**2.blockcahin network**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract SimpleStorage {

uint256 storedData;

function set(uint256 x) public {

storedData = x;

}

function get() public view returns (uint256) {

return storedData;

}

}

**3.Transactions and network**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.19;

contract SimpleStorage {

uint256 private storedData;

event DataStored(uint256 value);

function set(uint256 x) public {

storedData = x;

emit DataStored(x);

}

function get() public view returns (uint256) {

return storedData;

}

}

**4.Asset transfer:**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract SimpleAssetTransfer {

struct Asset {

string name;

address owner;

}

mapping(uint256 => Asset) public assets;

mapping(address => uint256[]) private ownerAssets;

event AssetTransferred(uint256 indexed assetId, address indexed from,

address indexed to);

function createAsset(uint256 \_id, string memory \_name) public {

assets[\_id] = Asset(\_name, msg.sender);

ownerAssets[msg.sender].push(\_id);

}

function transferAsset(uint256 \_assetId, address \_to) public {

require(assets[\_assetId].owner == msg.sender, "Not the asset owner");

require(\_to != address(0), "Invalid recipient address");

address previousOwner = assets[\_assetId].owner;

assets[\_assetId].owner = \_to;

removeAssetFromOwner(previousOwner, \_assetId);

ownerAssets[\_to].push(\_assetId);

emit AssetTransferred(\_assetId, previousOwner, \_to);

}

function getAssetOwner(uint256 \_assetId) public view returns (address) {

return assets[\_assetId].owner;

}

function getOwnerAssets(address \_owner) public view returns (uint256[]

memory) {

return ownerAssets[\_owner];

}

function removeAssetFromOwner(address \_owner, uint256 \_assetId) internal {

uint256[] storage assetsOfOwner = ownerAssets[\_owner];

for (uint256 i = 0; i < assetsOfOwner.length; i++) {

if (assetsOfOwner[i] == \_assetId) {

assetsOfOwner[i] = assetsOfOwner[assetsOfOwner.length - 1];

assetsOfOwner.pop();

break;

}

}

}

}

**5.Track fitness**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Fitness {

struct Member {

address memberAddress;

uint256 points;

bool isMember;

}

mapping(address => Member) public members;

address public owner;

event MemberAdded(address indexed memberAddress);

event PointsUpdated(address indexed memberAddress, uint256 points);

modifier onlyOwner() {

require(msg.sender == owner, "Only owner can execute this function");

\_;

}

modifier onlyMember() {

require(members[msg.sender].isMember, "Only members can execute this

function");

\_;

}

constructor() {

owner = msg.sender;

}

function addMember(address \_memberAddress) public onlyOwner {

require(!members[\_memberAddress].isMember, "Address is already a member");

members[\_memberAddress] = Member({

memberAddress: \_memberAddress,

points: 0,

isMember: true

});

emit MemberAdded(\_memberAddress);

}

function updatePoints(address \_memberAddress, uint256 \_points) public

onlyOwner {

require(members[\_memberAddress].isMember, "Address is not a member");

members[\_memberAddress].points += \_points;

emit PointsUpdated(\_memberAddress, members[\_memberAddress].points);

}

function getPoints(address \_memberAddress) public view onlyMember returns

(uint256) {

require(members[\_memberAddress].isMember, "Address is not a member");

return members[\_memberAddress].points;

}

function redeemPoints(address \_memberAddress, uint256 \_points) public

onlyOwner {

require(members[\_memberAddress].isMember, "Address is not a member");

require(members[\_memberAddress].points >= \_points, "Insufficient points");

members[\_memberAddress].points -= \_points;

emit PointsUpdated(\_memberAddress, members[\_memberAddress].points);

}

}

6.Car auction:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract CarAuction {

struct Car {

string make;

string model;

uint256 year;

}

struct Auction {

Car car;

address payable seller;

uint256 startingBid;

uint256 highestBid;

address payable highestBidder;

bool active;

}

uint256 public auctionCount;

mapping(uint256 => Auction) public auctions;

event AuctionCreated(uint256 indexed auctionId, address indexed seller,

string make, string model, uint256 year, uint256 startingBid);

event HighestBidIncreased(uint256 indexed auctionId, address indexed bidder,

uint256 amount);

event AuctionEnded(uint256 indexed auctionId, address indexed winner,

uint256 amount);

function createAuction(string memory \_make, string memory \_model, uint256

\_year, uint256 \_startingBid) public {

auctions[auctionCount] = Auction({

car: Car({

make: \_make,

model: \_model,

year: \_year

}),

seller: payable(msg.sender),

startingBid: \_startingBid,

highestBid: 0,

highestBidder: payable(address(0)),

active: true

});

emit AuctionCreated(auctionCount, msg.sender, \_make, \_model, \_year,

\_startingBid);

auctionCount++;

}

function bid(uint256 \_auctionId) public payable {

Auction storage auction = auctions[\_auctionId];

require(auction.active, "Auction is not active");

require(msg.value > auction.highestBid, "There already is a higher

bid");

require(msg.value >= auction.startingBid, "Bid is lower than the starting

bid");

if (auction.highestBidder != address(0)) {

auction.highestBidder.transfer(auction.highestBid);

}

auction.highestBid = msg.value;

auction.highestBidder = payable(msg.sender);

emit HighestBidIncreased(\_auctionId, msg.sender, msg.value);

}

function endAuction(uint256 \_auctionId) public {

Auction storage auction = auctions[\_auctionId];

require(msg.sender == auction.seller, "Only the seller can end the

auction");

require(auction.active, "Auction is not active");

auction.active = false;

auction.seller.transfer(auction.highestBid);

emit AuctionEnded(\_auctionId, auction.highestBidder,

auction.highestBid);

}

function getAuctionDetails(uint256 \_auctionId) public view returns (string

memory make, string memory model, uint256 year, uint256 startingBid, uint256

highestBid, address highestBidder, bool active) {

Auction storage auction = auctions[\_auctionId];

return (auction.car.make, auction.car.model, auction.car.year,

auction.startingBid, auction.highestBid, auction.highestBidder,

auction.active);

}

}