CS6290 Privacy-enhancing Technologies
Tutorial 4

This note is about writing and deploying your first smart contract. You'll learn how to store and retrieve a number on a simulated blockchain using Solidity and Remix IDE.

Prerequisites:

- A web browser (Chrome, Firefox, etc.)
- Remix IDE open in your browser: https://remix.ethereum.org/

1. Writing the Simple Storage Smart Contract (Solidity)

We will start with a smart contract in Solidity, a language for writing smart contracts on Ethereum. Here's the code we used (this example is borrowed from this excellent open course):

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.8;

contract SimpleStorage {

    uint256 favoriteNumber;

    struct People {
        uint256 favoriteNumber;
        string name;
    }
    // uint256[] public anArray;
    People[] public people;

    mapping(string => uint256) public nameToFavoriteNumber;

    function store(uint256 _favoriteNumber) public {
        favoriteNumber = _favoriteNumber;
    }

    function retrieve() public view returns (uint256){
        return favoriteNumber;
```

```
    }

    function addPerson(string memory _name,
    uint256 _favoriteNumber) public {
        people.push(People(_favoriteNumber, _name));
        nameToFavoriteNumber[_name] = _favoriteNumber;
    }
}
```

Explanation of the Code:

- `// SPDX-License-Identifier: MIT`: Specifies the license (MIT License - open source).
- `pragma solidity 0.8.8;`: Specifies the Solidity compiler version to be exactly `0.8.8`. Comments show alternative ways to specify versions or ranges.
- `contract SimpleStorage { ... }`: Defines a smart contract named `SimpleStorage`.
- `uint256 favoriteNumber;`: Declares a **state variable** `favoriteNumber` of type `uint256` (unsigned integer). Note: It's currently **not public**, so a getter function is not automatically created.
- `struct People { ... }`: Defines a **structure** named `People` to group data:
  - `uint256 favoriteNumber;`: A field to store a favorite number (within the `People` struct).
  - `string name;`: A field to store a name (within the `People` struct).
- `People[] public people;`: Declares a **state variable** `people` as a **dynamic array** of `People` structures, with `public` visibility. Creates a getter function to access elements by index.
- `mapping(string => uint256) public nameToFavoriteNumber;`: Declares a **state variable** `nameToFavoriteNumber` as a **mapping** (like a dictionary or hash table) that maps `string` names to `uint256` favorite numbers, with `public` visibility. Creates a getter function to retrieve numbers by name.
- `function store(uint256 _favoriteNumber) public { ... }`: Defines the **function** `store` to set the `favoriteNumber`.
  - `uint256 _favoriteNumber`: Input parameter for the favorite number.
  - `public`: Visibility - anyone can call this function.
  - `favoriteNumber = _favoriteNumber;`: Updates the `favoriteNumber` state variable.

- `function retrieve() public view returns (uint256) { ... }`: Defines a **view function** `retrieve` to retrieve the stored `favoriteNumber`.
  - `public view`: `public` visibility and `view` keyword for read-only access (no gas cost for reading).
  - `returns (uint256)`: Specifies that the function returns a `uint256`.
  - `return favoriteNumber;`: Returns the current `favoriteNumber` state variable.
- `function addPerson(string memory _name, uint256 _favoriteNumber) public { ... }`: Defines the **function** `addPerson` to add a new person with their favorite number.
  - `string memory _name`: Input parameter for the person's name.
  - `uint256 _favoriteNumber`: Input parameter for the person's favorite number.
  - `public`: Visibility - anyone can call this function.
  - `people.push(People(_favoriteNumber, _name));`: Creates a new `People` struct with the provided data and adds it to the `people` array.
  - `nameToFavoriteNumber[_name] = _favoriteNumber;`: Adds an entry to the `nameToFavoriteNumber` mapping, associating the `_name` with the `_favoriteNumber`.

2. Compiling the Smart Contract

1. In Remix IDE, go to the **Solidity Compiler** tab (Solidity logo icon).
2. Ensure the **Compiler** dropdown is set to **"0.8.8"** to match the `pragma solidity` version.
3. Click the blue **"Compile SimpleStorage.sol"** button.

Compilation translates Solidity code to **bytecode** for the Ethereum Virtual Machine (EVM). A **green checkmark** indicates success!

3. Deploying the Smart Contract

1. Go to the **Deploy & Run Transactions** tab (play button with Ethereum logo icon).
2. Set **Environment** to **"JavaScript VM (London)"**.
3. **Account** should be pre-selected.
4. Under **Contract**, select **"SimpleStorage - SimpleStorage.sol"**.

5. Click the blue **"Deploy"** button.

Deployment creates a **contract instance** on the JavaScript VM. Confirmation in the Remix console with a **green tick**. A **"Deployed Contracts"** section appears below.

4. Interacting with the Deployed Contract

In the **"Deployed Contracts"** section, expand the dropdown for `SIMPLESTORAGE AT [address]`. You will see:

- `favoriteNumber` (blue button)
- `retrieve` (blue button)
- `store` (orange button)

4.1. Initial Interaction (Using Account 1 - Deployer)

- **Get Initial Value:** Click the blue `favoriteNumber` or `retrieve` button. Console should show `0`, the initial value for `uint256`.

4.2. Storing and Retrieving a Number

1. **Store a Number:**
    - In the **"Deployed Contracts"** section, find your `SIMPLESTORAGE AT [address]` contract.
    - In the input field next to the orange `store` button, type a **number** (e.g., `42`).
    - Click the orange `store` button. A transaction will be executed.
2. **Verify Updated Value:**
    - Click the blue `favoriteNumber` or `retrieve` button. Console should now show the **number you stored** (e.g., `42`).

5. Further Exploration: Dynamic Greeting Contract

Now, try to work with the Dynamic Greeting contract by yourself! Here is the code:

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

contract DynamicGreeting {
    // State variable for the greeting message
    string public greeting;
    // State variable to store the address of the last
    // person who set the greeting
    address public lastSetter;

    // Constructor to set an initial greeting
    // when the contract is deployed
    constructor(string memory _initialGreeting) {
        greeting = _initialGreeting;
        // 'msg.sender' is the address deploying the contract
        lastSetter = msg.sender;
    }

    // Function to set a new greeting
    function setGreeting(string memory _newGreeting) public {
        greeting = _newGreeting; // Update the greeting message
        lastSetter = msg.sender;  // Update who set the greeting
    }

    // Function to get the current greeting (already
    // created by 'public greeting')
    // Function to get the address of the last setter
    // (already created by 'public lastSetter')
    // You can also add explicit getter functions if you prefer:
    function getGreeting() public view returns (string memory) {
        return greeting;
    }

    function getLastSetter() public view returns (address) {
        return lastSetter;
    }
}
```

Follow the steps below to deploy and interact with it:

5.1. Deploy and Initial Interaction

1. Compile the `DynamicGreeting` contract in Remix.

2. Deploy it to the JavaScript VM, providing an initial greeting during deployment.
3. Verify the initial greeting and `lastSetter` using the blue getter buttons.

5.2. Setting a New Greeting with a Different Account (Account 2)

This section demonstrates how different accounts interact with the same smart contract instance.

1. **Switch to Account 2:** In the **Deploy & Run Transactions** tab, change the **Account** dropdown to **"Account 2"**.
2. **Call `setGreeting` from Account 2:** Set a new greeting using the orange `setGreeting` button.
3. **Verify Updated State (Using Account 2 - Setter):** Check the greeting and `lastSetter`.
4. **Switch Back to Account 1 and Verify (Using Account 1 - Original Deployer):** Check the greeting and `lastSetter` again.

Key Learning from Dynamic Greeting Exploration:

- **Different Accounts, Shared Contract:** Multiple accounts interact with the *same* deployed contract instance.
- **`msg.sender` Identifies Caller:** `msg.sender` dynamically reflects the address of the account currently calling a function.
- **Persistent State Changes:** Changes to state variables are persistent and shared.
- **Transactions for State Changes:** Calling `setGreeting` requires a transaction.

6. Key Takeaways

- **Smart contracts** are code on a blockchain for automated agreements.
- **Solidity** is a language for Ethereum smart contracts.
- **State variables** store data in a contract.
- **Functions** define actions on a contract.
- `public` visibility for state variables creates **getter functions**.
- `view` functions are **read-only** (no gas for reading).
- Functions that **change the state** of the contract require **transactions** and **cost gas**.
- `uint256`, `string` and `address` are common data types.
- **Remix IDE** is a tool for smart contract development.
- **`msg.sender`** is the address of the account calling the function.

7. Next Steps

You can continue learning with:

- More Solidity tutorials: Explore CryptoZombies - an interactive game to learn Solidity.
- Solidity Documentation: Dive deeper with the official Solidity documentation.
- Building more complex contracts.
- Exploring different Solidity data types.
- Thinking about real-world smart contract applications. You can consider to explore our tutorial at IEEE SecDev2018.

Keep building and exploring the world of blockchain!