Exercise 7

1. What is the difference between an abstract contract, an interface, and a library in Solidity?
2. Are these statements true or false? Give a short explanation.
   (a) Every pure function is a view function.
   (b) A transaction that calls a view function does not cost any gas because it does not change state.
   (c) In Ethereum, a smart contract that has no functions that are declared `payable` cannot receive any Ether.
   (d) Solidity's standard `int` and `uint` datatypes can each hold 2^{32} different values.
   (e) If `msg.sender == tx.origin`, then this code was called directly by an externally owned account and not by another 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?
   (b) What are the three largest Ethereum test networks and what are their differences?
   (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.
   (d) Write a Solidity program with a function that returns "Hello, world!". Use a compiler version $\geq$ 0.5.1.
   (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?
   (f) What is the difference between the `--bin` and `--bin-runtime` compiler options? Which should you use to deploy your contract?
   (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?
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:

```
pragma solidity ^0.4.24;

contract Dangerous {

    mapping(address => uint) deposits;

    function depositMoney() public payable {
        deposits[msg.sender] += msg.value;
    }

    function withdraw(uint amount) public {
        require(deposits[msg.sender] >= amount);
        if (!msg.sender.call.value(amount)()){
            revert();
        }
        deposits[msg.sender] -= amount;
    }
}
```

   (a) Describe the basic functionality of this contract.

(b) What exactly happens in line 13?

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

(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.

(e) How could this attack be prevented?