

Tutorial 1: A global and censorship-resistant “hello world”

Patrick McCorry

Kings College London, UK
`patrick.mccorry@kcl.ac.uk`

Abstract. An introduction into Solidity, Remix and how to write a hello world smart contract. It is recommended to make notes on this tutorial sheet for future use.

1 Introduction to Smart Contracts

Before deep-diving into building a smart contract, we must first consider the basics. We’ll cover the best way to think about smart contracts, what tools we can use to write smart contracts, and some constraints we’ll need to consider.

1.1 What is Smart Contract?

A smart contract can be viewed as a trusted third party with public state. We consider it a programmatic third party as once the code is deployed, the smart contract is autonomous and no one can tamper with the code it is executing.¹ All contracts have a unique address on the network and conceptually there is only single (and global) instantiation of the contract.² As well, it is a finite state machine, where its next state is computed based on the current state and a set of inputs (i.e. an authorised command).

1.2 How do we interact with a smart contract?

Anyone can interact with a smart contract by proposing a transaction. A transaction references the smart contract’s address and it will carry a command in its payload. As we’ll learn later, the transaction must be digitally signed and the signer will have to pay for it. Once the transaction is signed, it can be published to the peer-to-peer network with the aim to reach a set of privileged peers, the miners, who have the authority to include it in the blockchain. The transaction (and thus the command’s execution) is only considered final when it is accepted into the blockchain. As we’ll learn in class, the blockchain only provides *eventual consistency* and it is possible for the transaction to be confirmed, unconfirmed, and then re-confirmed at the blockchain’s tip.

¹ As we will learn, we can’t change its functionality, but there is some room to interfere with the order of execution.

² In reality, every single computer running Ethereum will have a copy of the program, and re-execute every transaction for it.

1.3 What role does the blockchain play?

The blockchain is simply a global and cryptographic audit log. It contains the entire transaction history and all transactions are sequentially ordered. It lets anyone in the world download the blockchain to re-execute every transaction and eventually re-create the same copy of a database as everyone else. In effect, this creates all smart contracts, it re-executes every state transition, and eventually the user will have the same global view of all smart contracts as everyone else. The best way to think about smart contracts and the blockchain is *replication computing* as every peer on the network will eventually execute your transaction!

1.4 Who pays for global and replicated execution?

Yes, of course! There are plenty of different pricing models deployed in practice that may require the user to pay in real-time or for the smart contract developer to pay upfront. For simplicity, we'll only consider Bitcoin and Ethereum.

Bitcoin Fee Model It is pretty straight-forward. We just pay the bytesize of a transaction (i.e. 1 satoshi per byte). This simplicity is due to the restricted computational scripting language as it does not natively support expressive smart contracts. For most of our tutorials, we'll not consider Bitcoin smart contracts for that reason.

Ethereum Fee Model Ethereum has a virtual machine to support expressive smart contracts. Every operation code (opcode) in the virtual machine is associated with some *gas* and the network's consensus rules ensure everyone agrees to the same list of fixed gas costs. The user can decide a *gas price* they are willing to pay for the gas they want to consume (i.e. they need to pay for enough gas to execute 10 opcodes). Generally, it is a bit like filling your car at the gas station.

1.5 What limitations should we consider right now?

Computationally constrained programs All computation and storage is replicated across the world, and everything has a price. If you try to write a smart contract to perform a large-scale binary search (or even something silly like Angry Birds), it'll require more computational power than the entire network's capacity. The best way to think of a public blockchain like Ethereum is a computer from the 1970s as its slow, expensive and can't do very much. (Thankfully, we just escaped from the assembly language era). As we'll cover in week 3, this lack of scalability is due to a fundamental tradeoff between reaching global consensus and preventing 'forks' on the network.

No privacy - everything is public All smart contracts, transactions and information is replicated across the world. If your date of birth is used in an Ethereum smart contract, it'll be everywhere. This is exactly what motivates the use of cryptography in smart contracts. It lets us to perform computation while perserving the data's secrecy.

No access to the outside world A smart contract cannot reach out for external information from the internet (i.e. pull). Instead it is the users responsibility to supply the necessary information. Why? It is pretty simple, there is no guarantee that information pulled from the internet will persist forever. Without full information, new users cannot re-execute the relevant transactions in the blockchain (and thus it will stop new users re-creating the same copy of the database as everyone else).

Deterministic execution, but unpredictable execution As we have already mentioned, the execution of a smart contract depends on the order of transactions accepted into the blockchain. However if there are two or more transactions competing to get into the blockchain, then they may interfere with each other's computation. We'll come back to this issue during Week 5.

Event-driven execution Smart contracts 'do not run in the background' and instead they will only perform execution if the network accepts a relevant transaction. As we'll see in a future tutorial, if you want a smart contract to send coins after time t , then the user must send a transaction to the network after time t to withdraw their coins. The smart contract not send coins after time t by itself.

2 Solidity Basics

Solidity is the dominant high-level programming language for writing smart contracts in Ethereum (and several other cryptocurrencies). We'll cover some of the basics here, but there is plenty of online material to help you.

2.1 Declaring a smart contract

Before we start writing a smart contract, we need to give it a name and specify how the code should be compiled. All code is wrapped inside the `contract` declaration and `pragma solidity` states which Solidity compiler to use. We provide an example in Figure 1, and omit it from all future examples.

```
Outline of a smart contract

pragma solidity ^0.5.2
contract contractName {
    // Variables and Structs
    // Modifiers
    // Events
    // Function declarations
}
```

Fig. 1: How to declare a smart contract in Solidity

2.2 Functions (and visibility)

Solidity is an ever-changing language, and it is being developed to encourage explicitness from the developer. This is in response to several famous smart contract hacks due to ‘implicit’ bugs in recent years. We’ll cover some of the explicitness you’ll need to consider when defining a function.

Types of functions There are three types of functions in Solidity:

- **constructor** (param1, param2, ...) is responsible for initialising the contract and it will only be executed when the smart contract is instantiated on the network. (i.e. it is only executed when the code is deployed).³
- **function** {} is the *fallback* function. This function has no name and does not return any data. It is only executed if the transaction doesn’t specify a function or supply any data.
- **function** name(param1,...,param2,...) is a typical user-defined function.

As we’ll discover in future tutorials, the fallback function can notoriously be used to launch attacks on other contracts. For the rest of this tutorial, we’ll mostly consider user-defined functions as this is what you’ll be using.

Update contract state The function must explicitly state its relationship with the smart contract’s storage (i.e. the contract’s state):

- **pure** functions cannot modify or read the state.
- **view** functions can only to read the state, but never modify it.
- *No keyword.* Not defined as pure or view, it can modify and read the state.

Function Visibility The smart contract developer can decide who (and what) can call a function. There are three key words which we can also mostly re-use for the next section about variables:

- **public** the function is publicly and internally accessible.
- **private** the function can only be accessed by this contract. It is not visible to other users (or other contracts).
- **external** the function is publicly accessible, but this contract cannot internally access it.
- **internal** the function can only be accessed by this contract (or another contract that derives this contract’s functionality via inheritance). It is not visible to users (or other non-derived contracts).

Payable The keyword **payable** is required before a function can send or receive ether. Typically this is exclusively used for withdraw and deposit functions.

³ Fun Fact: In the past, the constructor was simply a function with the same name as the contract. Sometimes developers mis-spelt the function’s name and it would no longer be a constructor. This led to several broken contracts as anyone could call the constructor and re-initialise the contract. This motivated introducing the **constructor** keyword.

Return data All functions (except the constructor) can return one or more variables:

```
returns(< type1 >, < type2 >, ..., < typeN >)
```

The final line of code within a function should explicitly state which variables will be returned. Of course, it should go without saying that each variable must match the type defined in the function's definition:

```
return (variable1, variable2, ..., variableN).
```

Note the **data is not returned to the transaction caller**, it is only useful when this function is called by another function (or another contract). As we'll cover shortly, the keyword **event** is designed to notify external applications about events (or data) that has emerged due to execution within a smart contract.

Full function declaration Taken into account the above, the full abstract function definition:

```
function <parameter types> [internal,external,public]  
[pure,view,payable] returns(<return types>)
```

We provide some code examples for the various ways to declare a function in Figure 2.

```
Function Declarations  
  
uint256 public item1;  
uint256 public item2;  
address public owner;  
  
function getItem1() public view returns(uint256) {  
    return item1;  
}  
  
function getBothItems() public view returns(uint256, uint256) {  
    return(item1, item2)  
}  
  
function withdraw() public payable {  
    // Logic to send coins to a user  
}  
  
function verifySignature(bytes memory data, bytes32 signature,  
                        address signer) public pure returns(bool);
```

Fig. 2: How to declare various **function** in Solidity

2.3 Storage on the network

There are three forms of storage for a smart contract:

- **storage**. All contract variables reside in storage and the variables are persistent between function calls. It is expensive to store values in storage.
- **memory**. Short-term memory as the variables will only exist for this single function call. It is cheap to store values in memory as they are discarded after the function's execution is finished.
- **Stack**. This does not require a keyword, but it stores small local variables.

We'll come across **storage** and **memory** throughout the lab sessions. When defining a variable within a function, **memory** will store a copy of the value, whereas **storage** will store a pointer to the data. The code sample in Figure 3 should help you work out the difference.⁴

```
Memory and Storage

uint[] balance = [10];

function memoryBalance() public returns(uint) {
    uint[] memory bal = balance; // Copies value
    bal[0] = bal[0] + 1;
    return balance[0]; // Returns 10 as storage is not updated.
}

function storageBalance() public returns(uint) {
    uint[] storage bal = balance; // Stores pointer
    bal[0] = bal[0] + 1
    return balance[0]; // Returns 11 as storage is updated.
}
```

Fig. 3: The difference between **memory** and **storage**

2.4 Data Type and Structures

We'll briefly cover the basic data types and structures used in Ethereum (alongside some of their quirks). This is not an exhaustive list and it is ever-changing as Solidity is actively updated by the community. We recommend students to visit online resources for more information.

⁴ We haven't covered arrays yet. If the code sample doesn't make sense. Finish the tutorial and return to this later.

Value Types A value type is a ‘basic data type’ in Ethereum:

- Byte **byte** (alias **byte1**), **bytetwo**, ..., **bytes32**
- Unsigned integer **uint** (alias **uint256**) (i.e. positive whole numbers)
- Signed integer **int** (i.e. negative and positive whole numbers)
- Boolean **bool** (i.e. TRUE or FALSE)
- Ethereum account **address** (i.e. 20 bytes/hash of public key)
- String **string** (note: strictly not a basic value type, but an array)
- Bytes **bytes** (note: strictly note a basic value type, but an array)

Array (and multi-dimensional) An array is an ordered list of items which are indexed numerically. Solidity has three types of arrays (T is a value type and k is the length):

- Fixed size array T[k].
- Dynamic sized array T[].
- Multi-dimensional arrays T[][] [5] (dynamic array, and each sub-array has a fixed length of 5)

We need to consider constraints due to **memory** and **storage**. A **memory** array cannot be resized after it is created (i.e. it has a fixed `.length`). An array that will persist in the long-term using **storage** can be resized and new items appended using `T.push(item)`. We provide some code examples in Figure 4.

```
Dynamic and Fixed Arrays

address[] public whitelist; // Dynamic size array
// Do not use in practice, dangerous
// By the end of this course, you should know why
function updateWhitelist(address _user) public {
    whitelist.push(_user);
}
function deleteWhitelist() public {
    delete whitelist; // Deletes all entries, but can be re-used
}
// Two ways to declare memory arrays. Lets pretend k=2 for example.
function fixedSizeArray(address _user1, address _user2, uint256 _k) public {
    address[2] memory users;
    users[0] = _users1;
    users[1] = _users2;

    address[] memory usersAgain = new address[](_k);
    usersAgain[0] = _users1;
    usersAgain[1] = _users2;
}
```

Fig. 4: How to create, update and delete arrays.

Mappings A mapping links a set of keys to a set of values. It is declared as:

```
mapping(KeyType => ValueType)
```

The KeyType can be almost anything except for another mapping, dynamically sized arrays, a contract, an enum, and a struct. The ValueType can be anything including another mapping! One quirk in Solidity is that only a hash of the keyset is stored (and not the full list of keys). This prevents a smart contract easily fetching the full keyset and iterating over all key-value pairs. As well, a mapping cannot be deleted using the keyword `delete` and instead we must overwrite all values to 'delete'. We provide a code sample in Figure 5.

```
Mapping

mapping(address=> uint256) public balance;

function getBalance(address _user) public view returns(uint256) {
    return balance[_user];
}

function updateBalance(address _user, uint256 _balance) public {
    balance[_user] = _balance;
}
```

Fig. 5: How to create and use a **mapping** to store each user's balance

Enum The keyword **enum** let us provide human-readable names for a set of constants. A common use-case in real-world smart contracts is to specify its current state/flow. For example, is the smart contract turned on and ready to accept deposits from users? We provide a code sample in Figure 6.

```
Enum

enum Flag {ON, OFF}
Flag public flag;

function turnOn() public {
    flag = Flag.ON;
}

function turnOff() public {
    flag = Flag.OFF;
}
```

Fig. 6: How to create and store an **enum**

Struct A **struct** lets us define a human-readable name for a group of variables. Unlike typical object-oriented programming, a **struct** does not provide any functionality. We can treat a **struct** like a value type such that they can be used within arrays and mappings. As well, a **struct** can contain arrays and mappings too. We provide a code example in Figure 7.

```
Struct

struct Player {
    address addr;
    uint attack;
    uint defense;
    uint magic;
    string tagline;
}

Player[] characters;

function newCharacter(address _addr, string memory _tagline) public {
    // Creating an instance of the Player Struct
    Player memory character = Player(_addr, 0, 0, 0, _tagline)
    characters.push(character);
}

function getTagline(address _addr) public view returns(string memory) {
    for(uint i=0; i<characters.length; i++)
        // Accessing the 'addr' field in the struct
        if(characters[i].addr == _addr) {
            return characters[i].tagline;
        }
    return "";
}
```

Fig. 7: How to create and store an **struct**

Events This is a new idea introduced in Ethereum to let smart contracts communicate with the outside world. An **event** defines the information that will be published, and the keyword **emit** will publish the event during a function's execution. Applications watching the smart contract (and the blockchain) can perform tasks when they are notified about the **event**. For example, a casino operator's machine may wait for the smart contract to signal that a player has finished their move via an **event**. In response, the casino operator may broadcast a transaction to make their move in the game (i.e. roll the die, send the player their winnings, etc). We provide a code example in Figure 8.

```

Event

event Move(address Player, uint choice);

function playersChoice(uint _choice) public {
    emit Move(msg.sender, _choice);
}

```

Fig. 8: How to create and store an **event**

2.5 Control Logic

If statement Like other languages, the keywords are **if** and **else**. A code sample for Solidity is provided in Figure 9.

```

If Statements

function largestNumber(uint256 _a, uint256 _b) public pure
    returns(uint256) {
    if(_a > _b) { return(_a); } else { return(_b); }
}

```

Fig. 9: How to use an **if** statement to return the largest number

Loops Like other languages, Solidity supports **for** and **while** loops which we present in Figure 10.

```

Checking whitelist using loops

function inWhitelistForLoop(address _user) public view returns(bool) {
    for(uint256 i=0; i<whitelist.length; i++)
        if(whitelist[i] == _user) return TRUE;
    }
    return FALSE;
}

function inWhitelistDoWhile(address _user) public view returns(bool) {
    uint256 i = 0;
    do {
        if(whitelist[i] == _user) return TRUE;
        i=i+1;
    } while(i<whitelist.length);
    return FALSE;
}

```

Fig. 10: How to use **for** and **while** loops

3 Exercise: Global and Censorship-resistant Hello World

For the next few weeks, we'll be using an online tool, Remix⁵, to write and test our smart contracts. It lets us play with solidity (and walk through our code step-by-step) without any complicated set up. All smart contract execution in Remix is performed in a Javascript implementation of the Ethereum Virtual Machine (EVM) and not the live network. Thus it is free, fast and good for beginners like you!

Why Hello World? While it is nice for a new environment (and soon-to-be-friend) to digitally greet us, this exercise lets us sanity check the programming environment is *actually working*. After this tutorial, we encourage you to try and deploy your smart contract to Ethereum's test network. Once deployed, it'll exist forever and no single party (including nation states) can interfere or stop its execution. How cool is that?

Playing with Remix The first time Remix is opened, you'll find a simple ballot smart contract. Let's take this opportunity to play with it and come to grips with how Remix works.

- Study the code and try to understand how the ballot program works. (Make notes on this tutorial sheet!)
- In the *Compile* tab, click 'Start to Compile'.
- In the *Run* tab, click 'deploy'.
- In the *Run* tab, find 'Deployed Contracts' and your new ballot program. (Hint: You may need to click 'Ballot at 0x..... (memory)').
- Play with the functions and use the debug log. Can you understand the execution trace?
- Try to change the Ethereum account that is signing and sending the transaction. (Hint: Look for the dropdown box 'Account')

If the above is not clear, please chat with a teaching assistant who will go through it with you.

Building HelloWorld Let's get started with building our censorship-resistant Hello World smart contract. Try to complete the following tasks:

- Create a new file for the smart contract (or simply delete the ballot contract)
- Create the smart contract declaration and let's call it *HelloWorld*.
- Create the function `hello()` that returns the **string** "Hello world".
- Try calling the function and show a teaching assistant the message via the debug log.

As we mentioned earlier in the tutorial, returning data via a function call is not always visible to the transaction caller. It is only useful when it is invoked by another function (or contract). Let's update the function to emit an Event, so anyone can watching the contract can find the signal.

⁵ <https://remix.ethereum.org/>

- Create a new Message **event** that publishes the string.
- Update hello() to **emit** the new Message **event**.
- Try calling the function and show a teaching assistant the emitted event via the debug log.

Good job! Now developers can watch your smart contract and wait for the hello signal. Let's make it a bit more interesting, it would be nice if anyone can update the message emitted by your smart contract.

- Create a new function, updateMessage(**string** msg), that lets anyone supply a string that will later be emitted as an event using hello(). (Hint: You will need to store the message!)
- Try calling the function multiple times. Does the gas usage change? Explain the answer to a teaching assistant.

While the list of message updates is stored on the blockchain, this is not currently easily accessible within the contract environment. Let's update the contract to keep a list of all messages sent to the smart contract.

- You'll need to create an array of strings, and modify updateMessage() to push every new message to the list.
- Chat with a teaching assistant (or your friend) to check it is working.

Finally, it would be great to keep a record of the accounts who have previously updated the message. We need to introduce a new key word, **msg.sender**, which is the address (i.e. Ethereum account) of the function's immediate caller.

- Define a **mapping** that links an **address** to a **string[]**.
- Modify the updateMessage() function to push every new message to the caller's string array.
- Create a new function getMessage(**address** user, uint i) that returns the string at index i for a given user.
- Create a function latestMessage() that returns the latest message and the address that submitted it.
- Chat with a teaching assistant (or your friend) to check it is working.

Good job! That is all for today. Try to go over this tutorial again in your own time, especially if you didn't finish. There are no marks for completing the tutorial, but it'll help towards your assessed coursework!