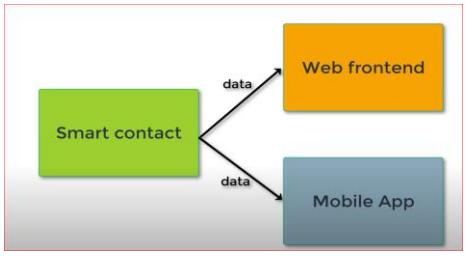
Solidity Test

1. Event and log record

1. What's the event and it used for?

In solidity, Events are dispatched signals which are fired by smart contract and can be subscribed by frontend, mobile app, DAPP etc. to perform action accordingly.



Characteristics & limitation of events:

- The event logs are used to debug application, detect specific events, and notify viewer of event logs that something has happened.
- The event data, *which is in the form of (or wrapper of) transaction logs*, stays in blockchain.
- Events are less expensive to use (a cheaper form of storage)
- Event parameters can be `indexed` and not indexed. The indexed parameters provide an efficient way of filtering events. (maximum 3 indexed parameters per event are allowed)
- The contract (or EVM) cannot read their own logs or the logs of other contracts.

Example:

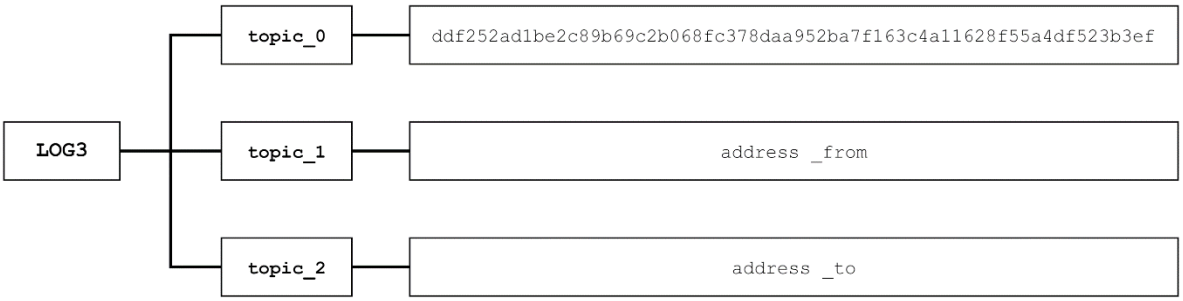
```
// Declaration of event
event Transfer(address indexed from, address indexed to, uint256 value);

function transfer(address to, uint tokens) public returns (bool success)
{
    // some logic of transferring fund

    // triggering event
    emit Transfer(msg.sender, to, tokens);
    return true;
}
```

2. What parts does the log record have to store?

Events are built on top of a lower level `log record` in EVM. Each log record of transaction receipt has two portions, **`topics` and `data`,** for keeping parameters.

Topic:

- Topics are 32 bytes long and are used to describe event signature and indexed parameters.
- Log structure can have up to 4 topics represented by a specific opcode (LOG0, LOG1…LOG4). e.g.
  - LOG0 includes no topic (anonymous event).
  - LOG1 includes one topic while LOG4 includes four topics.
- The first topic is used to store the hash of the event's signature: In above example : *keccak-256-hash(transfer(address,address,uint256))*
- Remaining three topics are used for up to three indexed parameters.
- In above `transfer` example, we have 2 indexed parameters so total topics would be 3 including 1st topic as hash of event signature (represented as LOG3).



Data:

- Data is the second part of a log record consists of additional data which has no size limit.
- Non-indexed parameters are always stored in the data.
- Since topics can only hold a maximum of 32 bytes of data, as a best practice, things like arrays or strings should be included as data.
- Below is the transaction receipt log of `transfer` example

```
{
  "blockHash": "0x8243343df08b9751f5ca0c5f8c9c0460d8a9b6351066fae0acbd4d3e776de8bb",
  "blockNumber": "0x429d3b",
  "contractAddress": null,
  "cumulativeGasUsed": "0x64b559",
  "gasUsed": "0xcaac",
  "logs": [
    {
      "address": "0xb59f67a8bff5d8cd03f6ac17265c550ed8f33907",
      "blockHash": "0x8243343df08b9751f5ca0c5f8c9c0460d8a9b6351066fae0acbd4d3e776de8bb",
      "blockNumber": "0x429d3b",
      "data": "0x0000000000000000000000000000000000000000000000000000012a05f200",
      "logIndex": "0x56",
      "topics": [
        "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
        "0x000000000000000000000000b46c2526e227482e2ebb8f4c69e4674d262e75",
        "0x000000000000000000000000054a2d42a40f51259dedd1978f6c118a0f0eff078"
      ],
      "transactionHash": "0xab059a62e22e230fe0f56d8555340a29b2e9532360368f810595453f6fdd213b",
      "transactionIndex": "0xac",
      "transactionLogIndex": "0x0",
      "type": "mined"
    }
  ],
  "logsBloom":
"0x00000000040000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000040000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000200000000000000000000000000100000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000001010000000000000000000000000000000400000000000020000000000000000000000000000000000000000000000000000000000000
00000",
  "root": "0x3ccba97c7fcc7e1636ce2d44be1a806a8999df26eab80a928205714a878d5114",
  "status": null,
  "transactionHash": "0xab059a62e22e230fe0f56d8555340a29b2e9532360368f810595453f6fdd213b",
  "transactionIndex": "0xac"
}
```


2. Where are the variables of different types stored?

Each variable declared and used within a contract has a data location. Generally, EVM has four locations where it can store items.

1. Storage
2. Memory
3. Calldata
4. Stack

Below are the explanation of each store type.

Storage:

This is global memory available to all functions within the contract.

- The `storage` is a permanent storage and data in the `storage` are written in the blockchain.
- State variables are always stored in `storage`.
- Every contract has its own storage and it is persistent between function calls and quite expensive to use.

Memory:

This is temporary memory available to every function within a contract.

- The life time of `memory` is limited to a function call and is meant to be used to temporarily store variables and their values.
- It can be used for both function declaration parameters as well as within the function logic.
- It is mutable (it can be overwritten and changed)
- It is non-persistent (data are erased after function execution)

Calldata:

This is where all incoming function execution data, including function arguments, is stored.

- `calldata` is very similar to memory but it is a special data location only available for `external` function call parameters.
- It can only be used for function declaration parameters (and not function logic)
- It is **immutable** (it can't be overwritten and changed)
- It must be used for dynamic parameters of an external function
- It is non-persistent (data are erased after function execution)

Stack:

EVM maintains a stack for loading variables and intermediate values

- Whenever there is a need to perform some computation (addition, multiplication.. etc.) , the variables are moved from memory/storage to stack and the result is put back to memory/storage
- Its used to hold small local variables of value type (bool, integer, address etc.)
- It is almost free to use, but can only hold a limited amount of values

3. Call and DelegateCal

1. What are Call and DelegateCall?

2. What’s the relationship between them?

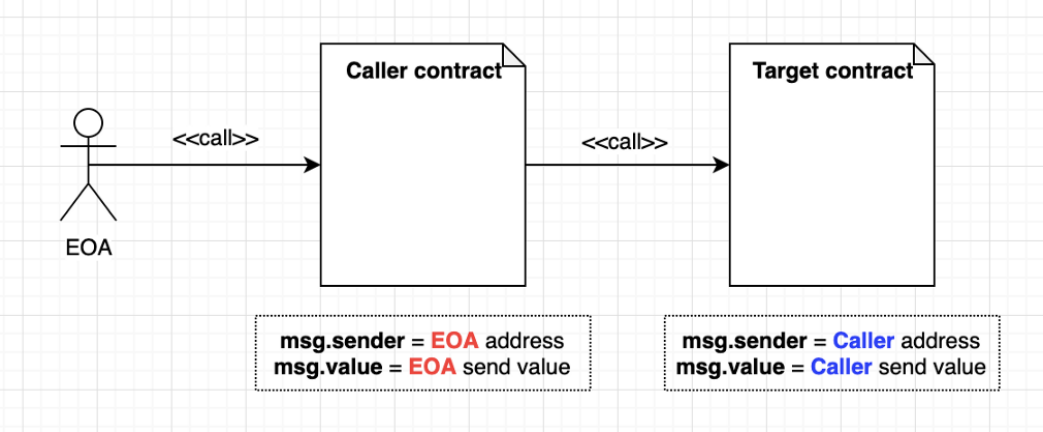
`call` and `delegatecall` both are low level functions and both return bool value.

In `call` the context is changed whereas in `delegatecall` the context does not change.

It means, when a logic contract is executed using `call`, the logic contract is executed in its own context and all the changes are retained with-in the logic contract itself. Whereas, in case of `delegatecall`, calling contract pulls code of logic contract into its context and changes are reflected in the context of the calling contract.

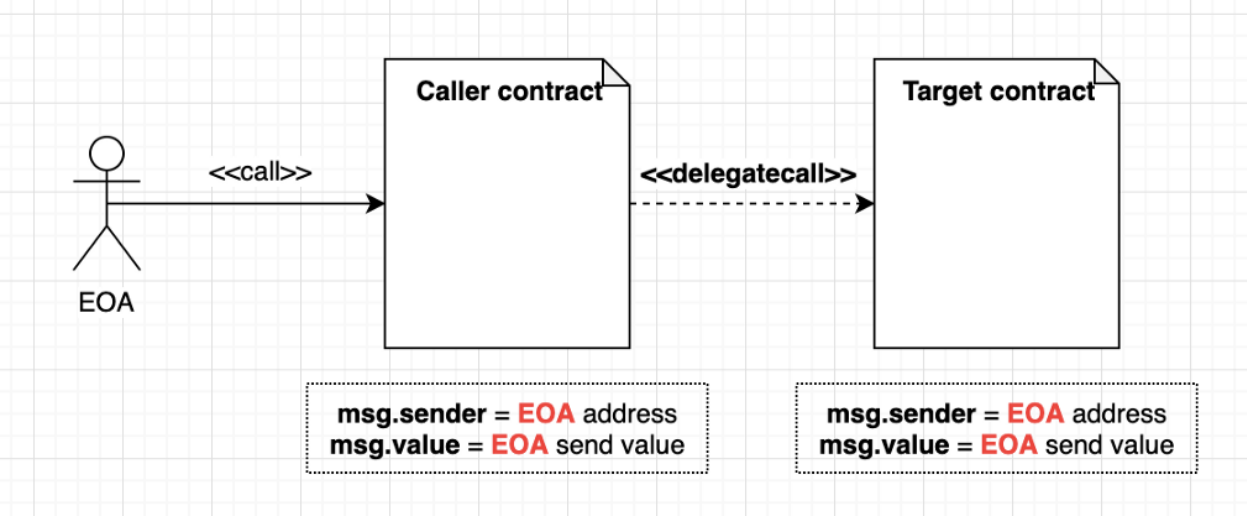
Call:

When an account `EOA` invokes Caller contact(A) and A does `call` on Target Contract (B), then `*msg.sender*` and `*msg.value*` are A's address and either value respectively, because B uses its own context to execute the code.



DelegateCall:

When an account `EOA` invokes Caller contact(A) and A does `delegatecall` on Target Contract (B), then `*msg.sender*` and `*msg.value*` are EOA's address and either value respectively, because code of B is loaded inside A's context.



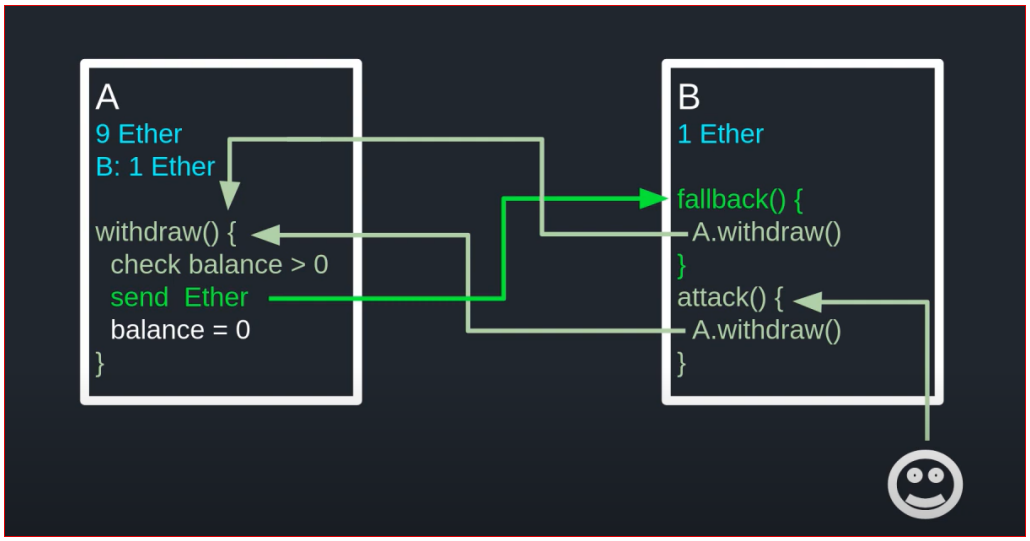

```
contract TargetContract {
    event Log(address from, uint token);
    function callMe() payable public {
        Log(msg.sender, msg.value);
    }
}


contract CallerContract {
    function callTargetContract(address _contactAdd) public {
        require(_contactAdd.call(bytes4(keccak256("callMe()"))));
        require(_contactAdd.delegatecall(bytes4(keccak256("callMe()"))));
    }
}
```

4. What's the Reentrancy attack/recursive call and how to avoid it?

A reentrancy attack can occur when a contract (A) makes an external call to another untrusted contract (B) to transfer ether. If the attackers can control the untrusted contract (B), they can make a recursive call back to the original contract (A) repeatedly before the invocation is completed.

For example, in below screenshot, this is how reentrancy attack can occur:

1. Initially, contract (A) has 1 ether of contract (B) and total ether is 10 (= 1 ether of B + 9 ether of others)
2. The attacker of B asks for withdrawal of his 1 ether by calling its some function attack() which invokes the A's withdraw() function.
3. A's withdraw() function checks B's balance (which is 1 ether) and sends 1 ether by triggering B's fallback function.
4. B's fallback function again calls A's withdraw() function in the middle of previous invocation .
5. A's withdraw() function again checks B's balance (which is still 1 ether because previous invocation has not yet completed and balance was not set to 0) and sends 1 more ether by triggering B's fallback function.
6. B's fallback function again calls A's withdraw() function repeatedly until total balance (10 ether) of A becomes zero.



There are few ways to prevent reentrancy attack:

- By updating the state variable before making any external calls to other contracts.

```
contract A {
// Old implementation
    function withdraw() external {
        uint256 amount = balances[msg.sender];
       require(msg.sender.call.value(amount)());
       balances[msg.sender] = 0;
}}
```

```
contract A {
// New implementation
    function withdraw() external {
        uint256 amount = balances[msg.sender];
        balances[msg.sender] = 0;
        require(msg.sender.call.value(amount)());
}}
```

- Use function modifiers that prevent re-entrancy

```
contract A {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
}

function withdraw() external noReentrant {
    uint256 amount = balances[msg.sender];
    balances[msg.sender] = 0;
    require(msg.sender.call.value(amount)());
}
```

5. What's the relationship and difference between Bytes and String? What are they used for respectively? When to use Bytes instead of String and String instead of Bytes?

Both `bytes` & `string` are dynamic array types which means that they can store data of any arbitrary size.
Each element of a variable of type `bytes` is a single byte whereas each element of a variable of type `string` is a UTF-8 character (some UTF-8 characters might be more than 1 byte) so there is not an immediate direct relation between each `bytes` index and the corresponding `string` index.
Both variable length types `string` and `bytes` can not be passed between contracts.

Below are more details and differences between `bytes` & `string`:

Bytes:

1. `bytes` can be used for arbitrary raw byte data
2. Fixed size is supported from `bytes1` to `bytes32`
3. Allows length & Index access *(ie. bytes[6] , bytes.length)*
4. storage `bytes` can be expended using push() operation. (Note: push() operation for memory `bytes` is not available)
5. `bytes` can be converted to `string` using string() constructor
6. As `bytes` supports fixed size variation, it consumes less gas

String:

1. `string` can be used for arbitrary string (UTF-8) characters
2. Fixed size is not supported for `string`
3. `string` is identical to `bytes` but does not allow length or index access
4. `string` can not be expanded as push() operation is not available.
5. `string` can be **seamlessly** converted to `bytes` using bytes() constructor
6. As `string` stores the data in UTF-8 encoding, it is quite expensive to compute the number of characters in the string (the encoding of some characters takes more than a single byte)

Practical Test

Question:

Please create a smart contract and deploy to the testnet.

Background:
Charles has a car selling shop and he is interested in using Ethereum as an efficient way to operate his business. The store can add the new cars into the inventory, and he can record down the car sales. Charles wants to see a dapp which associates an Ethereum address with a car to be sold.

The smart contract should consists of several functions:

1. Add new car into inventory
2. Record car sales record with the car identity

Car entity consists of:

- Price
- Ethereum address

Answer:

Please find below contract link in testnet
https://ropsten.etherscan.io/address/0x700fa7105876303c1cc240962ad48b902d72437a#code

Optional/Bonus Practical Test

Question:

Do you know the proxy/logic pattern of OpenZeppelin? Please use it for the below smart contracts (you can pick one of two):

https://github.com/daoventures/dao-protocol/tree/develop/YearnFarmerV2/contracts/vaults
Or
https://github.com/daoventures/dao-protocol/tree/develop/YearnFarmerV2/contracts/strategies

It would be better if you can deploy them to testnet and do some interactions.

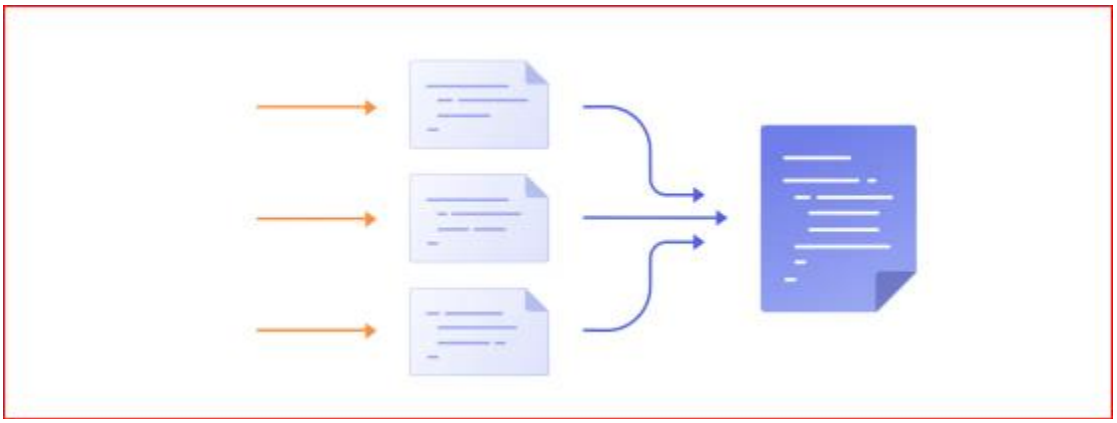
Answer:

Sorry, its not clear to me if I need to use minimal proxy OR upgradable proxy. Moreover, I am facing some compilation issue, so I couldn't deploy on testnet.

But, by looking at the contracts (in above links), I could see there is duplicate of contract code (except they refer different asset/tokens, eg: DAI, TUSD, USDC, USDT etc) which could be solved by cloning the contract using **minimal proxy**.

Instead of wasting a lot of gas to deploy the same code many times (for different assets/token), it's possible to use proxy contracts to cheaply re-use logic/implemtation contract many times, but have separate storage for each proxy user.

For example, in below image, all proxy contracts re-use same logic/implementation contract.



Lets say, we have a logic contract with generic implementation in "**DAOVaultMedium.sol**" and its constructor can accept token address and strategy address, then below is the contract factory which can be used to clone/proxy the logic contract.

```
pragma solidity ^0.4.23;

import "@optionality.io/clone-factory/contracts/CloneFactory.sol";
import "contracts/DAOVaultMedium.sol"

// Storage factory which creates clone/proxy contract
contract StorageFactory is DAOVaultMedium, CloneFactory {

    // admin who can create clone/proxy of logic contract
    address public admin;

    // address of logic/implementation contract
    address public impAddress;

    // Event whenever a clone is created
    event CloneCreated(address impleAddress);

    constructor (address _impAddress) public {
        admin = msg.sender;
        impAddress = _impAddress;
    }

    //modifier
    modifier onlyAdmin()
    {
        require(admin == msg.sender, "Only admin is allowed to create the
clone");
        _;
```

```
    }


function clone(address _token, address _strategy) public onlyAdmin() {
        address clonedAdd = createClone(impAddress);
        DAOVaultMedium(clonedAdd).init(_token, _strategy);

         // Trigger the event when a clone is created
        emit CloneCreated(clonedAdd);
    }
}
```