Exercise 1

Please consider the numbered lines of code:

```
 1 pragma solidity 0.4.21;
 2
 3 contract PiggyBank {
 4     address owner;
 5     uint248 balance;
 6     bytes32 hashedPassword;
 7
 8     function piggyBank(bytes32 _hashedPassword) {
 9        owner = msg.sender;
10        balance += uint248(msg.value);
11        hashedPassword = _hashedPassword;
12    }
13
14    function () payable {
15        if (msg.sender != owner) revert();
16        balance += uint248(msg.value);
17    }
18
19    function kill(bytes32 password) {
20        if (keccak256(owner, password) != hashedPassword) revert();
21        selfdestruct(owner);
22    }
23 }
```

The following observations can be made:

1. At line 5 `balance` is stored and managed as a `uint248` variable. But `msg.value` is implicitly defined as a `uint256`. So sending a very big amount to the fallback function for example will result in an overflow
2. Usage of the `+=` operator at lines 10 and 16 *in combination with a situation where an overflow could arise*. A check should be made to see if the result is greater than original. If so, then it's ok, if not, it should be reverted. Or just use "SafeMath" library which does the management for you
3. `msg.value` is used in the function `piggyBank` at line number 10, but the function is not payable. So there is no way this function could work. Calling it will result in an error and the transaction would be reverted
4. Given the fact that a smart contract is created by its owner (at least the initial owner, ownership could be transferred afterwards), the function `piggyBank` is supposed to be a constructor, so the first letter should be upper case. Now it's not a constructor, just a normal function. Now, anybody who calls the function with some small value, could claim ownership, and afterwards self-destruct the contract and send the accumulated balance in the contract to its own address. So the constructor will set the owner and allows money to be sent to the contract. After, the owner can use the fallback method incrementally to send more money
5. Fallback functions should be simple. If too many things happen it's better to use a separate function
6. Line 20 makes sure that only people who know the password of the owner can kill the contract. That should be the owner. The owner could call this function with the clear text password and the contract hashes the password with the address of the owner to see if the stored hashed password is the same. But this approach is susceptible to *front-running*: A hacker could see the sent transaction (by the owner), and decide to replay the same transaction, with the same password (which is publicly available on the blockchain). By specifying a high gas price, he could front-run the transaction that was sent by the owner. Miners might process the transaction sent by the hacker sooner than the owner's transaction. Apart from this, it would be just easier to use the same method of ownership checking as is used in the fallback function. Even better would be to create a separate

smart contract “Owned” that would abstract the owner variable and also create functions and modifiers that can be used in order to manage ownership

7. The variables `owner`, `balance` and `hashedPassword` should be made public. Everything is available for everybody on the blockchain anyway, making these variables public will provide an interface to get access to these values more easily.
8. The `kill` function can be tagged as public, which is the default setting, but will make the warning disappear

Please check zip file that contains a directory with an improved version of this contract.

Exercise 2

Please consider the numbered lines of code:

```
 1 pragma solidity ^0.4.5;
 2
 3 interface WarehouseI {
 4    function setDeliveryAddress(string where);
 5    function ship(uint id, address customer) returns (bool handled);
 6 }
 7
 8 contract Store {
 9    address wallet;
10    WarehouseI warehouse;
11
12    function Store(address _wallet, address _warehouse) {
13        wallet = _wallet;
14        warehouse = WarehouseI(_warehouse);
15    }
16
17    function purchase(uint id) returns (bool success) {
18        wallet.send(msg.value);
19        return warehouse.ship(id, msg.sender);
20    }
21 }
```

The following observations can be made:

1. Line 1 should be changed to: `pragma solidity ^0.4.11`. The keyword “interface” was introduced in that version. So unless you use a compiler 0.4.11 version or higher, Solidity will complain with a compilation error
2. `msg.value` is used at line number 18. But the function is not payable. So this will result in an error. In the following issues, it is assumed that the `purchase` function has been tagged with the `payable` clause.
3. If `wallet` is an EAO, the `send` call at line 18 could result in an out-of-gas situation, which doesn’t result in a revert, but causes the `send` call to return false. It could also be that wallet is a smart contract, but this cannot be decided 100% upon, judging the code. In case `wallet` is a smart contract, when doing the `send` at line 18, in case this wallet contract has a fallback function, if a `revert` is done in that fallback function, it will not throw a failure (revert the transaction), it will return false. In both cases, the `purchase` function does not check the return value of the send call and eventually ship the product (without payment). Basically, when using `send`, the return value should always be checked.
4. The checking of the value of the good against `msg.value` is done in the `ship` method: Design mistake. This can be deduced the following way:
    a. The name of the parameter `customer` of the ship function at line 5 coincides with `msg.sender` at line 19
    b. Using proof by contradiction: If the purchase method would not have been called by the customer EAO directly, but by another smart contract, then `msg.sender` is not preserved, and as a result, the

address of that smart contract would be passed. This contradicts the with the name of the customer parameter at line 5

So the `purchase` method is called directly by the customer EAO. This means that the only place where the value of the good that is being purchased by the customer, is checked against the value that is sent, `msg.value`, is the `ship` function. This is a design error in my opinion (not a programming error). The shipping function should not be responsible for making this check, this responsibility belongs to the `purchase` function. Given the fact that the return value of the `ship` function is not checked, we can also deduce that the `ship` function must do a revert in case of error, because returning false in that case would result in the `send` call at line 18 not to be reverted, but that is just an additional observation

5. It's a best practice to make the transfer or ether the last statement in a function, so the call to `ship` and `send` should be switched
6. The function purchase should check the parameters `_wallet` and `_warehouse`. These addresses should not be 0x0
7. The variables `wallet` and `warehouse` should be made public. Everything is available for everybody on the blockchain anyway, making these variables public will provide an interface to get access to these values more easily.
8. The constructor `Store` and the function `purchase` should be marked as public to make the warning disappear
9. Not an error, but functions in interfaces should be declared as "external", as they are not intended to be used from within the implementation

Please check zip file that contains a directory with an improved version of this contract.

Exercise 3

Please consider the numbered lines of code:

```
1  pragma solidity ^0.4.9
2
3  contract Splitter {
4     address one;
5     address two;
6
7     function Splitter(address _two) {
8         if (msg.value > 0) revert();
9         one = msg.sender;
10        two = _two;
11    }
12
13    function () payable {
14        uint amount = address(this).balance / 3;
15        require(one.call.value(amount)());
16        require(two.call.value(amount)());
17    }
18 }
```

The following observations can be made:

1. The minimum solidity version is 0.4.9. The 0.4.9 compiler doesn't support the `revert` instruction at line number 8. This should be replaced with a `throw` instruction
2. Same is true for the `require` instructions at lines 15 and 16. These should be replaced with `if … throw`
3. `msg.value` is used at line number 8, whilst the constructor is not payable, which will result in a compiler error
4. There is a design mistake. The creator of the Splitter contract, the contract owner, is address `one`. Address `one` is for one-third beneficiary of the split. Address `two` is also for one-third beneficiary of the split. The Splitter contract itself is for the last one-third beneficiary of the split. But the ether that is left in the Splitter contract after the split, cannot be recovered. There should have been a withdrawal or kill method that allows the owner (address `one`) to withdraw the ether. In the case that there were a withdrawal or kill method, address `one` would eventually have two-thirds of the split. If this is the functionality… ok… but it seems weird
5. The total amount is divided by three at line number 14. So the Splitter contract might end up with more ether due to the truncating behavior of the integer division
6. The act of splitting should not be done in the fallback function. Fallback methods should be kept simple. A separate function should be created to split the funds
7. As said, a separate function should be made to split the funds. Also this function should not send the funds to untrusted accounts. Instead, a withdrawal function should be made in order to comply with the withdrawal pattern in order to prevent reentrance attacks. The beneficiaries can call the withdraw function to withdraw their parts
8. With the current state of the code, a reentrance-attack could be executed. This will be explained in the next part

The reentrance-attack explained

First, consider the following version that will compile without errors (compiler version 0.4.10 is used):

```
1  pragma solidity ^0.4.10
2
3  contract Splitter {
4     address one;
5     address two;
6
7     function Splitter(address _two) payable {
8         if (msg.value > 0) revert();
9         one = msg.sender;
10        two = _two;
11    }
12
13    function () payable {
14        uint amount = address(this).balance / 3;
15        require(one.call.value(amount)());
16        require(two.call.value(amount)());
17    }
18 }
```

There are two possible scenarios, one scenario where address one is a malicious contract and the other scenario where address two is the malicious contract.

Scenario 1

A possible reentrance attack could be successfully played the following way. In this scenario, address `two` is the malicious contract

1. The Splitter contract is created and the parameter `two` to the constructor is the malicious contract
2. Somebody wants to split money over the Splitter contract, address `one` and address `two` by sending ether to the fallback function of the Splitter contract
3. The Splitter contract sends 1/3 of the total amount of ether to address `one`
4. The Splitter contract sends 1/3 of the total amount of ether to address `two`
5. Address `two` is not an EAO, but a malicious contract with a fallback function
6. The fallback function of contract `two` sends 1 wei to the the Splitter contract again
7. The fallback function of the Splitter contract has 1/3 (+1 wei) of the total amount left and sends 1/3 of that amount to address `one`
8. The fallback function of Splitter contract also sends 1/3 of that amount to the malicious contract `two`.
9. The fallback function invokes the fallback function of the Splitter contract again (go to 6.), until the balance of the Splitter contract is close to 0 or x invocations have been performed
10. At the end, Splitter contract is near to 0, address `one` has 1/2 of the total amount and the malicious contract `two` has the other half of the total amount that was sent initially
11. This is a clear hack, where the victim is the Splitter contract (deprived of all the money). Address `one` receives (1/2 - 1/3) more than intended. The same is true for the malicious contract `two`. It must be noted that address one is the owner of the contract.

The zip file contains the contract source file `HackerTwo.sol` which contains the code of the malicious contract. The test case `splitter.js` contains the test case “Address two is a malicious contract” which proves the above deduction.

Scenario 2

A possible reentrance attack could be successfully played the following way. In this scenario, address `one` is the malicious contract

1. A malicious contract (address one) creates the Splitter contract and specifies another address as the co-beneficiary
2. Some EAO or contract sends ether to the Splitter fallback function and *expects that the money will be evenly split the Splitter contract, address* `one` *and address* `two`
3. When the Splitter contract sends ether to address `one`, in the fallback function of contract `one`, the Splitter fallback function is called again *with the same amount that was sent to contract* `one`
4. So the Splitter fallback function is invoked again. It sends ether again to address `one`. But this time, (the second time) it doesn't invoke again the Splitter contract (only the first time). The final amount that is present now in contract `one` is 1/3 of the total amount that was send to be split
5. Then the Splitter contract sends 1/3 to address `two` (still in the recursion that was caused by contract `one`). The balance of address `two` is now 1/3 of the total amount sent
6. The recursion that was caused by contract `one` finishes and the second invocation to the send() function to address `two` is done, and again receives 1/3 of the total amount. So the final amount that was sent to address `two` is 2/3 of the total amount
7. The Splitter contract's balance will by 0

So contract `one` is used as intermediary account to invoke the Splitter contract again and send the money back to the Splitter contract in order to not *cause a rollback*, because in the case of a rollback the whole transaction will be rolled back and the hack doesn't work.

Contract `one` and EAO (or contract) `two` are obviously friends in this scenario. So in the end they end up with all the money, 2/3 for address `two` and 1/3 for contract `one`.

The zip file contains the contract source file `HackerOne.sol` which contains the code of the malicious contract. The test case `splitter.js` contains the test case "Address one is a malicious contract" which proves the above deduction.