COMP26020: Programming Languages and Paradigms
Lab 5 - Solidity

Joseph Razavi and Richard Banach

1 Introduction

This lab exercise is about learning a programming language with unusual aspects from its documentation. We focus on the Solidity programming language, in particular Solidity version 6, which you can read about here:

https://docs.soliditylang.org/en/v0.6.0/

Solidity is a language designed to write so-called "smart contracts". These are pieces of code which are supposed to run on a public "blockchain" – a system which keeps a log of every event which happens, and where no user can single-handedly affect what happens. That means that once your code is deployed, you can no longer influence it, unless you have programmed mechanisms to do so. And if you find a bug, the bug is there forever!

In addition, the blockchain is designed to support payments of various kinds – for instance a smart contract has a balance of currency (called 'wei' for the Etherium blockchain on which Solidity contracts run) which it must use to pay for its own computing resources. Contracts can charge each other and pay each other for services.

Whether or not any of this is a sensible technical or social project is perhaps debatable, but it certainly creates interesting design challenges for a programming language – and where weird programming languages lead, let us follow!

Read about Solidity's notion of a contract, and its execution model (the 'Ethereum Virtual Machine') here:

https://docs.soliditylang.org/en/v0.6.0/introduction-to-smart-contracts.html

Refer to the Solidity documentation to complete the exercises below. Aside from the above these sections are particularly useful:

- https://docs.soliditylang.org/en/v0.6.0/solidity-by-example.html
- https://docs.soliditylang.org/en/v0.6.0/solidity-in-depth.html

If you prefer videos, I have made available on Blackboard some videos designed to help you get started. Note these videos belong to the lab and are not part of the content of any week. Solidity will be used only for the lab, and is not examinable.

In this lab exercise, rather than deploying our code on the real public blockchain (and having to pay to run it!) we will use a simulated version of the Etherium Virtual Machine which is used for developing code and testing it before deploying it for real. You must use the version provided on Blackboard; see next section.

2 Setup

Make sure you have downloaded Remix from Blackboard:

https://online.manchester.ac.uk/bbcswebdav/courses/I3132-COMP-26020-1221-1YR-040494/remix-d624303.zip

(If the link above does not work, check the Lab 5 folder on Blackboard for information.)

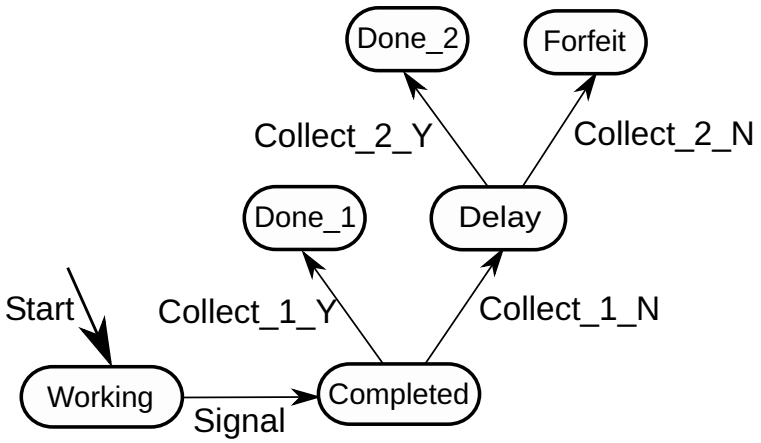
and that you can compile and run programs. To do this, you might need to click on the 'plug' icon on the left hand menu, and made sure 'Solidity compiler' and 'deploy and run transactions' are enabled. This will let you compile and run Solidity programs in Remix as seen in the videos. Remix is a browser based editor, and has been tested for this course on Google Chrome on Linux and Windows. With other browsers you may get strange behaviour. It is better to edit in a separate text editor and paste into Remix for testing, as it can have problems with saving files and allowing text to be copied out of it in some browsers. Make sure you always have a copy of your code in another editor so that you don't lose your work. Clone the gitlab repository

`COMP26020-lab5-S-Solidity_username`

where username is replaced by your username. This contains the files you will need for the exercise.

3 Background

The exercises concern three contracts which should interact with each other, alongside other contracts which we assume exist (but do not implement or worry about the implementation of). The first contract we consider is a 'paylock'. The idea is that a supplier does some work, which can then be collected by a customer. If the customer collects early, they get a discount, and how much discount they get depends on how early: there are two deadlines. If they miss the second deadline they forfeit their discount altogether.



The blobs indicate possible states of the paylock, and the arrows represent function calls. The 'Start' arrow represents the constructor. The idea is that the functions should only succeed if the paylock is in the state at the beginning of the arrow, and then the resulting state should be the one at the end. Of course, there are other conditions: `collect_1_Y` should only succeed if called before the first deadline, and `collect_1_N` should only succeed if called once the first deadline has passed; similar considerations apply to the other two `collect` functions. Look in the file `paylock.sol` to see a partially finished implementation of the paylock. The first two exercises (see next section) concern only the logic of the paylock. They are about adding features to the implementation, though we never complete a realistic implementation.

The subsequent exercises are about implementing a supplier which has to interact with both the paylock contract and a rental contract which it needs to use to complete its work. As above, we will only model

certain aspects of these contracts. On the one hand this makes the exercises manageable, but on the other hand it can be confusing if not pointed out: you would naturally wonder when we would add the rest of the necessary features!

4 Exercises

The implementation of the paylock which you are given does not model the passage of time. To do this, we will add a tick function, representing the passage of one unit of time. We shall assume for the moment that the tick function is going to be called by a neutral third party, who we trust to call it at a regular interval. For now we also trust all other contracts in the universe not to call this function. (And assume that the blockchain updates quickly enough that this is a reasonable model of time! This is not how one would deal with time in a real smart contract system.)

EXERCISE 1: (2 marks)

Add an `int` variable `clock` and a `tick` function which models the passage of time. Modify the various collect functions to adhere to the deadlines, where we consider the first deadline to happen if the clock has reached 4 units of time or more, and the second deadline to be when the clock has increased by 4 units of time or more from when `collect_1_N` was called.

We now need to make sure this `tick` function can only be called by the agreed third party.

EXERCISE 2: (2 marks)

Add an address variable `timeAdd` to the contract. Add an argument to the constructor and set the value of `timeAdd` to that argument. Now modify `tick` so that it can only be called by someone from the address `timeAdd` .

Tip: when testing your code, copy one of the addresses from the 'Account' dropdown menu and paste it into the constructor argument. That should make it easier to experiment.

Look in the file `supplier.txt` and paste its contents at the end of `paylock.sol` . Note how the `Supplier` contract interacts with the paylock, indicating to the paylock when it has finished its task. In the next exercise, we will make it interact with the `Rental` contract too. The idea is that in order to finish its job, the `Supplier` must rent a resource, then return it, before calling finish will succeed.

EXERCISE 3: (2 marks)

Add functions `aquire_resource` and `return_resource` which must be called in that order to the `Supplier` contract. To do this you will need to add new local variables. Add a local variable representing an instance of the `Rental` contract, and allow the address of an instance of `Rental` to be passed as an argument to the constructor. Modify the `aquire_resource` and `return_resource` functions so that they call the appropriate functions of the Rental contract.

Tip: Since the constructor of `Supplier` requires the addresses of a `Paylock` and a `Rental`, make sure you deploy instances of those first when testing.

We will now make our model of the `Rental` contract somewhat more realistic, by requiring the payment of a deposit which is returned once the rented resource is re- turned. For the purposes of the lab we assume that the deposit is `1 wei`.

Since the `Rental` contract is not supposed to assume that it is being called be a `Supplier`, it should assume that the contract it is connected to implements a `receive` function; you can read about this in the Solidity language documentation:
https://docs.soliditylang.org/en/v0.6.0/contracts.html#receive-ether-function.

Since we are not allowed to assume the calling contract is a `Supplier`, it is also useful to look at the functions which can be applied to any address:

https://docs.soliditylang.org/en/v0.6.0/types.html#members-of-addresses .

In fact, our intention is to make as few assumptions about the other contract as possible, so we will use the low-level `.call()` function. Find out how to make this work and attach a value to it.

EXERCISE 4: (2 marks)

Modify the `Rental` contract in the following way. First find the commented line
`//CHECK FOR PAYMENT HERE`
and replace it with something which prevents the function from succeeding unless proper payment is made. You will also have to make the functions payable. Then find the commented line
`//RETURN DEPOSIT HERE`
and replace it with a single use of the `.call` function which returns the deposit. Modify the `Supplier` contract so that it has a `receive` function, and make sure that `Rental` does not assume that the contract which calls its functions is an instance of `Supplier`. Modify the external function calls made by `Supplier` to `Rental` so that they transfer the deposit as appropriate.

At this point you should **copy the file paylock.sol to supplier2.sol and work in supplier2.sol** .

The rental contract as implemented has a security flaw which is described in (which is described in the 'Reentrancy' section of chapter 9 of Antonopoulos's book Mastering Etherium (available online from the library, and also at

https://github.com/ethereumbook/ethereumbook/blob/develop/09smart-contracts-security.asciidoc

EXERCISE 5: (1 mark)

Modify the `Supplier` contract to take advantage of this security flaw to take more Ether belonging to the `Rental` contract than it has sent to the contract, if more ehter is available. Make sure this work is saved in the file `supplier2.sol`

At this point you should **copy the file supplier2.sol to suppler3.sol and work in supplier3.sol** .

EXERCISE 6: (1 mark)

Re-order the lines of the `retrieve_resource` function of the `Rental` contract so that the vulnerability above is fixed. Make sure this work is saved in the file `supplier2.sol`

Note: You need only prevent the attack described here while preserving correct functionality; you do not need to solve any other security flaws.

5 Submission

Submission is by gitlab, following the same procedure as the other labs for this unit. Ensure that you have pushed a commit containing your submission (i.e. make sure you have added all files to the repository), tagged with the tag **lab5-submission** , by **6pm on 05/05**.

Check SPOT to make sure your submission has been received correctly, and contact me (Joe) if you notice any strange behaviour from SPOT.