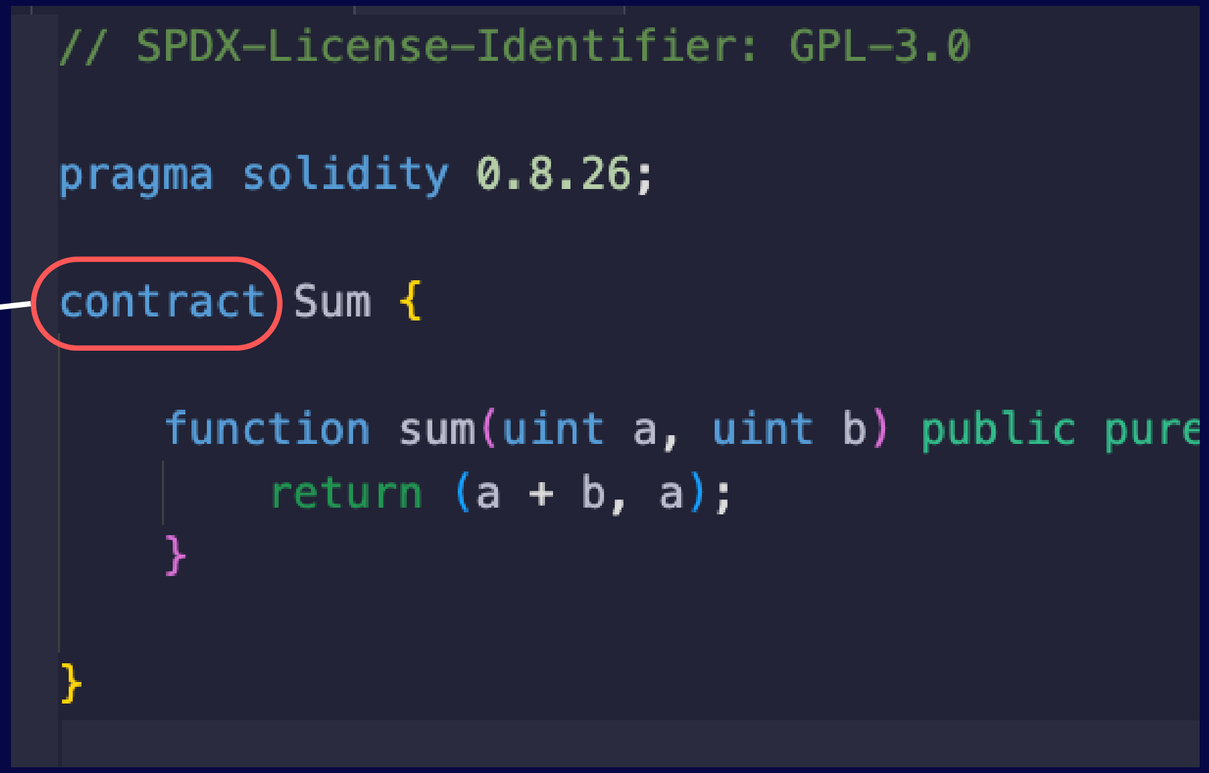
Exact version

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Contracts

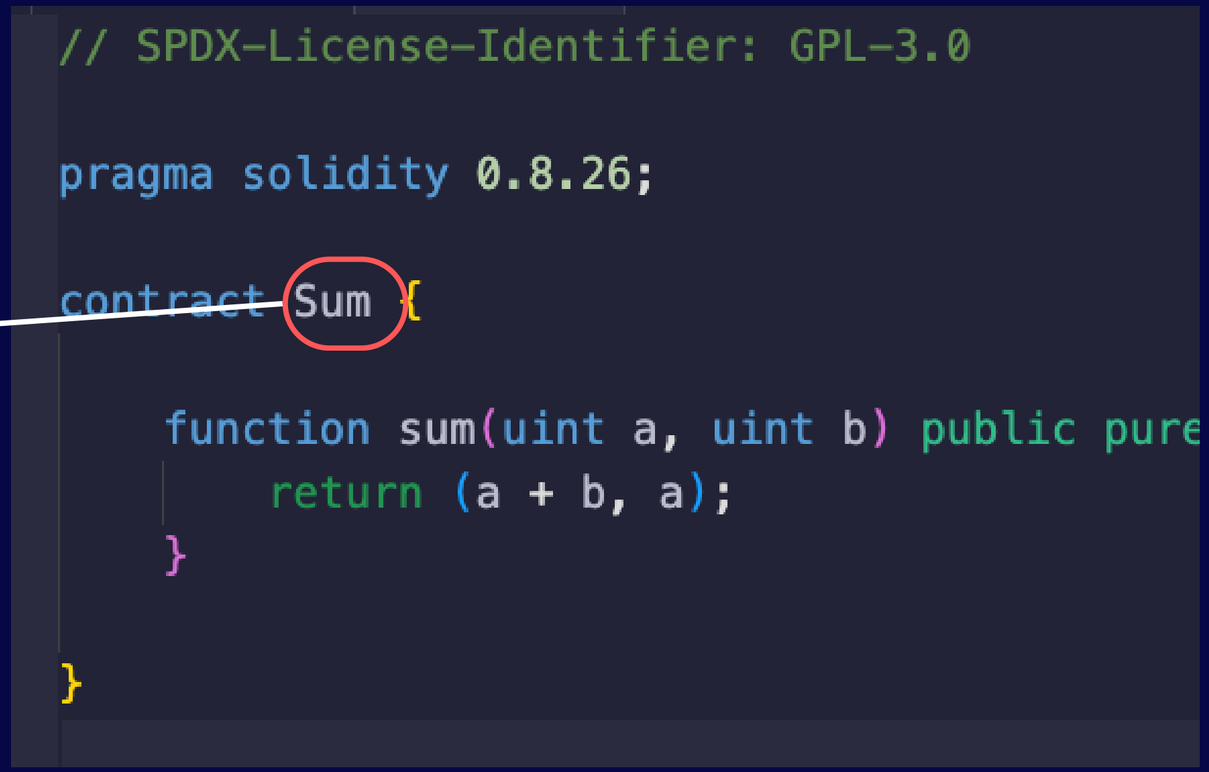
- Declaring the contract



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {

    function sum(uint a, uint b) public pure
        return (a + b, a);
    }

}
```

Contracts

- **Name of the contract**



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {

    function sum(uint a, uint b) public pure
        return (a + b, a);
    }

}
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Variables

In Solidity, variables are used to store data within a contract. They can hold different types of values and are fundamental to writing smart contracts.

Variables

In Solidity, variables are used to store data within a contract. They can hold different types of values and are fundamental to writing smart contracts.

For example - For a Decentralized Uber app,
You would store things like

1. Rides
2. Users
3. Locations
4. Paymments

in state

Variables

1. **Unsigned Numbers**
    a. **uint8 - Small numbers**
    b. **uint16 - 16 bit numbers**
    c. **uint256 - 256 bit number (uint)**

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {
    uint8 gender;
    uint16 calls;
    uint bigNumber = 100000;


}

```

Variables

1. **Unsigned Numbers**
    a. **uint8 - Small numbers**
    b. **uint16 - 16 bit numbers**
    c. **uint256 - 256 bit number (uint)**
2. **Signed numbers**
    a. **int**
    b. **int32**

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {
    uint8 gender;
    uint16 calls;
    uint bigNumber = 100000;
    int y = -5000000;
    int32 x = -100;



}
```

Variables

1. **Unsigned Numbers**
    a. **uint8 - Small numbers**
    b. **uint16 - 16 bit numbers**
    c. **uint256 - 256 bit number (uint)**
2. **Signed numbers**
    a. **int**
    b. **int32**
3. **Booleans**

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {
    uint8 gender;
    uint16 calls;
    uint bigNumber = 100000;
    int y = -5000000;
    int32 x = -100;

    bool isActive = true;


}
```

Variables

1. **Unsigned Numbers**
    a. **uint8 - Small numbers**
    b. **uint16 - 16 bit numbers**
    c. **uint256 - 256 bit number (uint)**
2. **Signed numbers**
    a. **int**
    b. **int32**
3. **Booleans**
4. **Addresses**

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {
    uint8 gender;
    uint16 calls;
    uint bigNumber = 100000;
    int y = -5000000;
    int32 x = -100;

    bool isActive = true;
    address owner = 0x8fFfF8B3F309bAb3D8F2909eE1DcB5A06d3178D1;


}
```

Variables

1. **Unsigned Numbers**
    a. **uint8 - Small numbers**
    b. **uint16 - 16 bit numbers**
    c. **uint256 - 256 bit number (uint)**
2. **Signed numbers**
    a. **int**
    b. **int32**
3. **Booleans**
4. **Addresses**
5. **Strings**

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.26;

contract Sum {
    uint8 gender;
    uint16 calls;
    uint bigNumber = 100000;
    int y = -5000000;
    int32 x = -100;

    bool isActive = true;
    address owner = 0x587EFaEe4f308aB2795ca35A27Dff8c1dfAF9e3f;
    string name = "Solidity Contract";


}
```

Constructor

Constructor

A constructor in Solidity is a special function that is executed only once during the deployment of the contract. Its primary purpose is to initialize the contract's state variables and set up any required logic when the contract is deployed to the Ethereum blockchain.

Constructor

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.0;
3
4   contract Calculator {
5
6       uint256 currentValue;
7
8       constructor(uint256 _initialValue) {    infinite gas 12400 gas
9           currentValue = _initialValue; // Set the initial value
10      }
11
12  }
13
```

Constructor

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

contract Calculator {

    uint256 currentValue;

    constructor(uint256 _initialValue) {    infinite gas 12400 gas
        currentValue = _initialValue; // Set the initial value
    }

}

```

_ to avoid names from conflicting

Functions

Functions

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

contract Calculator {

    uint256 currentValue;

    constructor(uint256 _initialValue) {  infinite gas 69400 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {  infinite gas
        currentValue += value;
    }

}
```

Functions

Name of the function

Functions

Type of the argument

Functions

Name of the argument

Functions

Function body

Functions

Visibility

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

contract Calculator {

    uint256 currentValue;

    constructor(uint256 _initialValue) {  infinite gas 69400 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {  infinite gas
        currentValue += value;
    }

}
```

Functions

Visibility

public,
private,external,
internal

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

contract Calculator {

    uint256 currentValue;

    constructor(uint256 _initialValue) {  infinite gas 69400 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {  infinite gas
        currentValue += value;
    }

}
```

Functions

Visibility Comparison Table

Visibility	Same Contract	Derived Contracts	Other Contracts	External Users
`public`	✅	✅	✅	✅
`external`	❌ (directly)	❌	✅	✅
`internal`	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Assignment

Try finishing the Calculator contract (add, multiply, divide, subtract, getValue)

Solution

```sol
contract Calculator {
    uint256 public currentValue;

    constructor(uint256 _initialValue) {   infinite gas 177200 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {   infinite gas
        currentValue += value;
    }

    function subtract(uint256 value) public {   infinite gas
        currentValue -= value;
    }

    function multiply(uint256 value) public {   infinite gas
        currentValue *= value;
    }

    function divide(uint256 value) public {   infinite gas
        currentValue /= value;
    }

    function getCurrentValue() public view returns (uint256) {   2410 gas
        return currentValue;
    }
}
```

View

In Solidity, the view keyword indicates that the function does not modify the state of the blockchain.

It is a type of function modifier that tells the Ethereum Virtual Machine (EVM) that the function is read-only and will not alter any state variables or perform any operations that would require a transaction.

```
contract Calculator {
    uint256 public currentValue;

    constructor(uint256 _initialValue) {   infinite gas 177200 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {   infinite gas
        currentValue += value;
    }

    function subtract(uint256 value) public {   infinite gas
        currentValue -= value;
    }

    function multiply(uint256 value) public {   infinite gas
        currentValue *= value;
    }

    function divide(uint256 value) public {   infinite gas
        currentValue /= value;
    }

    function getCurrentValue() public view returns (uint256) {   2410 gas
        return currentValue;
    }
}
```

Two types of functions

State-changing functions: Functions that modify state variables, interact with other contracts, or send/receive Ether require a transaction and are considered "non-view" functions.

View functions: These are functions that only read from the blockchain state (like reading a variable) but do not modify it. They can be called without creating a transaction and do not consume gas when called externally via a call.

State changing function

```
contract Calculator {
    uint256 public currentValue;

    constructor(uint256 _initialValue) {   infinite gas 177200 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {   infinite gas
        currentValue += value;
    }

    function subtract(uint256 value) public {   infinite gas
        currentValue -= value;
    }

    function multiply(uint256 value) public {   infinite gas
        currentValue *= value;
    }

    function divide(uint256 value) public {   infinite gas
        currentValue /= value;
    }

    function getCurrentValue() public view returns (uint256) {   2410 gas
        return currentValue;
    }
}
```

View function

```
contract Calculator {
    uint256 public currentValue;

    constructor(uint256 _initialValue) {   infinite gas 177200 gas
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {   infinite gas
        currentValue += value;
    }

    function subtract(uint256 value) public {   infinite gas
        currentValue -= value;
    }

    function multiply(uint256 value) public {   infinite gas
        currentValue *= value;
    }

    function divide(uint256 value) public {   infinite gas
        currentValue /= value;
    }

    function getCurrentValue() public view returns (uint256) {   2410 gas
        return currentValue;
    }
}
```

Try compiling, running this code on remix

https://remix.ethereum.org/

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

contract Calculator {
    uint256 public currentValue;

    constructor(uint256 _initialValue) {
        currentValue = _initialValue; // Set the initial value
    }

    function add(uint256 value) public {
        currentValue += value;
    }

    function subtract(uint256 value) public {
        currentValue -= value;
    }

    function multiply(uint256 value) public {
        currentValue *= value;
    }

    function divide(uint256 value) public {
        currentValue /= value;
    }

    function getCurrentValue() public view returns (uint256) {
        return currentValue;
    }
}
```

Inheritance

Base contract

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

contract Vehicle {
    string public brand;

    constructor(string memory _brand) {    infinite gas 121400 gas
        brand = _brand;
    }

    function description() public pure virtual returns (string memory) {    infinite gas
        return "This is a vehicle";
    }
}
```

Vehicle.sol

Base contract

Can be overriden by a child class

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

contract Vehicle {
    string public brand;

    constructor(string memory _brand) {    infinite gas 121400 gas
        brand = _brand;
    }

    function description() public pure virtual returns (string memory) {    infinite gas
        return "This is a vehicle";
    }
}
```

Vehicle.sol

Child contract

```
import "./Vehicle.sol"; // Import the Vehicle contract

contract Car is Vehicle {
    uint public numberOfDoors;

    // Constructor that also initializes the parent contract
    constructor(string memory _brand, uint _numberOfDoors) Vehicle(_brand) {   infinite gas 1406
        numberOfDoors = _numberOfDoors;
    }

    // Override the description function to make it more specific for a car
    function description() public pure override returns (string memory) {   infinite gas
        return "This is a car";
    }
}
```

Child contract

Calling the parent contracts consturctor

```
import "./Vehicle.sol"; // Import the Vehicle contract

contract Car is Vehicle {
    uint public numberOfDoors;

    // Constructor that also initializes the parent contract
    constructor(string memory _brand, uint _numberOfDoors) Vehicle(_brand) {    infinite gas 1406
        numberOfDoors = _numberOfDoors;
    }

    // Override the description function to make it more specific for a car
    function description() public pure override returns (string memory) {    infinite gas
        return "This is a car";
    }
}
```

Child contract

```
import "./Vehicle.sol"; // Import the Vehicle contract

contract Car is Vehicle {
    uint public numberOfDoors;

    // Constructor that also initializes the parent contract
    constructor(string memory _brand, uint _numberOfDoors) Vehicle(_brand) {   infinite gas 1406
        numberOfDoors = _numberOfDoors;
    }

    // Override the description function to make it more specific for a car
    function description() public pure override returns (string memory) {   infinite gas
        return "This is a car";
    }
}
```

Child class overrides this fn

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

require

```
function setNumber(uint256 _number) public {
    // Use `require` to ensure the number is positive
    require(_number > 0, "Number must be greater than zero");
    storedNumber = _number;
}
```

For loops

```
function addNumbers(uint256[] memory _numbers) public pure returns (uint) {
    uint ans = 0;
    for (uint256 i = 0; i < _numbers.length; i++) {
        ans += _numbers[i];
    }
    return ans;
}
```

If else

```
function categorizeNumber(uint256 _number) public pure returns (string memory) {
    if (_number < 10) {
        return "Small";
    } else if (_number >= 10 && _number < 100) {
        return "Medium";
    } else {
        return "Large";
    }
}
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Mappings

In Solidity, a mapping is a data structure that allows you to store and look up key-value pairs. It's similar to a hash table or dictionary in other programming languages.

Mappings

Create a mapping

```
mapping(address => string) public names;
```

Mappings

Create a mapping

```
mapping(address => string) public names;
```

Insert into a mapping

```
names[msg.sender] = _name;
```

Mappings

Create a mapping

```
mapping(address => string) public names;
```

Insert into a mapping

```
names[msg.sender] = _name;
```

Get from a mapping

```
names[_address];
```

Assignment

Create a User contract where users can come and sign up

Solution

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

contract NameRegistry {

    mapping(address => string) public names;

    function setName(string memory _name) public {   infinite gas
        names[msg.sender] = _name;
    }

    function getName(address _address) public view returns (string memory) {
        return names[_address];
    }
}
```

Is this as simple in Solana?

NO

Arrays

There are two types of arrays in Solidity

Arrays

There are two types of arrays in Solidity

Fixed-size arrays: Arrays where the size is defined when the array is created, and it cannot be changed afterward.
Dynamic arrays: Arrays where the size can change dynamically during execution (i.e., you can add or remove elements).

Fixed size arrays

```solidity
pragma solidity ^0.8.0;

contract FixedArrayExample {
    // Declare a fixed-size array of uint with size 3
    uint[3] public numbers;

    function setNumbers(uint _num1, uint _num2, uint _num3) public {
        numbers[0] = _num1;
        numbers[1] = _num2;
        numbers[2] = _num3;
    }

    function getNumber(uint index) public view returns (uint) {
        return numbers[index];
    }
}
```

Fixed size arrays

```solidity
pragma solidity ^0.8.0;

contract FixedArrayExample {
    // Declare a fixed-size array of uint with size 3
    uint[3] public numbers;

    function setNumbers(uint _num1, uint _num2, uint _num3) public {
        numbers[0] = _num1;
        numbers[1] = _num2;
        numbers[2] = _num3;
    }

    function getNumber(uint index) public view returns (uint) {
        return numbers[index];
    }
}
```

Is this right?

Dynamic Array

Dynamic Array

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

contract DynamicArrayExample {
    uint[] public numbers;

    function addNumber(uint _number) public {    46864 gas
        numbers.push(_number);
    }

    function getNumbersCount() public view returns (uint) {
        return numbers.length;
    }

    function getNumber(uint index) public view returns (uint) {
        return numbers[index];
    }
}
```

Is this right?

Struct

In Solidity, a struct is a custom data type that allows you to group multiple variables (of different types) together into a single unit.

Struct

```solidity
pragma solidity ^0.8.0;

contract PersonContract {
    struct Person {
        string name;
        uint age;
        address addr;
    }

    Person public person;

    function setPerson(string memory _name, uint _age, address _addr) public {
        person.name = _name;
        person.age = _age;
        person.addr = _addr;
    }

    function getPerson() public view returns (string memory, uint, address) {
        return (person.name, person.age, person.addr);
    }
}
```

Struct

```solidity
pragma solidity ^0.8.0;

contract PersonContract {
    struct Person {
        string name;
        uint age;
        address addr;
    }

    Person public person;

    function setPerson(string memory _name, uint _age, address _addr) public {
        person.name = _name;
        person.age = _age;
        person.addr = _addr;
    }

    function getPerson() public view returns (string memory, uint, address) {
        return (person.name, person.age, person.addr);
    }
}
```

Assignment - Can you create a PersonsContract?

Assignment - Can you create a PersonsContract?

```
contract PersonContract {
    struct Person {
        string name;
        uint age;
        address addr;
    }

    mapping(address => Person) public persons;

    function setPerson(string memory _name, uint _age) public {   infinite gas
        persons[msg.sender] = Person({
            name: _name,
            age: _age,
            addr: msg.sender
        });
    }

    function getPerson() public view returns (string memory, uint, address) {
        Person memory person = persons[msg.sender];
        return (person.name, person.age, person.addr);
    }
}
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports

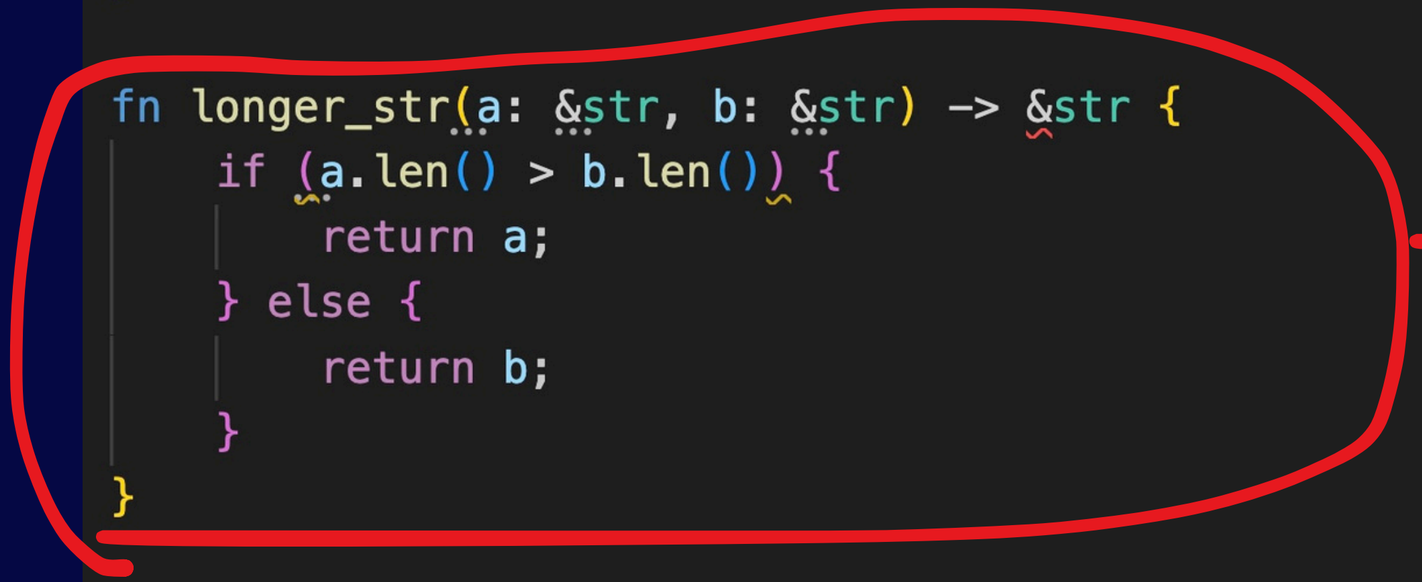
Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Memory vs stack vs storage

Before we get into this,
lets understand whats wrong in the following rust code

```rust
fn main() {
    let ans = longer_str(a: &String::from("harkirat"), b: &String::from("raman"));
}

fn longer_str(a: &str, b: &str) -> &str {
    if (a.len() > b.len()) {
        return a;
    } else {
        return b;
    }
}

fn bigger_number(a: u32, b: u32) -> u32 {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

```rust
fn main() {
    let ans = longer_str(a: &String::from("harkirat"), b: &String::from("raman"));
}

fn longer_str(a: &str, b: &str) -> &str {
    if (a.len() > b.len()) {
        return a;
    } else {
        return b;
    }
}

fn bigger_number(a: u32, b: u32) -> u32 {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```



Throws an error

```rust
fn main() {
    let ans = longer_str(a: &String::from("harkirat"), b: &String::from("raman"));
}

fn longer_str(a: &str, b: &str) -> &str {
    if (a.len() > b.len()) {
        return a;
    } else {
        return b;
    }
}

fn bigger_number(a: u32, b: u32) -> u32 {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Doesnt throw an error

Stack variables (numbers)

Jargon #2 - Ownership

petal-estimate-4e9.notion.sit...

Complete rust bootca... / Jargon #2 - Ownership

Built with

Stack variables

Example #1 - Passing stack Variables inside functions

Rust

Copy

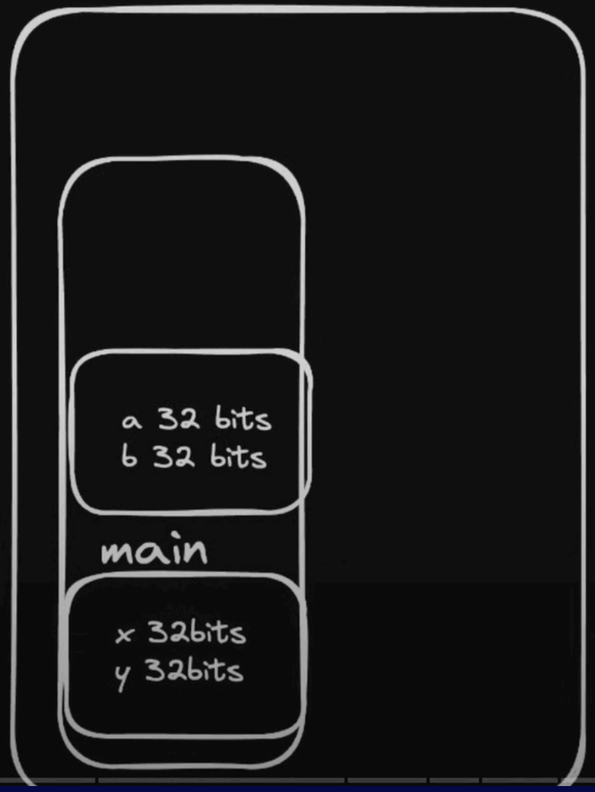
```rust
fn main() {
    let x = 1; // crated on stack, owner is
    let y = 3; // created on stack, owner i
    println!("{}", sum(x, y));
    println!("Hello, world!");
}

fn sum(a: i32, b: i32) -> i32 {
    let c = a + b; // owner of c, a, b is t
    return c;
}
```

(3) Jargon #2 - Ownership | ChatGPT | Excalidraw

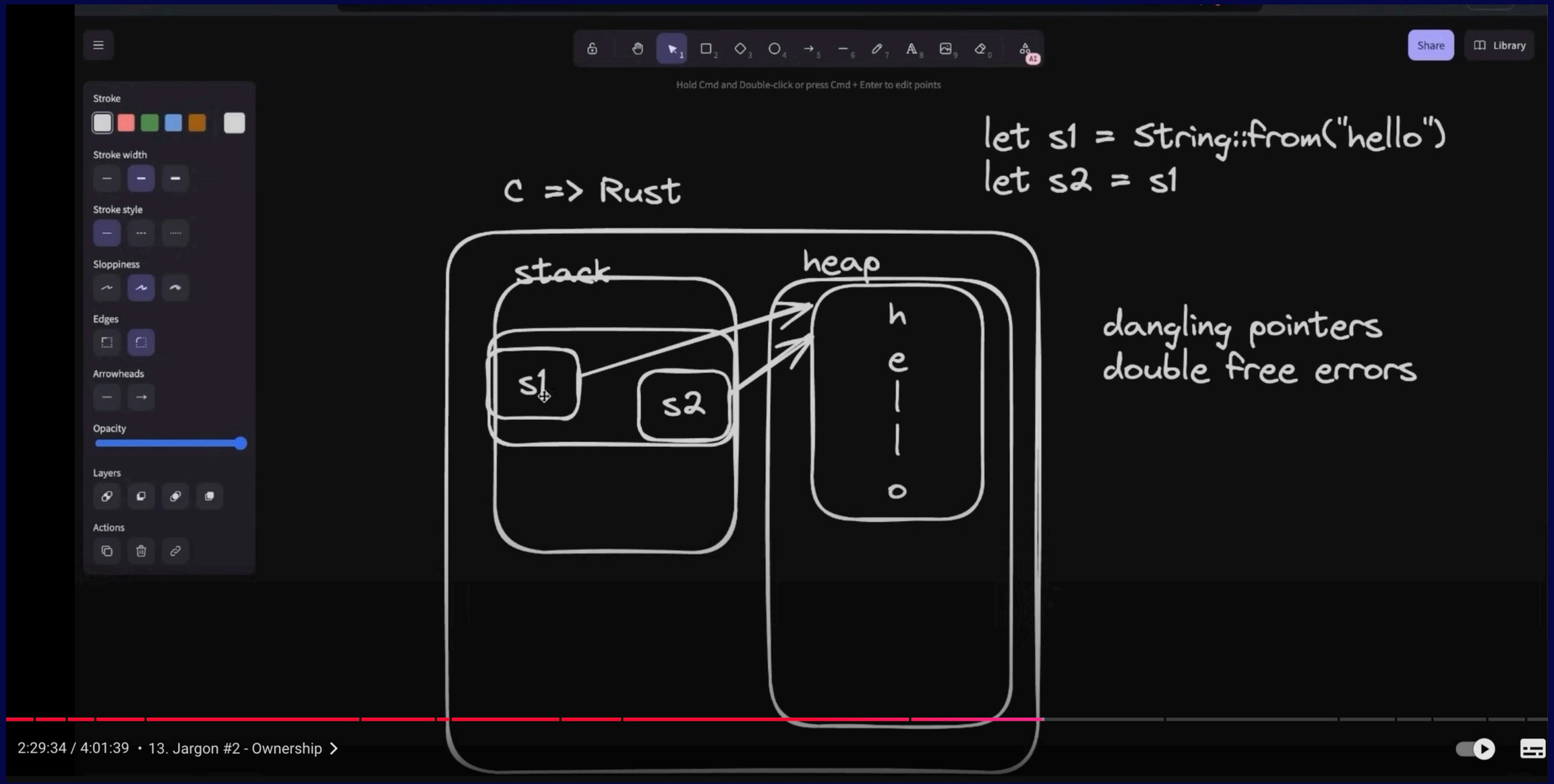
excalidraw.com

To move canvas, hold mouse wheel or spacebar while dragging, or use the hand tool

Share | Library



Heap (String)


Share
Library
Hold Cmd and Double-click or press Cmd + Enter to edit points
Stroke
Stroke width
Stroke style
Sloppiness
Edges
Arrowheads
Opacity
Layers
Actions
let s1 = String::from("hello")
let s2 = s1
C => Rust
stack
heap
s1
s2
h
e
l
l
o
dangling pointers
double free errors
2:29:34 / 4:01:39 • 13. Jargon #2 - Ownership

Memory vs stack vs storage

In Solidity, Memory, Stack, and Storage are three distinct locations where data can be stored. Each has its own characteristics, use cases, and costs.

Storage

Storage refers to the persistent data that is saved on the blockchain. It is used for state variables that you declare at the contract level. Data stored in storage is written to the blockchain and remains there permanently, across function calls and transactions, until it is explicitly modified.

Storage

Storage refers to the persistent data that is saved on the blockchain. It is used for state variables that you declare at the contract level. Data stored in storage is written to the blockchain and remains there permanently, across function calls and transactions, until it is explicitly modified.

Writing to storage is costly in terms of gas because it requires changes to the blockchain state, which involves network consensus and storage allocation on the blockchain.

Memory

Memory refers to temporary data storage that only exists during the execution of a function. It is cheaper than storage because it is not stored on the blockchain and is only kept in the node's memory while the function is executing. Once the function finishes execution, the data is discarded.

Memory

Memory refers to temporary data storage that only exists during the execution of a function. It is cheaper than storage because it is not stored on the blockchain and is only kept in the node's memory while the function is executing. Once the function finishes execution, the data is discarded.

Temporary: Data in memory is erased once the function execution ends.

Cheaper than storage: Writing to memory is significantly cheaper in terms of gas costs compared to storage because it does not involve writing to the blockchain.

Not persistent: Data in memory is not stored permanently and cannot be accessed outside the function that created it.

Used for Function Arguments/Local Variables: When passing large structures or arrays into functions, they are often stored in memory for cheaper gas consumption.

Memory

Memory

```
contract StringExample {
    function addStrings(string memory a, string memory b) public pure returns (string memory) {
        string memory result = string(abi.encodePacked(a, b)); // Correct usage
        return result;
    }
}
```

Memory

Are these also stored in memory?

```
function addNumbers(uint a, uint b) public pure returns (uint) {
    uint result = a + b;  // result is stored in memory
    return result;
}
```

Memory

Are these also stored in memory?

No, they are stored on the stack

```
function addNumbers(uint a, uint b) public pure returns (uint) {
    uint result = a + b;  // result is stored in memory
    return result;
}
```

Stack

The stack in Solidity is a limited, low-level data structure used to store small, temporary values that are used during the execution of a function. It is akin to a "call stack" in other programming languages. When you call a function, the EVM pushes temporary values (such as function arguments and local variables) onto the stack

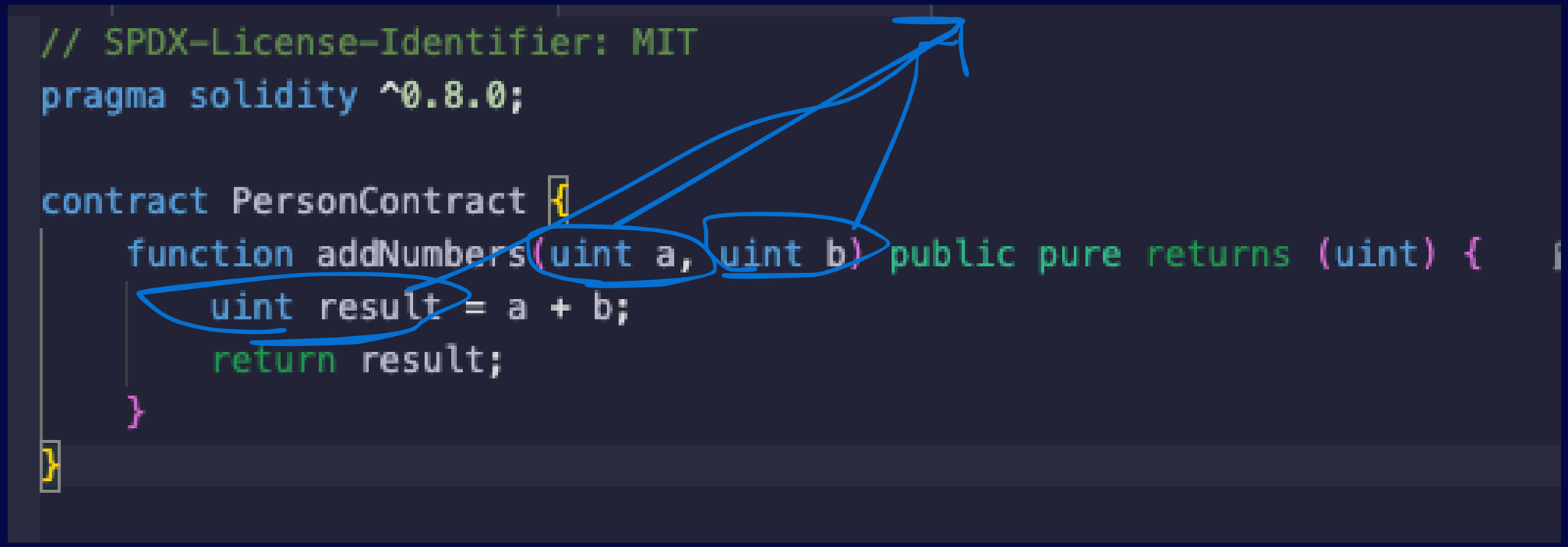
Stack

When a Solidity function is executed, the following steps occur:

1. Push to the stack: Local variables and function arguments are pushed onto the stack. For example, if you have a function with some local variables, those variables will be pushed onto the stack when the function is executed.
2. Pop from the stack: When the execution reaches the end of the function, the EVM pops the local variables and temporary values off the stack.
3. Scope: Variables stored in the stack are local to the function and are only valid during that function's execution. They are cleared as soon as the function finishes executing.

Stack

STORED ON A STACK



Stack

Arrays are also stored in memory

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

contract PersonContract {
    function addNumbers(uint[] memory a) public pure returns (uint) {
        uint ans = 0;
        for (uint i = 0; i < a.length; i++) {
            ans += a[i];
        }
        return ans;
    }
}
```

MODIFIERS

Modifiers

In Solidity, modifiers are a powerful feature that allows you to modify the behavior of functions in a reusable and declarative way. They are used to add additional checks or functionality to a function or group of functions, before or after the main logic is executed.

onlyOwner modifier

```solidity
pragma solidity ^0.8.0;

contract Example {
    address public owner;

    constructor() {  260138 gas 235600 gas
        owner = msg.sender; // Set the deployer as the owner
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "You are not the owner!");
        _;
    }

    // Function that only the owner can call
    function setOwner(address newOwner) public onlyOwner {  26859 gas
        owner = newOwner;
    }

    function sum(uint a, uint b) public view onlyOwner returns (uint) {
        return a + b;
    }
}
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Returning tuples

```
function sumAndMul(uint a, uint b) public view onlyOwner returns (uint, uint) {
    return (a + b, a * b);
}
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

pure functions

In Solidity, pure functions are functions that do not read from or modify the blockchain state. They only rely on their input parameters to perform calculations or operations and return a result. Importantly, pure functions do not interact with any state variables or external contracts.

Not a pure function (reads from state)

```
modifier onlyOwner() {
    require(msg.sender == owner, "You
    _;
}

// Function that only the owner can c
function setOwner(address newOwner) p
    owner = newOwner;
}

function sumAndMul(uint a, uint b) public pure onlyOwner returns (uint, uint) {
    return (a + b, a * b);
}
```

```
TypeError: Function declared as pure, but this expression (potentially) reads from the environment
or state and thus requires "view".
  --> contracts/2_Owner.sol:20:52:
   |
20 |     function sumAndMul(uint a, uint b) public pure onlyOwner returns (uint, uint) {
   |                                                    ^^^^^^^^^
```

```
function sumAndMul (uint a, uint b) public pure
```

Is a Pure function

```
function sumAndMul(uint a, uint b) public pure returns (uint, uint) {
    return (a + b, a * b);
}
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Events

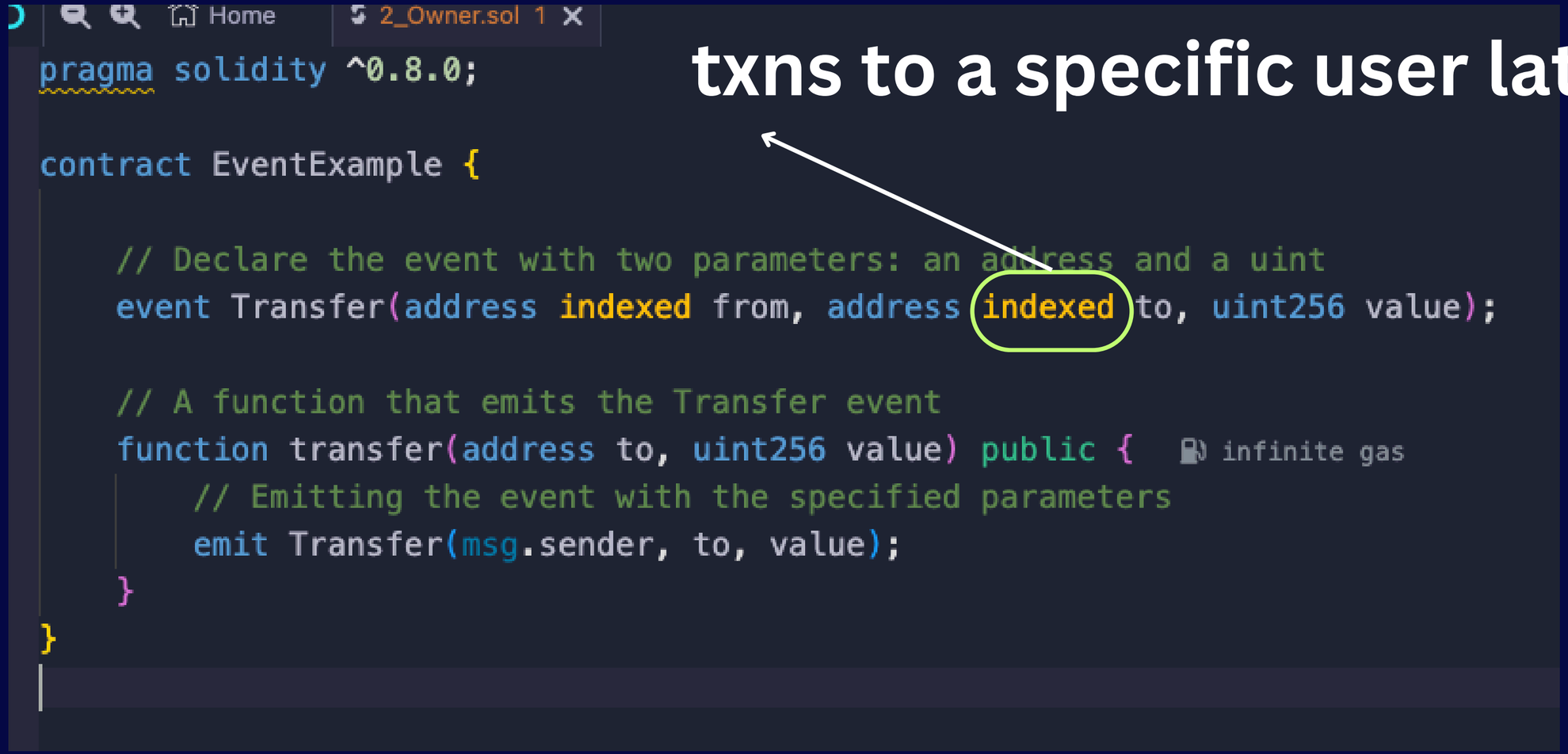
In Ethereum, events are a mechanism that allows smart contracts to log information on the blockchain, which can then be accessed by external consumers (e.g., front-end applications, other contracts, or off-chain services like oracles). Events enable smart contracts to emit logs that can be used for debugging, indexing, or triggering external actions based on contract activity.

Home 2_Owner.sol 1

```solidity
pragma solidity ^0.8.0;

contract EventExample {

    // Declare the event with two parameters: an address and a uint
    event Transfer(address indexed from, address indexed to, uint256 value);

    // A function that emits the Transfer event
    function transfer(address to, uint256 value) public {   infinite gas
        // Emitting the event with the specified parameters
        emit Transfer(msg.sender, to, value);
    }
}
```

Can be used to search all txns from a specific user later

Can be used to search all txns to a specific user later



Get all transactions from a specific address

a.js > ...

```
const Web3 = require('web3');
const web3 = new Web3("http://localhost:8545");

// Contract ABI and address
const contractAddress = "0xYourContractAddress";
const abi = [...
];

// Create a contract instance
const contract = new web3.eth.Contract(abi, contractAddress);

// Listen for the Transfer event
contract.events.Transfer({
  filter: { from: "0xSenderAddress" }, // Optional filtering
  fromBlock: 0
}, (error, event) => {
  if (error) {
    console.error(error);
  } else {
    console.log(`Transfer from ${event.returnValues.from} to ${event.returnValues.to} of value ${event.returnValues.value}`);
  }
});
```

Listen to any new txn

```
const ethers = require("ethers");

// Assume you have a provider connected to an Ethereum node
const provider = new ethers.JsonRpcProvider("http://localhost:8545");

// Contract ABI and address
const contractAddress = "0xYourContractAddress";
const abi = [
  "event Transfer(address indexed from, address indexed to, uint256 value)",
  "function transfer(address to, uint256 value) public"
];

// Create a contract instance
const contract = new ethers.Contract(contractAddress, abi, provider);

// Listen for the Transfer event
contract.on("Transfer", (from, to, value) => {
    console.log(`Transfer from ${from} to ${to} of value ${value}`);
});
```

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

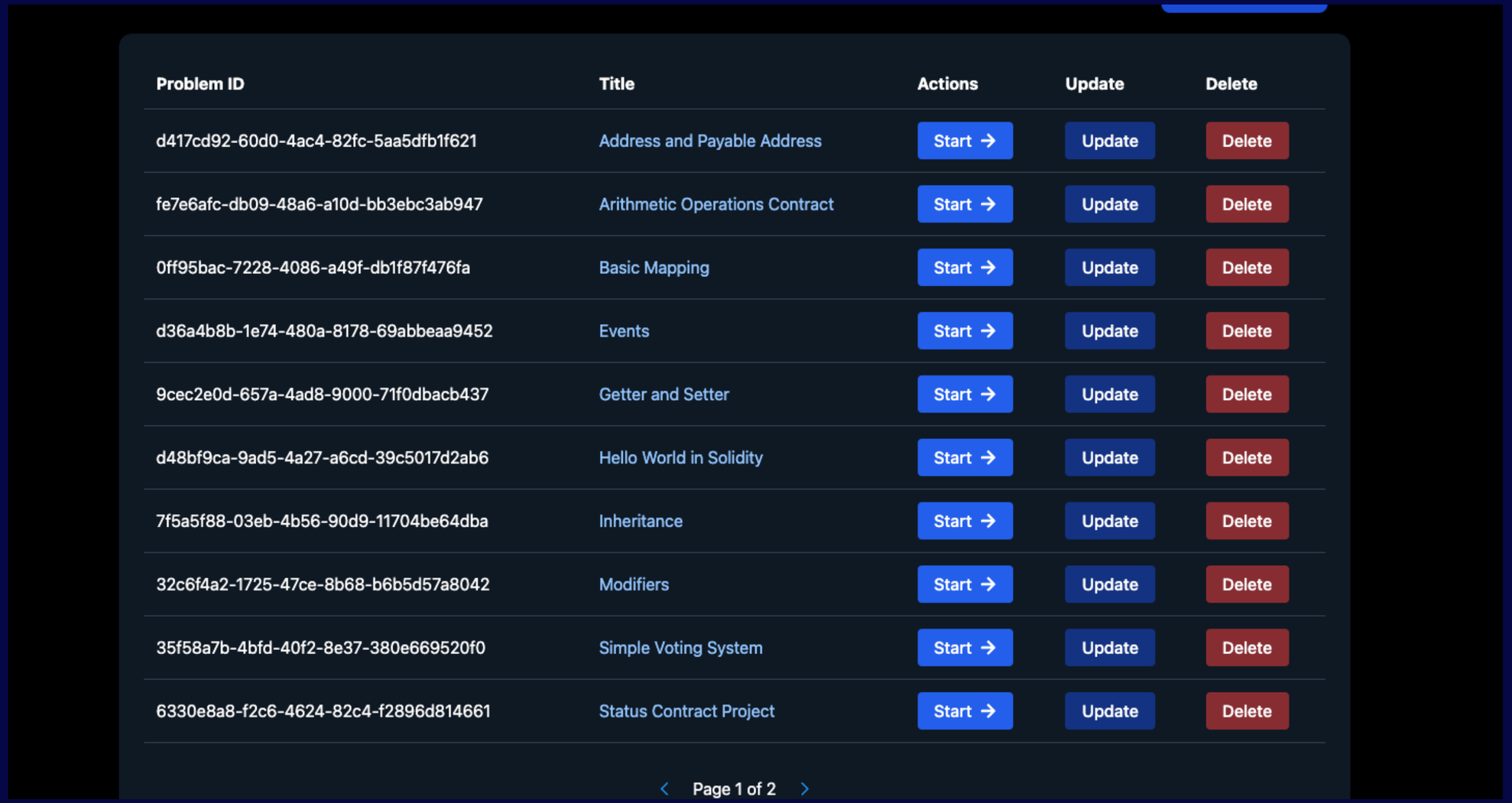
Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

External contracts

```solidity
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";

contract EventExample is Ownable {

    constructor() Ownable(msg.sender) {   infinite gas 239200 gas

    }

    // Declare the event with two parameters: an address and a uint
    event Transfer(address indexed from, address indexed to, uint256 value);

    // A function that emits the Transfer event
    function transfer(address to, uint256 value) public onlyOwner {   infinite gas
        // Emitting the event with the specified parameters
        emit Transfer(msg.sender, to, value);
    }
}
```

Assignments

https://github.com/100xdevs-cohort-3/solidity-assignments



Problem ID	Title	Actions	Update	Delete
d417cd92-60d0-4ac4-82fc-5aa5dfb1f621	Address and Payable Address	Start →	Update	Delete
fe7e6afc-db09-48a6-a10d-bb3ebc3ab947	Arithmetic Operations Contract	Start →	Update	Delete
0ff95bac-7228-4086-a49f-db1f87f476fa	Basic Mapping	Start →	Update	Delete
d36a4b8b-1e74-480a-8178-69abbeaa9452	Events	Start →	Update	Delete
9cec2e0d-657a-4ad8-9000-71f0dbacb437	Getter and Setter	Start →	Update	Delete
d48bf9ca-9ad5-4a27-a6cd-39c5017d2ab6	Hello World in Solidity	Start →	Update	Delete
7f5a5f88-03eb-4b56-90d9-11704be64dba	Inheritance	Start →	Update	Delete
32c6f4a2-1725-47ce-8b68-b6b5d57a8042	Modifiers	Start →	Update	Delete
35f58a7b-4bfd-40f2-8e37-380e669520f0	Simple Voting System	Start →	Update	Delete
6330e8a8-f2c6-4624-82c4-f2896d814661	Status Contract Project	Start →	Update	Delete

‹ Page 1 of 2 ›

Part 1/2

Solidity version
contract keyword
Variables
Data types
If elses
Loops
require
Functions
Arguments
Return types
Function visibility
Inheritance, Exports, imports
external

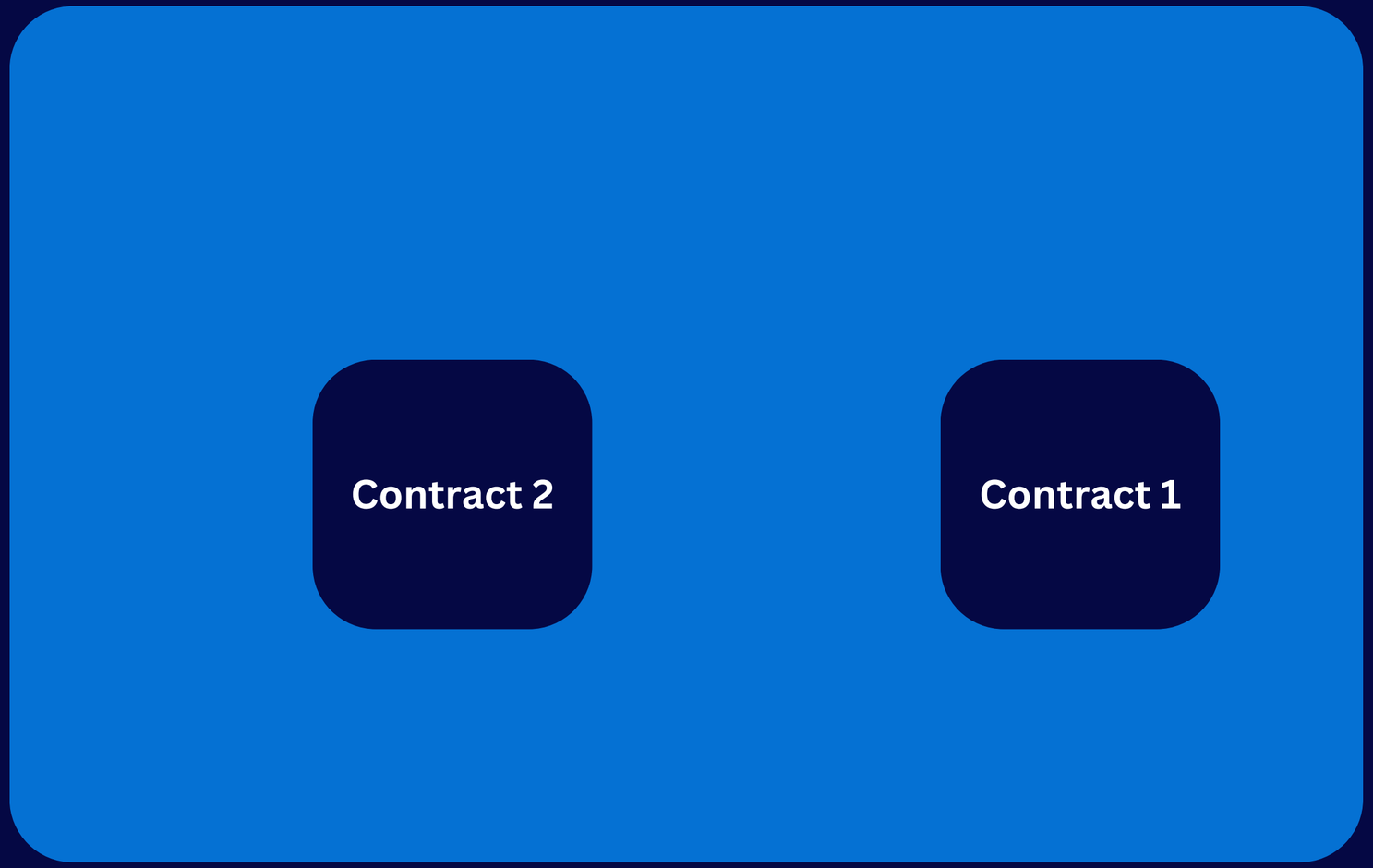
Mappings, Arrays, structs
memory, stack, storage
Modifiers
Custom Modifiers
Returning tuples
pure functions
Events
External contracts, Ownable

Part 3

Interfaces
CCIs
Payable

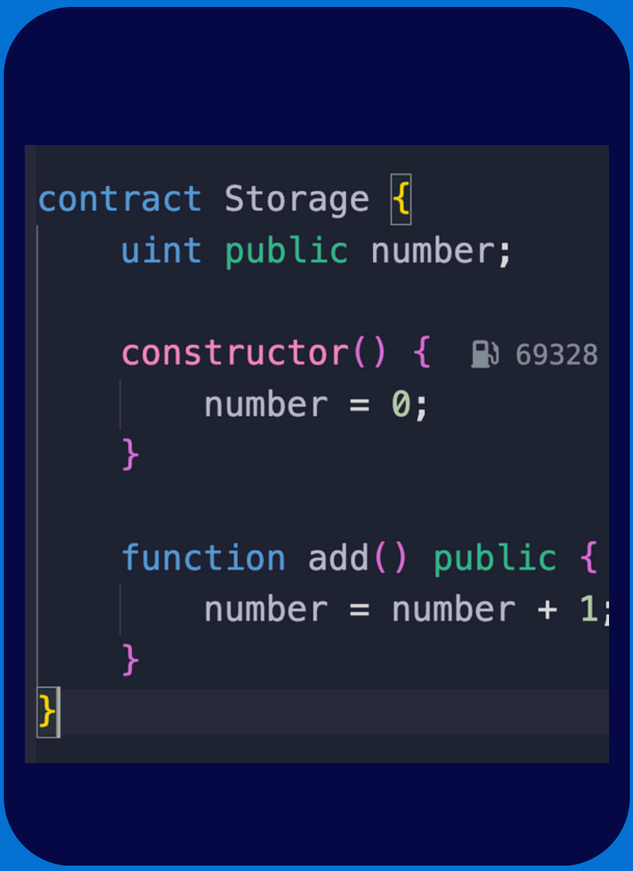
CCI (Cross contract invocation)

CCI (Cross contract invocation)

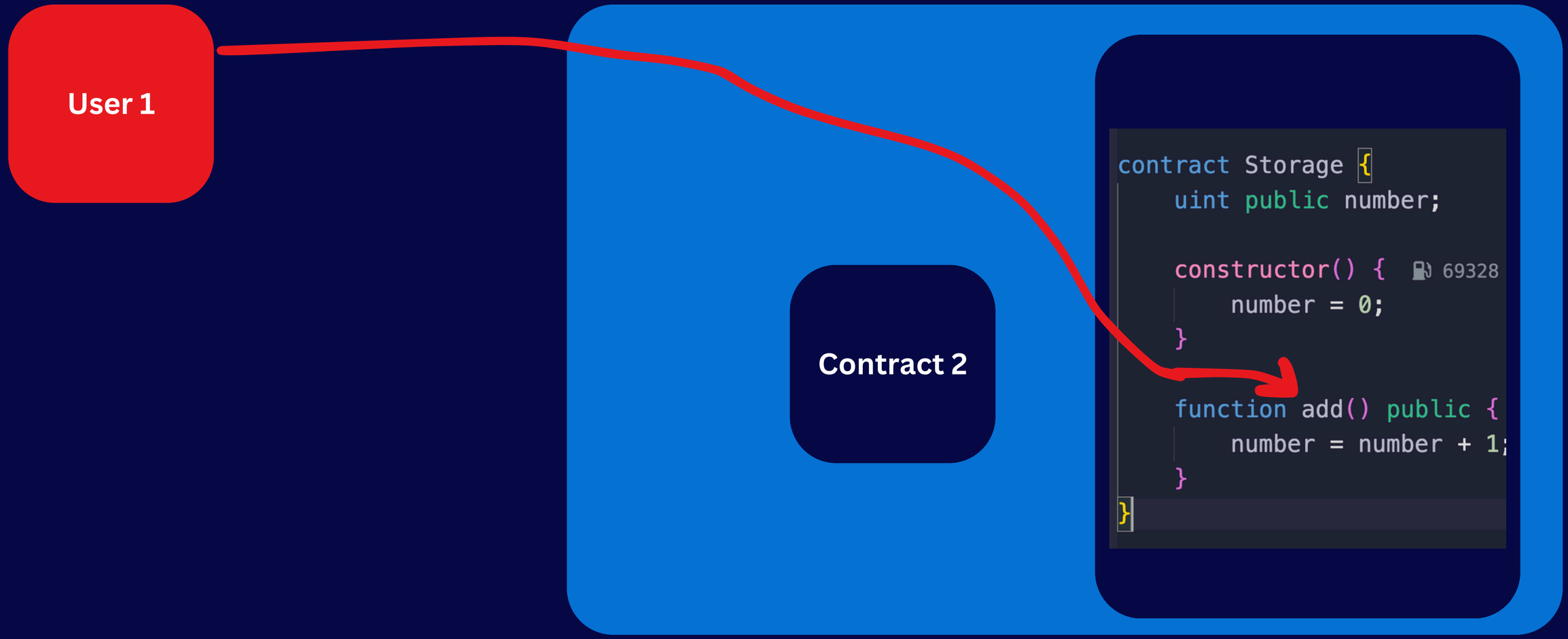


CCI (Cross contract invocation)

Contract 2

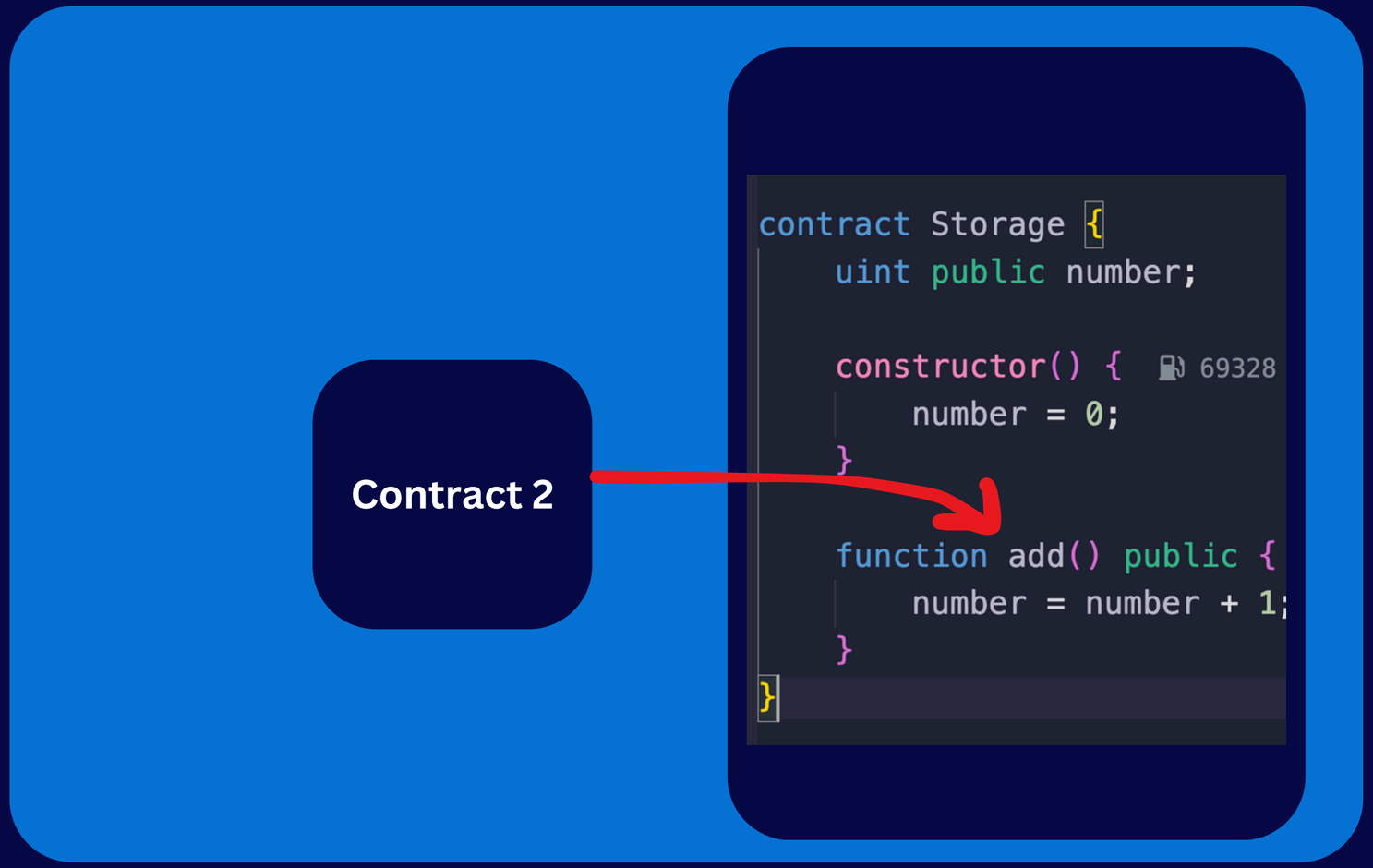


CCI (Cross contract invocation)

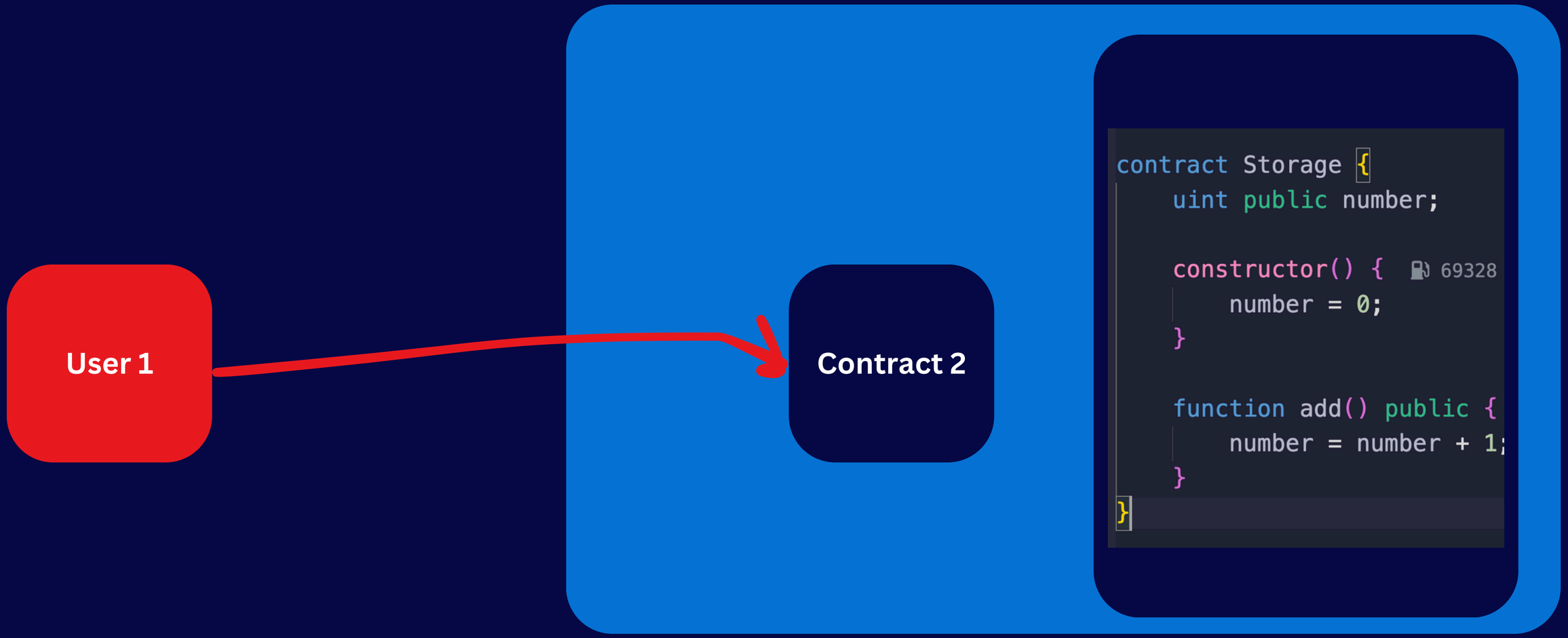


CCI (Cross contract invocation)

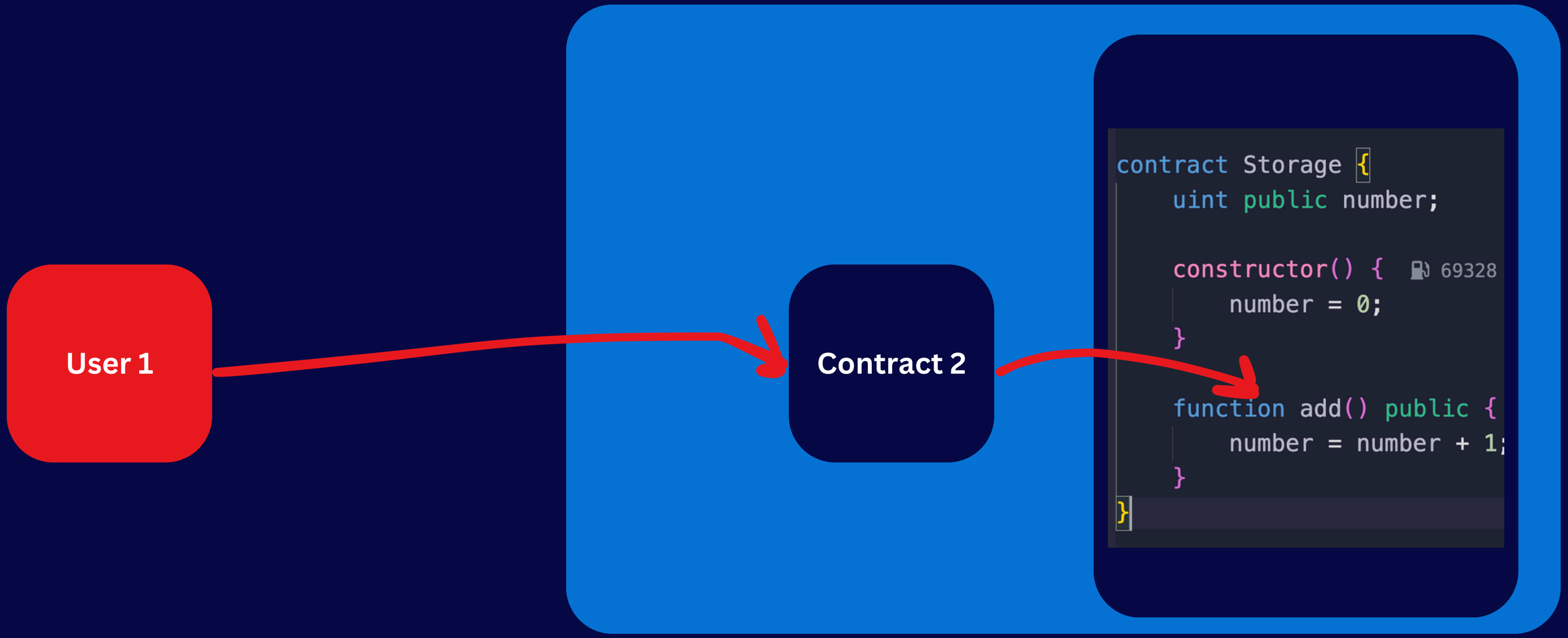
CCI (Cross contract invocation)



CCI (Cross contract invocation)



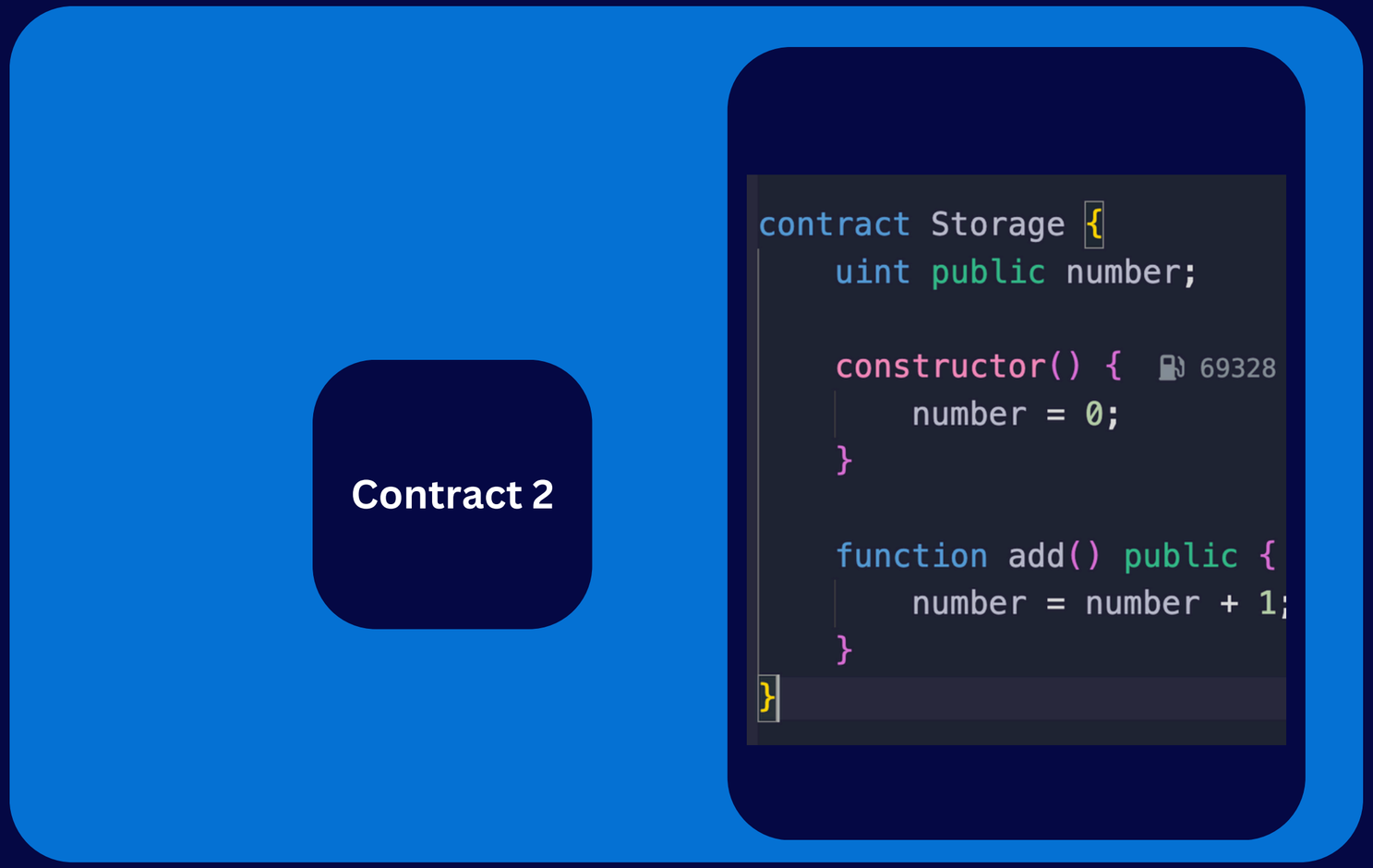
CCI (Cross contract invocation)



CCI (Cross contract invocation)

A cross-contract call refers to a scenario where one smart contract interacts with another smart contract by invoking its functions.

This is a fundamental concept in blockchain programming, enabling modular, reusable, and composable systems.



Before we can call a contract,

we need to define the structure of the contract in an Interface

Interfaces

An interface in Solidity is a way to define a contract's external functions **without** providing their implementation.

Properties of interfaces

1. **Function declarations only:**
    a. **Interfaces only allow function declarations without implementations.**
    b. **Functions must be external.**
2. **No state variables or constructors:**
    a. **Interfaces cannot have state variables or constructors.**
3. **No inheritance from other contracts:**
    a. **Interfaces can inherit only from other interfaces.**
4. **Interactions with other contracts:**
    a. **Interfaces are commonly used to interact with already deployed contracts, enabling modularity and upgradability.**

Lets say we want to CCI into this contract

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

```
contract ParentContract {

}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external; - gas
    function getMessage() external view returns (string memory); - gas
}

contract ParentContract {

}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external; - gas
    function getMessage() external view returns (string memory); - gas
}

contract ParentContract {

}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external; - gas
    function getMessage() external view returns (string memory); - gas
}

contract ParentContract {

}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external;  - gas
    function getMessage() external view returns (string memory);  - gas
}

contract ParentContract {
    address private contractAAddress;

    constructor(address _contractAAddress) {  infinite gas 12400 gas
        contractAAddress = _contractAAddress;
    }
}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external;  - gas
    function getMessage() external view returns (string memory);  - gas
}

contract ParentContract {
    address private contractAAddress;

    constructor(address _contractAAddress) {  infinite gas 12400 gas
        contractAAddress = _contractAAddress;
    }
}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external;  - gas
    function getMessage() external view returns (string memory);  - gas
}

contract ParentContract {
    address private contractAAddress;

    constructor(address _contractAAddress) {  infinite gas 233400 gas
        contractAAddress = _contractAAddress;
    }

    function setMessageFromOtherContract(string memory newMessage) public {
        IMessageContract(contractAAddress).setMessage(newMessage);
    }
    // Get the message from Contract A
    function getMessageFromOtherContract() public view returns (string memory
        return IMessageContract(contractAAddress).getMessage();
    }
}
```



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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Lets say we want to CCI into this contract

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

interface IMessageContract {
    function setMessage(string memory newMessage) external;  - gas
    function getMessage() external view returns (string memory);  - gas
}

contract ParentContract {
    address private contractAAddress;

    constructor(address _contractAAddress) {  infinite gas 233400 gas
        contractAAddress = _contractAAddress;
    }

    function setMessageFromOtherContract(string memory newMessage) public {
        IMessageContract(contractAAddress).setMessage(newMessage);
    }
    // Get the message from Contract A
    function getMessageFromOtherContract() public view returns (string memory
        return IMessageContract(contractAAddress).getMessage();
    }
}
```

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

contract MessagContract {
    string private message;

    // Set a new message
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // Retrieve the stored message
    function getMessage() public view returns (string memory
        return message;
    }
}
```

Payables

Payables

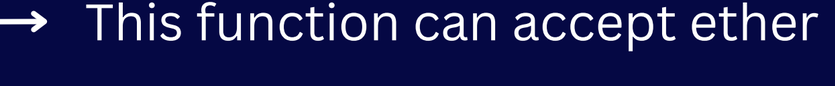
In Solidity, payables refer to functions or addresses that are capable of receiving Ether (ETH) transfers. Solidity is Ethereum's programming language, and Ether is the native cryptocurrency of the Ethereum network. The payable keyword is used to indicate that a function or an address can accept Ether sent to it.

Payables

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

contract MyContract {
    uint public amount;
    function deposit() public payable {   infinite gas
        amount += msg.value;
    }

    function withdraw(address payable recipient) public {
        payable(recipient).transfer(amount);
        amount = 0;
    }

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Payables

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

contract MyContract {
    uint public amount;
    function deposit() public payable {  infinite gas
        amount += msg.value;
    }

    function withdraw(address payable recipient) public {
        payable(recipient).transfer(amount);
        amount = 0;
    }

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```



Payables

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

contract MyContract {
    uint public amount;
    function deposit() public payable {  infinite gas
        amount += msg.value;
    }

    function withdraw(address payable recipient) public {
        payable(recipient).transfer(amount);
        amount = 0;
    }

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Actually transfer the ether