**1.Basic**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

// Parent Contract

contract ParentContract {

address public owner;

// Constructor

constructor() {

owner = msg.sender; // Set contract creator as owner

}

// Public function

function getOwner() public view returns (address) {

return owner;

}

// Visibility: Internal function

function \_getBalance() internal view returns (uint) {

return owner.balance; // Return owner's balance

}

// Error handling with require

function transferOwnership(address newOwner) public {

require(newOwner != address(0), "New owner address invalid");

owner = newOwner;

}

}

// Child Contract inheriting ParentContract

contract ChildContract is ParentContract {

// Payable function

function deposit() public payable {

// Payable function to receive Ether

}

// Public function using internal function

function showOwnerBalance() public view returns (uint) {

return \_getBalance(); // Access internal function from parent contract

}

// Struct definition

struct Person {

string name;

uint age;

}

// Mapping

mapping(address => Person) public people;

// Function to add people to mapping

function addPerson(string memory \_name, uint \_age) public {

people[msg.sender] = Person(\_name, \_age);

}

// Enum definition

enum Status { Active, Inactive }

// Fixed array

uint[5] public fixedArray = [1, 2, 3, 4, 5];

// Dynamic array

uint[] public dynamicArray;

// Special arrays (bytes and string)

bytes public byteArray;

string public textString = "Hello, Solidity";

// Function to add to dynamic array

function addToArray(uint \_value) public {

dynamicArray.push(\_value);

}

}

**2. Merkle Tree**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract MerkleTree {

bytes32[] public leaves; // Array of leaf hashes

bytes32 public merkleRoot; // Store the Merkle root

// Add new leaf (hashed) to the leaves array

function addLeaf(bytes32 \_leaf) public {

leaves.push(\_leaf);

}

// Function to construct the Merkle Tree and compute Merkle Root

function generateMerkleRoot() public {

require(leaves.length > 0, "No leaves available");

uint n = leaves.length;

bytes32[] memory currentLevel = leaves;

// Continue hashing pairs of nodes until one hash (the root) remains

while (n > 1) {

uint nextLevelLength = n / 2;

if (n % 2 == 1) {

nextLevelLength++;

}

bytes32[] memory nextLevel = new bytes32[](nextLevelLength);

for (uint i = 0; i < n / 2; i++) {

nextLevel[i] = keccak256(abi.encodePacked(currentLevel[2 \* i], currentLevel[2 \* i + 1]));

}

// If there is an odd number of elements, promote the last one

if (n % 2 == 1) {

nextLevel[nextLevelLength - 1] = currentLevel[n - 1];

}

currentLevel = nextLevel;

n = nextLevelLength;

}

// The final root hash

merkleRoot = currentLevel[0];

}

// Verify whether a leaf belongs to the Merkle Tree using Merkle proof

function verify(bytes32 leaf, bytes32[] memory proof, bytes32 root) public pure returns (bool) {

bytes32 computedHash = leaf;

// Rebuild the hash from the proof

for (uint i = 0; i < proof.length; i++) {

bytes32 proofElement = proof[i];

if (computedHash <= proofElement) {

// Hash current computed hash with the proof element

computedHash = keccak256(abi.encodePacked(computedHash, proofElement));

} else {

// Hash proof element with current computed hash

computedHash = keccak256(abi.encodePacked(proofElement, computedHash));

}

}

// Check if the rebuilt hash matches the root

return computedHash == root;

}

}

**3.Bank**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DecentralizedBank {

// Mapping of user addresses to their Ether balances

mapping(address => uint) private balances;

// Event to log successful deposits

event Deposit(address indexed user, uint amount);

// Event to log successful withdrawals

event Withdrawal(address indexed user, uint amount);

// Deposit function (payable): Allows users to deposit Ether into the bank

function deposit() public payable {

require(msg.value > 0, "Deposit amount must be greater than zero");

balances[msg.sender] += msg.value;

emit Deposit(msg.sender, msg.value); // Emit deposit event

}

// Withdraw function: Allows users to withdraw Ether from their balance

function withdraw(uint \_amount) public {

require(balances[msg.sender] >= \_amount, "Insufficient balance");

// Deduct the amount from the user's balance

balances[msg.sender] -= \_amount;

// Transfer Ether to the user

payable(msg.sender).transfer(\_amount);

emit Withdrawal(msg.sender, \_amount); // Emit withdrawal event

}

// Function to check the user's balance

function getBalance() public view returns (uint) {

return balances[msg.sender];

}

}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Decentralized Bank</title>

</head>

<body>

<h1>Decentralized Bank</h1>

<div>

<label>Deposit Amount (in Ether):</label>

<input type="number" id="depositAmount" step="0.01" />

<button onclick="deposit()">Deposit</button>

</div>

<div>

<label>Withdraw Amount (in Ether):</label>

<input type="number" id="withdrawAmount" step="0.01" />

<button onclick="withdraw()">Withdraw</button>

</div>

<div>

<button onclick="getBalance()">Check Balance</button>

<p id="balance"></p>

</div>

<script src="https://cdn.jsdelivr.net/npm/web3@latest/dist/web3.min.js"></script>

<script>

// MetaMask provider

const web3 = new Web3(window.ethereum);

// Contract ABI and Address (replace with your contract's actual values)

const contractABI = [ /\* ABI array from your compiled contract \*/ ];

const contractAddress = 'YOUR\_CONTRACT\_ADDRESS';

const bankContract = new web3.eth.Contract(contractABI, contractAddress);

// Request MetaMask account access

async function requestAccount() {

await window.ethereum.request({ method: 'eth\_requestAccounts' });

}

// Deposit Ether into the bank

async function deposit() {

const amount = document.getElementById("depositAmount").value;

const accounts = await web3.eth.getAccounts();

const value = web3.utils.toWei(amount, 'ether');

await bankContract.methods.deposit().send({

from: accounts[0],

value: value

});

alert('Deposit successful!');

}

// Withdraw Ether from the bank

async function withdraw() {

const amount = document.getElementById("withdrawAmount").value;

const accounts = await web3.eth.getAccounts();

const value = web3.utils.toWei(amount, 'ether');

await bankContract.methods.withdraw(value).send({

from: accounts[0]

});

alert('Withdrawal successful!');

}

// Check balance

async function getBalance() {

const accounts = await web3.eth.getAccounts();

const balance = await bankContract.methods.getBalance().call({ from: accounts[0] });

document.getElementById("balance").innerText = web3.utils.fromWei(balance, 'ether') + ' ETH';

}

</script>

</body>

</html>

**4. Lottery**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.3;

contract Lottery {

address public owner;

address payable[] public players;

uint private lotteryId;

mapping (uint => address payable) public lotteryHistory;

event PlayerEntered(address indexed player, uint amount);

event WinnerPicked(address indexed winner, uint amount);

constructor() {

owner = msg.sender;

lotteryId = 1;

}

function getWinnerByLottery(uint lottery) public view returns (address payable) {

require(lottery > 0, "Invalid lottery ID");

require(lottery <= lotteryId, "Lottery ID exceeds current lottery");

return lotteryHistory[lottery];

}

function getLotteryId() public view returns (uint) {

return lotteryId;

}

function getBalance() public view returns (uint) {

return address(this).balance;

}

function getPlayers() public view returns (address payable[] memory) {

return players;

}

function enter() public payable {

require(msg.value > 0.01 ether, "Minimum entry fee is 0.01 ether");

require(lotteryId == 1, "Cannot enter after winner is picked");

players.push(payable(msg.sender));

emit PlayerEntered(msg.sender, msg.value);

}

function getRandomNumber() public view returns (uint) {

return uint(keccak256(abi.encodePacked(owner, blockhash(block.number - 1))));

}

function pickWinner() public onlyowner {

require(players.length > 0, "No participants in the lottery");

uint index = getRandomNumber() % players.length;

players[index].transfer(address(this).balance);

emit WinnerPicked(players[index], address(this).balance);

lotteryHistory[lotteryId] = players[index];

lotteryId++;

players = new address payable[](0);

}

modifier onlyowner() {

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

\_;

}

}

**5.Crowdfunding**

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Crowdfunding {

mapping(address=>uint) public contributors;

address public manager;

uint public minimum;

uint public deadline;

uint public target;

uint public raised;

uint public numofcontributors;

struct Request{

string description;

address payable recepient;

uint value;

bool completed;

uint noofVoters;

mapping (address=>bool) voters;

}

mapping (uint=> Request) requests;

uint public numofrequests;

constructor(uint \_target, uint \_deadline){

target=\_target;

deadline=block.timestamp+\_deadline;

minimum=100 wei;

manager=msg.sender;

}

modifier onlyManager(){

require(msg.sender==manager, "You are not me");

\_;

}

function createRequest(string calldata \_description, address payable \_recipient, uint \_value) public {

Request storage newRequest=requests[numofrequests];

numofrequests++;

newRequest.description=\_description;

newRequest.recepient=\_recipient;

newRequest.value=\_value;

newRequest.completed=false;

newRequest.noofVoters=0;

}

function contribution() public payable {

require(block.timestamp<deadline,"Deadline has passed");

require(msg.value>=minimum,"Minimum required is 100 wei");

if (contributors[msg.sender]==0){

numofcontributors++;

}

contributors[msg.sender]=msg.value;

raised+=msg.value;

}

function getBalance() public view returns(uint){

return address(this).balance;

}

function refund() public {

require(block.timestamp>deadline && raised<target, "You are not eligible for refund");

require(contributors[msg.sender]>0, "You are not a contributor");

payable(msg.sender).transfer(contributors[msg.sender]);

contributors[msg.sender]=0;

}

function voteRequest(uint \_request) public {

require(contributors[msg.sender]>0, "You are not a contributor");

Request storage thisRequest=requests[\_request];

require(thisRequest.voters[msg.sender]==false, "You have already voted");

thisRequest.voters[msg.sender]=true;

thisRequest.noofVoters++;

}

function makePayment(uint \_request) public onlyManager{

require(raised>=target, "Target is not Reached");

Request storage thisRequest=requests[\_request];

require(thisRequest.completed==false, "the request has been completed");

require(thisRequest.noofVoters>numofcontributors/2,"Majority does not support that");

thisRequest.recepient.transfer(thisRequest.value);

thisRequest.completed=true;

}

}

**6.Event Organization**

// SPDX-License-Identifier: MIT

pragma solidity >0.8.0 <0.9.0;

contract EventOrganization{

uint eventID;

struct Event {

address Event\_Organizer;

string Event\_Name;

uint Event\_Date;

uint Event\_Price;

uint Event\_Total\_Tickets;

uint Event\_Remain\_Tickets;

}

mapping(uint => Event) public events;

mapping(address => mapping(uint => uint)) public tickets;

function createEvent(string memory \_event\_name, uint \_event\_date, uint \_event\_price, uint \_event\_total\_tickets) external {

require(\_event\_date > block.timestamp + 1 days, "You can't set event in past, event date must be after one day");

require(\_event\_total\_tickets > 0 , "You can organize event, when amount of tickets more than 0");

events[eventID] = Event(msg.sender, \_event\_name, \_event\_date, \_event\_price, \_event\_total\_tickets, \_event\_total\_tickets);

eventID++;

}

function Buy\_Tickets(uint Event\_ID, uint Event\_Quantity) external payable {

require(events[Event\_ID].Event\_Date != 0, "Event not exist");

require(events[Event\_ID].Event\_Date > block.timestamp, "Event has already occured");

Event storage \_event = events[eventID];

require(msg.value == (\_event.Event\_Price\*Event\_Quantity));

require(\_event.Event\_Remain\_Tickets >= Event\_Quantity, "Not enough tickets");

require(msg.value == events[Event\_ID].Event\_Price , "Please Pay Correct Amount");

payable(events[Event\_ID].Event\_Organizer).transfer(msg.value);

\_event.Event\_Remain\_Tickets -= Event\_Quantity;

tickets[msg.sender][Event\_ID] += Event\_Quantity;

}

function Transfer\_Tickets(uint Event\_ID, uint Ticket\_Quantity, address to) external {

require(events[Event\_ID].Event\_Date != 0, "Event not exist");

require(events[Event\_ID].Event\_Date > block.timestamp, "Event has already occured");

require(tickets[msg.sender][Event\_ID] >= Ticket\_Quantity, "You do not have enough tickets");

tickets[msg.sender][eventID] -= Ticket\_Quantity;

tickets[to][eventID] += Ticket\_Quantity;

}

}

**Theory:**

### **Blockchain Terms**

1. **Blockchain**:  
   A decentralized, immutable ledger for recording transactions securely.
2. **Smart Contract**:  
   A self-executing program on a blockchain that automates tasks based on predefined conditions.
3. **Ethereum**:  
   A blockchain platform designed for deploying smart contracts and decentralized applications (dApps).
4. **Address**:  
   A unique identifier for accounts or smart contracts on the blockchain.
5. **Gas**:  
   Computational fees required to execute operations on the Ethereum network, paid in Ether.
6. **Ether (ETH)**:  
   The native cryptocurrency of the Ethereum network, used to pay for transactions and computational costs.
7. **msg.sender**:  
   The address of the account or contract calling the current function in a smart contract.
8. **Immutability**:  
   Once data is written to the blockchain, it cannot be altered or deleted.
9. **Consensus Mechanism**:  
   The protocol used to agree on the blockchain's current state (e.g., Proof of Stake or Proof of Work).

### **Solidity Terms**

1. **State Variable**:  
   Variables stored on the blockchain and persist as part of the contract's state (e.g., address public owner;).
2. **Constructor**:  
   A special function executed once during contract deployment, often used for initialization.
3. **Visibility Modifiers**:  
   Define access levels for functions and variables:
   * public: Accessible by anyone.
   * external: Callable only from outside the contract.
   * internal: Accessible only by the contract and its children.
   * private: Accessible only within the defining contract.
4. **Mapping**:  
   A key-value pair structure used for efficient data lookup (mapping(address => uint)).
5. **Struct**:  
   A custom data structure to group related variables (struct Person { string name; uint age; }).
6. **Enum**:  
   A user-defined type with a fixed set of constant values (enum Status { Active, Inactive }).
7. **Modifier**:  
   A reusable code block that alters the behavior of functions (e.g., onlyOwner for restricting access).
8. **Payable**:  
   A function modifier that allows the contract to receive Ether (function deposit() public payable {}).
9. **Fallback Function**:  
   A special unnamed function triggered when a contract receives plain Ether or invalid function calls.
10. **Events**:  
    Logging mechanism for emitting data on the blockchain, useful for tracking contract activity.
11. **ABI (Application Binary Interface)**:  
    A JSON format defining how to interact with a contract's functions and events.
12. **Gas Limit**:  
    The maximum gas allocated for a transaction to execute.
13. **Revert, Require, Assert**:  
    Statements for error handling in Solidity:
    * require: Checks inputs and conditions.
    * assert: Verifies invariants.
    * revert: Explicitly halts execution.

### **Relationships Between Solidity and Blockchain**

Solidity is the programming language used to write smart contracts that are deployed and executed on blockchains like Ethereum. These contracts interact with accounts, process transactions, and manage blockchain data securely and transparently.