Homework Assignment 1

- Due: 11:59pm on Monday, October 20, 2025 (Pacific Time)
  - Late Submission Due: 11:59pm on Monday, October 27, 2025 (Pacific Time)
- Submit via Gradescope (Course Entry Code: VW3K2R)
- Starter Code: hw1-starter.zip
  - Foundry is required in order to build, run and test the code if you built on top of the starter code.
- Contact *Yanju Chen* on slack if you have any questions.

Overview and Getting Started

Decentralized apps run autonomously on top of blockchains. By safeguarding user privacy, many apps are built as smart contracts for various application scenarios, e.g., voting, gaming, financial management, etc. In this homework, we will be building a prototype of decentralized property management system in Solidity programming language.

Well, because it's a prototype, it's designed to do only a few simple things. Let's call it `RentManager`, and we will use it to coordinate hosts and guests, where hosts can send invoices to guests living in their properties, and guests can pay them --- that's it!

Even though we are writing smart contracts for blockchain, we can still run them locally with the help of Foundry. To start working on this homework assignment, it's highly recommended that you finish setting up Foundry on your local machine, by finishing at least the following two steps:

1. Install Foundry. Simply check out the "Installation" section of https://book.getfoundry.sh/getting-started/installation.
2. Test out Foundry with a simple case. Just follow: https://book.getfoundry.sh/getting-started/first-steps.

You are welcome to read more about Foundry in their official documentation called Foundry Book (https://book.getfoundry.sh/), which will be super helpful for the next homework assignment; but for this homework, you only need the above two steps in order to get started.

After you are done with Foundry, you may get the starter pack of this homework assignment. The starter pack is organized as a Foundry project, where you will be filling in the missing implementation for the `RentManager.sol` file in the `src/` folder. Read the specification below as well as the comments in `RentManager.sol` to learn more about what behavior is expected for each function.

High-Level Specification

> *You can find more detailed specification from the comments for each function in `RentManager.sol`. This section provides a high-level description of functionality.*

In this project, we use "user" and "address" interchangeably. That means, we identify an user by her address, which is similar to how Ethereum works.

User Registration

The user registration runs access control of the system: The major functionality of the system is only available for registered users; therefore, one needs to register before using it. A user can register herself by calling one of the following two registration functions:

- `function registerHost() public returns (bool)` will register the user as a host and return `true` if it executes successfully; otherwise `false` is returned.
- `function registerGuest() public returns (bool)` will register the user as a guest and return `true` if it executes successfully; otherwise `false` is returned.

Balance Management (Already Implemented)

> The functions below have already been implemented for you, so you don't need to do anything unless you want to implement your own version.

The system needs utilities to keep track of the balance of each user. Balance of guests can be used to pay the invoices sent by the hosts. The starter pack has the following functions for balance management that are already implemented for you:

- `function addBalance(uint256 amount) public returns (bool)` will increase the caller's balance by `amount`.
- `function viewBalance() public view returns (uint256)` will return the caller's current balance.
- `function viewRole() public view returns (uint8)` returns the current role of the caller, which could be 0 (unregistered), 1 (admin, the one that deployes the contract), 2 (host) or 3 (guest). Find more details in `RentManager.sol`.

Note that since it's a prototype, there isn't any real transaction happening on the blockchain that transfer any token from somewhere to this system; the balance management functions are just mock functions the you can call before testing the invoice management functions.

Invoice Management

The invoice management provides three major functions to model interactions between hosts and guests:

- `function sendInvoice(address toAddr, uint256 amount) public returns (bool)` allows a host to send out an invoice to a guest. Before creating an invoice, the admin will collect a fixed amount of `service_fee` from the host that initiates the invoice.
- `function viewInvoice() public view returns (uint256, uint256, address, address)` returns the information of the most recent invoice related to the caller: if the caller is a host, it returns the most recent invoice sent; if the caller is a guest, it returns the most recent invoice received.
- `function payInvoice(uint256 amount) public returns (bool)` allows a guest to pay her most recent invoice received with `amount` from her balance.

For simplicity, a host can only have one active invoice (i.e., invoice that hasn't been fully paid out); the same applies to a guest.

Local Test Cases

The starter pack comes with some Foundry test cases that you can use for debugging. They are located in `test/RentManager.t.sol`. To test your implementation, simply invoke Foundry from the root path of the starter pack and issue the following command:

```
forge test
```

You will then see a series of automated testing running.

You can also write your own test cases by appending more test functions in `test/RentManager.t.sol`.

Submission and Evaluation

You will be sumitting only the `RentManager.sol` file via Gradescope. You can build upon the provided template in the starter pack, or build entirely on independently on your own --- either way is great, but *make sure your submitted smart contract has the same signature for each function* as defined in the template of `RentManager.sol`. The autograder relies on the function signatures to locate and test different usage defined in the specification.

The maximum points you can get is 60 pts (25 test cases), which break down to:

- 15 pts (1 test case): submission of a non-empty and compilable `RentManager.sol` file.
- 1 pt (1 test case): passing test of admin functionality.
- 3 pts (2 test cases): passing tests of non-registered user functionalities.
- 24 pts (11 test cases): passing tests of guest user functionalities.
- 17 pts (10 test cases): passing tests of host user functionalities.

You get different amount of pts for each test case passed based on each one's importance.

Note that:

- The extra 15 pts won't be awarded if you only submit an empty solution; so please do try your best to understand the problem and write down the solution.
- There's a grace period of 1 hour after the regular deadline. Note that the grace period is intended for you to wrap up and save your unfinished work for submission at the last minute, and if you could not catch that, your subsequent submissions will be marked as late submissions.
- The points you get will be converted to percentage when computing score of this homework towards your final grade. For example, if you get 60/60 pts, you get 100% of the homework score, and if you get 30/60 pts, you'll get 50% of the homework score, etc.
- Points earned from late submission will be discounted by 50%. Late submission is due 1 week after the regular deadline. Subsequent submissions after the late submission deadline will receive 0 point. There's no grace period of late submission deadline.

Hints and Useful Resources

- You may find this Solidity language examples useful: https://solidity-by-example.org/
- Foundry itself is a tool that provides powerful testing utilities for Solidity. Check out its documentation here: https://book.getfoundry.sh/

Academic Integrity

Please refer to UCSB's adacemic integrity guidance (here) if you have any questions.