Exercise 1:

- **Need to add the complier version**
- **Constructor needs keccak256() function included for the password/owner**
- **Constructor needs payable added for msg.value**
- **Fallback functions need to be kept simple. Shouldn't be payable**
- **Needs an LogDepositReceived event log created**
- **Needs a setDepositValue() function created so you the Owner can deposits new funds. Only accessed by the Owner. Log the Deposit. Make payable**
- **Needs a getDepositValue () function created to get the balance**
- **Create a Modifier for functions to assure the msg.sender = owner**
- **Create a Modifier for functions to assure the msg.value != 0**

New Contract:

```
pragma solidity ^0.4.6;

contract PiggyBankNew {
    address owner;
    bytes32 hashedPassword;
    uint248 balance;

    modifier onlyOwner () {
        assert(msg.sender==owner);
        _;
    }

    modifier hasValue () {
        assert(msg.value!=0);
        _;
    }

    event LogDepositReceived(address _sender, uint248 _value);

    function PiggyBankNew (bytes32 _hashedPassword) hasValue payable public {
        owner = msg.sender;
        balance += uint248(msg.value);
        hashedPassword = keccak256(msg.sender, _hashedPassword);
        LogDepositReceived(msg.sender, balance);
    }

    function setDepositValue(bytes32 _hashedPassword) onlyOwner hasValue payable public returns(bool success) {
        if (keccak256(owner, _hashedPassword) != hashedPassword) revert();
        balance += uint248(msg.value);
        LogDepositReceived(msg.sender, balance);
        return true;
    }

    function getDepositBalance(bytes32 _hashedPassword) constant public returns(uint248) {
        if (keccak256(owner, _hashedPassword) != hashedPassword) revert();
        return balance;
    }

    function kill(bytes32 _hashedPassword) onlyOwner public {
        if (keccak256(msg.sender, _hashedPassword) != hashedPassword) revert();
        selfdestruct(owner);
    }

    function () public {revert(); }

}
```

Exercise 2:

- **WarehouseI is an Interface and is not constructed correctly**
    - **setDeliveryAddress() needs a return statement**
    - **ship() needs to be private so everyone can't call the function**
    - **Needs event log created to know what needs to be shipped**
- **Store needs to inherit WarehouseI**
    - **Store needs an event Log so the Store knows what was purchased (what, who, how much)**
    - **Store function purchase(), need to make the function Payable**
    - **Store function purchase(), need to change "wallet.send" to "wallet.transfer"**
    - **Mapping needs to be created for a CustomerStruct to get track of the customer address**
    - **Needs a Modifier to assure the msg.value != 0**
    - **Purchase() function, changed "wallet" to msg.sender so anyone can buy a product at the Store**

<u>New Contracts</u>

```
pragma solidity ^0.4.6;

contract WarehouseI {

    event LogShipReceived(uint logId, address logCustomer, string logAddress);

    function setDeliveryAddress(string _customerAddress) public returns (bool handled);

    function ship(uint id, address customer) private returns (bool handled);

}

contract Store is WarehouseI {
    address wallet;

    struct CustomerStruct {
        string customerAddress;
    }

    mapping (address => CustomerStruct) customerStructs;

    modifier hasValue () {
        assert(msg.value!=0);
        _;
    }

    event LogPurchased(uint logId, address logCustomer, uint logValue);


    function Store(address _wallet) public {
        wallet = _wallet;
    }

    function ship(uint id, address customer) private returns (bool handled)
    {
        LogShipReceived(id, customer, customerStructs[customer].customerAddress);
        return true;
    }

    function setDeliveryAddress(string _customerAddress) public returns (bool handled){
        customerStructs[msg.sender].customerAddress = _customerAddress;
        return true;
    }

    function purchase(uint id) hasValue payable public returns (bool success) {
        wallet.transfer(msg.value);
        ship(id, msg.sender);
        LogPurchased(id, msg.sender, msg.value);
        return true;
    }
}
```

Exercise 3:

- **Splitter Constructor: Need to change if (msg.value > 0) throw;   to assert(msg.value > 0)**
- **Splitter Constructor: Needs Payable attach so a sender can send value to the contract**
- **Fallback() function needs Payable removed**
- **Attacker could create a Loop that keeps calling the Splitters Fallback function, until the Contract balance is drained.**

How the Attacker Could Attack the Contract

```
pragma solidity ^0.4.6;

contract SplitterNew {
    address one;
    address two;

    function SplitterNew(address _two) payable {
        assert(msg.value > 0);
        one = msg.sender;
        two = _two;
    }

    function () {
        uint amount = this.balance / 3;
        if (!one.call.value(amount)()) revert();
        if (!two.call.value(amount)()) revert();
    }
}

contract Attacker {

    SplitterNew v;

    event LogFallback(uint count1, uint balance);

    function Attacker(address victim) payable {
        v = SplitterNew(victim);
    }

    function attack() {
        for(uint i=0; i<30; i++){ // Denial of Service Attack - Loops
            v.call();
            LogFallback(i, v.balance);
        }
    }
}
```

Exercise 3 (How to Fix the Contract):

- **Splitter: Create new withdraw() function that includes a Boolean variable to track if the contract has already paid or not.**
- **Splitter withdraw() function: Make Private**
- **Splitter withdraw() function: change .send() to .transfer()**
- **Splitter Fallback: delete code**

New Contract

```
pragma solidity ^0.4.6;

contract SplitterNew {
    address one;
    address two;
    bool alreadyPaid;

    function SplitterNew(address _two) payable {
        assert(msg.value > 0);
        one = msg.sender;
        two = _two;
    }

    function withdraw() private returns(bool success) {
        if (!alreadyPaid) {
            alreadyPaid = true;
            uint amount = this.balance / 3;
            one.transfer(amount);
            two.transfer(amount);
            return true;
        }else
        {
            return false;
        }
    }

    function () public {revert(); }
}
```