Exp. No.	**1.Verifying If/Else Conditions on a Local Blockchain Environment using Truffle.**
Date:	

Aim:

To deploy and test a Solidity contract on a local blockchain environment using Truffle and Ganache, and verify the functionality of if/else conditions.

Hardware Requirements:

- Computer with at least 4 GB RAM
- Processor with at least dual-core CPU
- Stable internet connection for package downloads

Software Requirements:

- Operating System: Windows, macOS, or Linux
- Development Tools:
    - Node.js and npm (for package management)
    - Ganache (for local blockchain simulation)
    - Truffle Suite (for contract compilation, deployment, and testing)
    - MetaMask (for transaction verification)

Algorithm:

Step 1: Set Up Development Environment:

- Install Node.js and npm.
- Install Ganache and Truffle globally using npm.

Step 2: Write Solidity Contract:

- Create a new contract file, IfElse.sol, with if/else and ternary statements.
- Implement the foo() and ternary() functions.

Step 3: Compile and Deploy Contract:

- Compile the contract with truffle compile.

Step 4: Write Unit Tests:

- Create a test file under test/IfElseTest.js.
- Write test cases to validate the foo() and ternary() functions in the contract.

Step 5: Run Tests:

- Execute truffle test to check the functionality of the contract.

Code:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.26;

contract IfElse {

    function foo(uint256 x) public pure returns (uint256) {

        if (x < 10) {

            return 0;

        } else if (x < 20) {

            return 1;

        } else {

            return 2;

        }

    }

    function ternary(uint256 _x) public pure returns (uint256) {

        // if (_x < 10) {

        //     return 1;

        // }

        // return 2;
```

```
    // shorthand way to write if / else statement
      // the "?" operator is called the ternary operator
      return _x < 10 ? 1 : 2;
  }
}
```

Result:

Exp. No.	**2.Using For and While Loops in Solidity on a Local Blockchain Environment**
Date:	

Aim:

To implement and test the use of for and while loops in Solidity, demonstrating skip and exit operations in the for loop and simple iteration in the while loop.

Hardware and Software Requirements

- Hardware: A computer with internet access.
- Software:
    - Node.js and npm (Node Package Manager)
    - Truffle Suite
    - Ganache for a local blockchain environment

Algorithm

Step 1: Set up the development environment by installing Truffle and Ganache.

Step 2: Create a new Solidity contract named Loop.

Step 3: Write a loop function with:

Step 4: A for loop that iterates up to 10 times, uses continue to skip at i == 3, and break to exit the loop at i == 5.

Step 5: A while loop that iterates until j reaches 10.

Step 6: Deploy the contract on the local blockchain (Ganache) using Truffle.

Step 7: Test the contract functionality and verify the expected loop behavior.

Exp. No.	
Date:	

Code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract Loop {
    function loop() public {
        // for loop
        for (uint256 i = 0; i < 10; i++) {
            if (i == 3) {
                // Skip to next iteration with continue
                continue;
            }
            if (i == 5) {
                // Exit loop with break
                break;
            }
        }

        // while loop
        uint256 j;
        while (j < 10) {
            j++;
        }
    }
}
```

Result:

Exp. No.	**3.Using Mappings and Nested Mappings in Solidity on a Local Blockchain Environment**
Date:	

Aim:

To implement and test basic and nested mappings in Solidity, demonstrating how to set, get, and delete values in mappings.

Hardware and Software Requirements

- **Hardware**: A computer with internet access.
- **Software**:
  - Node.js and npm (Node Package Manager)
  - Truffle Suite
  - Ganache for a local blockchain environment

Algorithm

Step 1: **Set Up Development Environment**:

- Install Node.js and npm.
- Install Truffle globally: npm install -g truffle.
- Download and install Ganache for a local blockchain environment.

Step 2: **Create a New Truffle Project**:

- Open a terminal and create a new directory for the project.
- Initialize a new Truffle project using truffle init.

Step 3: **Write Solidity Contracts**:

- Create two contracts, Mapping and NestedMapping:

1.Mapping Contract:

- Implement a mapping that stores address => uint256.
- Implement get(), set(), and remove() functions to interact with the mapping.

2.Nested Mapping Contract:

- Implement a nested mapping that stores address => mapping(uint256 => bool).
- Implement get(), set(), and remove() functions to interact with the nested mapping.

Exp. No.	
Date:	

Step 4: **Deploy Contracts Using Truffle**:

- Compile the contracts using truffle compile.
- Deploy the contracts on Ganache using truffle migrate.

Step 5: **Test Contract Functions**:

- Interact with the deployed contracts using Truffle console or a script.
- Test the get(), set(), and remove() functions for both the basic and nested mappings.

Step 6: **Verify Output**:

- Confirm that the correct values are returned for get().
- Verify that values can be updated and deleted with set() and remove(), respectively.
- Ensure that nested mappings work as expected.

Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract Mapping {
    // Mapping from address to uint
    mapping(address => uint256) public myMap;

    function get(address _addr) public view returns (uint256) {
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return myMap[_addr];
    }

    function set(address _addr, uint256 _i) public {
        // Update the value at this address
        myMap[_addr] = _i;
    }

    function remove(address _addr) public {
        // Reset the value to the default value.
        delete myMap[_addr];
    }
}
```

Exp. No.	
Date:	

```
contract NestedMapping {
    // Nested mapping (mapping from address to another mapping)
    mapping(address => mapping(uint256 => bool)) public nested;

    function get(address _addr1, uint256 _i) public view returns (bool) {
        // You can get values from a nested mapping
        // even when it is not initialized
        return nested[_addr1][_i];
    }

    function set(address _addr1, uint256 _i, bool _boo) public {
        nested[_addr1][_i] = _boo;
    }

    function remove(address _addr1, uint256 _i) public {
        delete nested[_addr1][_i];
    }
}
```

Result:

Exp. No.	**4.Understanding Data Locations - Storage, Memory, and Calldata in Solidity**
Date:	

Aim:

To understand and demonstrate the usage of data locations (storage, memory, and calldata) in Solidity, and how they impact the storage and access of data in smart contracts.

Hardware and Software Requirements:

- Hardware: A computer with internet access.
- Software:
    - Node.js and npm (Node Package Manager)
    - Truffle Suite
    - Ganache for a local blockchain environment
    - Solidity (Version: ^0.8.26)

Algorithm

Step 1: Set Up Development Environment:

- Install Node.js and npm.
- Install Truffle globally: npm install -g truffle.
- Download and install Ganache for a local blockchain environment.

Step 2: Create a New Truffle Project:

- Open a terminal and create a new directory for the project.
- Initialize a new Truffle project using truffle init.

Step 3: Write Solidity Contracts:

- Write a contract that demonstrates the usage of storage, memory, and calldata:
    - Storage: Declare state variables that are stored on the blockchain.
    - Memory: Declare variables that are only in memory during function execution.
    - Calldata: Declare special function parameters that are read-only and cannot be modified.
    - The contract will include:

        1. A f() function that calls _f() with state variables, manipulates storage variables, and returns memory variables.

        2.g() and h() functions that demonstrate the usage of memory and calldata data locations.

Exp. No.	
Date:	

Step 4: Deploy Contracts Using Truffle:

- Compile the contracts using truffle compile.
- Deploy the contracts on Ganache using truffle migrate.

Step 5: Test Contract Functions:

- Test the f(), g(), and h() functions by interacting with them through the Truffle console.

Step 6: Verify Output:

- Ensure that state variables are stored on the blockchain and retain their values across transactions.
- Verify that memory variables exist only during function execution and are not stored on the blockchain.
- Test that calldata variables are read-only and cannot be modified within the contract.

Code

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.26;


contract DataLocations {

    uint256[] public arr;

    mapping(uint256 => address) map;


    struct MyStruct {

        uint256 foo;

    }
```

```
mapping(uint256 => MyStruct) myStructs;


  function f() public {
      // call _f with state variables
      _f(arr, map, myStructs[1]);


      // get a struct from a mapping
      MyStruct storage myStruct = myStructs[1];
      // create a struct in memory
      MyStruct memory myMemStruct = MyStruct(0);
  }


  function _f(
      uint256[] storage _arr,
      mapping(uint256 => address) storage _map,
      MyStruct storage _myStruct
  ) internal {
      // do something with storage variables
  }
```

```
  // You can return memory variables
    function g(uint256[] memory _arr) public returns (uint256[] memory) {
        // do something with memory array
    }

    function h(uint256[] calldata _arr) external {
        // do something with calldata array
    }
}
```

Result:

Exp. No.	**5.Understanding Ether and Wei in Solidity**
Date:	

Aim:
To understand and demonstrate the conversion between Ether, Wei, and Gwei in Solidity, and how transactions and values are represented in different units.

Hardware and Software Requirements:

- Hardware: A computer with internet access.
- Software:
    - Node.js and npm (Node Package Manager)
    - Truffle Suite
    - Ganache for a local blockchain environment
    - Solidity (Version: ^0.8.26)

Algorithm

Step 1: Set Up Development Environment:

- Install Node.js and npm.
- Install Truffle globally: npm install -g truffle.
- Download and install Ganache for a local blockchain environment**.**

Step 2: Create a New Truffle Project:

- Open a terminal and create a new directory for the project.
- Initialize a new Truffle project using truffle init.

Step 3: Write Solidity Contract:

- **Write a contract that demonstrates the usage of Ether, Wei, and Gwei.**
- **In the contract, define variables for:**
    - OneWei: Represents 1 Wei.
    - OneGwei: Represents 1 Gwei.
    - OneEther: Represents 1 Ether.
- **Add boolean variables to check whether the conversions work correctly:**
    - isOneWei: Should return true if oneWei equals 1.
    - isOneGwei: Should return true if oneGwei equals 1e9 (10^9).
    - isOneEther: Should return true if oneEther equals 1e18 (10^18).

Step 4: Deploy Contracts Using Truffle:

- Compile the contract using truffle compile.
- Deploy the contract on Ganache using truffle migrate.

Step 5: Test Contract Functions:

- Test the contract by interacting with the oneWei, oneGwei, and oneEther variables.
- Verify the boolean values for isOneWei, isOneGwei, and isOneEther.

Step 6: Verify Output:

- Ensure that the conversions between Wei, Gwei, and Ether are correct.
- Verify that the boolean checks (isOneWei, isOneGwei, isOneEther) return the correct values.

Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract EtherUnits {
    uint256 public oneWei = 1 wei;
    // 1 wei is equal to 1
    bool public isOneWei = (oneWei == 1);

    uint256 public oneGwei = 1 gwei;
    // 1 gwei is equal to 10^9 wei
    bool public isOneGwei = (oneGwei == 1e9);

    uint256 public oneEther = 1 ether;
    // 1 ether is equal to 10^18 wei
    bool public isOneEther = (oneEther == 1e18);
}
```

Result:

Exp. No.	**6.Gas and Gas Limit in Solidity**
Date:	

Aim:

To understand how gas works in Ethereum transactions, including how gas is consumed, gas price, and gas limit, and to demonstrate an example of gas usage in a contract.

Hardware and Software Requirements:

- **Hardware**: A computer with internet access.
- **Software**:
    - Node.js and npm (Node Package Manager)
    - Truffle Suite
    - Ganache for a local blockchain environment
    - Solidity (Version: ^0.8.26)

Algorithm

Step 1: **Set Up Development Environment**:

- Install Node.js and npm.
- Install Truffle globally: npm install -g truffle.
- Download and install Ganache for a local blockchain environment.

Step 2: **Create a New Truffle Project**:

- Open a terminal and create a new directory for the project.
- Initialize a new Truffle project using truffle init.

Step 3: **Write Solidity Contract**:

- Write a contract that demonstrates gas usage, including:
    - A variable i that increments inside a while loop indefinitely.
    - A function forever() that runs the loop until all gas is consumed, causing the transaction to fail.

Step 4: **Deploy Contracts Using Truffle**:

- Compile the contract using truffle compile.
- Deploy the contract on Ganache using truffle migrate.

Exp. No.	
Date:	

Step 5: **Test Gas Consumption**:

- Call the forever() function and observe the gas usage.
- Monitor the gas used and how the transaction fails when all gas is consumed.

Step 6: **Verify Output**:

- Ensure that the contract consumes all available gas and the transaction fails due to gas depletion.
- Check that the gas usage is correctly tracked in the Truffle console.

Code

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract Gas {
    uint256 public i = 0;

    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

Result:

Exp. No.	**7.Implement advanced user-defined value types (UDVTs) in Solidity, demonstrating their usage in the context of clock and duration values.**
Date:	

Aim:
To understand and implement advanced user-defined value types (UDVTs) in Solidity, demonstrating their usage in the context of clock and duration values.

Hardware and Software Requirements:

- Hardware: A computer with internet access.
- Software:
    - Node.js and npm (Node Package Manager)
    - Truffle Suite
    - Ganache for a local blockchain environment
    - Solidity (Version: ^0.8.26)

Algorithm

Step 1: Set Up Development Environment:

- Install Node.js and npm.
- Install Truffle globally: npm install -g truffle.
- Download and install Ganache for a local blockchain environment.

Step 2: Create a New Truffle Project:

- Open a terminal and create a new directory for the project.
- Initialize a new Truffle project using truffle init.

Step 3: Write Solidity Contract:

- Define user-defined value types (UDVTs) for Duration, Timestamp, and Clock.
- Implement a library LibClock for wrapping and unwrapping the UDVTs into a Clock type.
- Implement an example contract Examples showing both basic usage (without UDVTs) and advanced usage (with UDVTs).

Step 4: Deploy Contract Using Truffle:

- Compile the contract using truffle compile.
- Deploy the contract on Ganache using truffle migrate.

Exp. No.	
Date:	

Step 5: Test UDVTs:

- Test the example_no_uvdt and example_uvdt functions.
- Verify that the example_no_uvdt function works with basic types, while the example_uvdt function shows the correct usage of UDVTs.

Step 6: Verify Output:

- Ensure that UDVTs are properly used to wrap and unwrap Duration and Timestamp.
- Confirm that the wrap function in LibClock fails when input order is incorrect (wrong compilation will occur if order is incorrect).
- Check for correct handling of UDVTs and basic types.

Code

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.26;



// Code copied from optimism

// https://github.com/ethereum-optimism/optimism/blob/develop/packages/contracts-bedrock/src/dispute/lib/LibUDT.sol

type Duration is uint64;

type Timestamp is uint64;

type Clock is uint128;

library LibClock {

   function wrap(Duration _duration, Timestamp _timestamp)

       internal

       pure

       returns (Clock clock_)

   {

       assembly {
```

```
        // data | Duration | Timestamp
        // bit  | 0 ... 63 | 64 ... 127
        clock_ := or(shl(0x40, _duration), _timestamp)
    }
}

function duration(Clock _clock)
    internal
    pure
    returns (Duration duration_)
{
    assembly {
        duration_ := shr(0x40, _clock)
    }
}

function timestamp(Clock _clock)
    internal
    pure
    returns (Timestamp timestamp_)
{
    assembly {
        timestamp_ := shr(0xC0, shl(0xC0, _clock))

    }
```

```
    }
}

// Clock library without user defined value type
library LibClockBasic {
    function wrap(uint64 _duration, uint64 _timestamp)
        internal
        pure
        returns (uint128 clock)
    {
        assembly {
            clock := or(shl(0x40, _duration), _timestamp)
        }
    }
}
contract Examples {
    function example_no_uvdt() external view {
        // Without UDVT
        uint128 clock;
        uint64 d = 1;
        uint64 t = uint64(block.timestamp);
        clock = LibClockBasic.wrap(d, t);

    // Oops! wrong order of inputs but still compiles
        clock = LibClockBasic.wrap(t, d);
```

```
    }

    function example_uvdt() external view {

        // Turn value type into user defined value type

        Duration d = Duration.wrap(1);

        Timestamp t = Timestamp.wrap(uint64(block.timestamp));

        // Turn user defined value type back into primitive value type

        uint64 d_u64 = Duration.unwrap(d);

        uint64 t_u64 = Timestamp.unwrap(t);


        // LibClock example

        Clock clock = Clock.wrap(0);

        clock = LibClock.wrap(d, t);

        // Oops! wrong order of inputs

        // This will not compile

        // clock = LibClock.wrap(t, d);

    }

}
```

Results:

Exp. No.	8.Real-Time Function Handling and Output Management in Solidity
Date:	

Aim:

To understand the use of functions in Solidity for returning multiple values, handling array inputs/outputs, and leveraging key-value pair function calls, demonstrating how to manage real-time data interactions and function optimizations in decentralized applications.

Algorithm:

Step 1: Define a function that returns multiple values such as uint, bool, or address.

Step 2: Return values can be either unnamed, named, or assigned explicitly within the function.

Step 3: Demonstrate how to handle multiple return values using destructuring assignments.

Step 4: Define a function that accepts an array as an input and processes it.

Step 5: Define a function that returns an array and interacts with external function calls.

Step 6: Call functions using key-value pairs for better readability and manageability of inputs.

Exp. No.	
Date:	

Code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;
contract Function {
    // Functions can return multiple values
    function returnMany() public pure returns (uint256, bool, uint256) {
        return (1, true, 2);
    }
    // Return values can be named
    function named() public pure returns (uint256 x, bool b, uint256 y) {
        return (1, true, 2);
    }
    // Return values can be assigned to their name, no need for return statement
    function assigned() public pure returns (uint256 x, bool b, uint256 y) {
        x = 1;
        b = true;
        y = 2;
    }
    // Use destructuring assignment when calling another function that returns multiple values
    function destructuringAssignments()
        public
        pure
        returns (uint256, bool, uint256, uint256, uint256)
```

```
    {
        (uint256 i, bool b, uint256 j) = returnMany();

        // Values can be left out in destructuring
        (uint256 x, , uint256 y) = (4, 5, 6);

        return (i, b, j, x, y);
    }
    // Function accepting array as input
    function arrayInput(uint256[] memory _arr) public {}
    // Function returning array output
    uint256[] public arr;
    function arrayOutput() public view returns (uint256[] memory) {
        return arr;
    }
}

// Call function with key-value inputs
contract XYZ {
    function someFuncWithManyInputs(
        uint256 x,
        uint256 y,
        uint256 z,
        address a,
        bool b,
```

```
        string memory c
    ) public pure returns (uint256) {}

    function callFunc() external pure returns (uint256) {
        return someFuncWithManyInputs(1, 2, 3, address(0), true, "c");
    }
    function callFuncWithKeyValue() external pure returns (uint256) {
        return someFuncWithManyInputs({
            a: address(0),
            b: true,
            c: "c",
            x: 1,
            y: 2,
            z: 3
        });
    }
}
```

Result:

Exp. No.	**9. Creating and Deploying a Smart Contract (MetaCoin) to a Local Blockchain Using Truffle's Development Environment**
Date:	

Aim:

To create a Truffle project, compile and deploy a smart contract (MetaCoin) to a personal blockchain using Truffle's development environment.

Algorithm:

Step 1: Install Truffle using npm: npm install -g truffle

Step 2: Create a new Truffle project using the MetaCoin template: **truffle unbox metacoin**

Step 3: Explore the project structure:

- Review the smart contract in **contracts/MetaCoin.sol.**
- Review the deployment script in **migrations/1_deploy_contracts.js.**
- Review the test files in **test/TestMetaCoin.sol** and **test/metacoin.js.**

Step 4: Compile the smart contract: **truffle compile**

Step 5: Start a local blockchain for deployment using Truffle Develop: **truffle develop**

Step 6: Deploy the contract to the local blockchain: **migrate**

Step 7: Test the smart contract functions:

- Use **truffle test** to ensure the contract works as expected.

Exp. No.	
Date:	

Command to get the Code:

Install Truffle globally

npm install -g truffle

Create a new Truffle project with the MetaCoin box

truffle unbox metacoin

Compile the smart contract

truffle compile

Start Truffle's local development blockchain

truffle develop

Deploy the smart contract

migrate

Test the contract's functions

truffle test

Result:

Exp. No.	**10.Creating and Deploying an ERC721 Smart Contract for Non-Fungible Tokens (NFTs) on a Local Blockchain**
Date:	

Aim:

To create, compile, and deploy an ERC721 smart contract for minting and transferring Non-Fungible Tokens (NFTs) on a local blockchain using Truffle.

Algorithm:

1. **Install Truffle:** Install Truffle globally using npm with the command **npm install -g truffle.**
2. **Initialize a Truffle Project:** Initialize a new Truffle project using **truffle init** in the desired directory.
3. **Unbox ERC721 Template:** Use **truffle unbox metacoin** to get a project template or create an ERC721 template manually.
4. **Create ERC721 Contract:** Implement the ERC721 standard by writing the smart contract code that includes functions for minting, burning, and transferring tokens.
5. **Compile the Contract:** Compile the smart contract using the command **truffle compile.**
6. **Configure Truffle:** Configure **truffle-config.js** to point to a local development blockchain (like Ganache or Truffle Develop).
7. **Deploy Contract:** Create a migration script in the **migrations/** folder and deploy the contract using the command **truffle migrate.**
8. **Test the Contract:** Test functions like minting, burning, and transferring NFTs using Truffle's built-in test suite or custom test scripts.
9. **Interact with Contract:** Use Truffle console or a front-end interface to interact with the deployed contract.

Exp. No.	
Date:	

Code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

interface IERC165 {
    function supportsInterface(bytes4 interfaceID)
        external
        view
        returns (bool);
}

interface IERC721 is IERC165 {
    function balanceOf(address owner) external view returns (uint256 balance);
    function ownerOf(uint256 tokenId) external view returns (address owner);
    function safeTransferFrom(address from, address to, uint256 tokenId)
        external;
    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes calldata data
    ) external;
    function transferFrom(address from, address to, uint256 tokenId) external;
    function approve(address to, uint256 tokenId) external;
    function getApproved(uint256 tokenId)
        external
        view
        returns (address operator);
    function setApprovalForAll(address operator, bool _approved) external;
    function isApprovedForAll(address owner, address operator)
        external
        view
        returns (bool);
}

interface IERC721Receiver {
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
```

```
    ) external returns (bytes4);
}

contract ERC721 is IERC721 {
    event Transfer(
        address indexed from, address indexed to, uint256 indexed id
    );
    event Approval(
        address indexed owner, address indexed spender, uint256 indexed id
    );
    event ApprovalForAll(
        address indexed owner, address indexed operator, bool approved
    );

    // Mapping from token ID to owner address
    mapping(uint256 => address) internal _ownerOf;

    // Mapping owner address to token count
    mapping(address => uint256) internal _balanceOf;

    // Mapping from token ID to approved address
    mapping(uint256 => address) internal _approvals;

    // Mapping from owner to operator approvals
    mapping(address => mapping(address => bool)) public isApprovedForAll;

    function supportsInterface(bytes4 interfaceId)
        external
        pure
        returns (bool)
    {
        return interfaceId == type(IERC721).interfaceId
            || interfaceId == type(IERC165).interfaceId;
    }

    function ownerOf(uint256 id) external view returns (address owner) {
        owner = _ownerOf[id];
        require(owner != address(0), "token doesn't exist");
    }

    function balanceOf(address owner) external view returns (uint256) {
        require(owner != address(0), "owner = zero address");
        return _balanceOf[owner];
    }

    function setApprovalForAll(address operator, bool approved) external {
        isApprovedForAll[msg.sender][operator] = approved;
```

```
    emit ApprovalForAll(msg.sender, operator, approved);
}

function approve(address spender, uint256 id) external {
    address owner = _ownerOf[id];
    require(
        msg.sender == owner || isApprovedForAll[owner][msg.sender],
        "not authorized"
    );

    _approvals[id] = spender;

    emit Approval(owner, spender, id);
}

function getApproved(uint256 id) external view returns (address) {
    require(_ownerOf[id] != address(0), "token doesn't exist");
    return _approvals[id];
}

function _isApprovedOrOwner(address owner, address spender, uint256 id)
    internal
    view
    returns (bool)
{
    return (
        spender == owner || isApprovedForAll[owner][spender]
            || spender == _approvals[id]
    );
}

function transferFrom(address from, address to, uint256 id) public {
    require(from == _ownerOf[id], "from != owner");
    require(to != address(0), "transfer to zero address");

    require(_isApprovedOrOwner(from, msg.sender, id), "not authorized");

    _balanceOf[from]--;
    _balanceOf[to]++;
    _ownerOf[id] = to;

    delete _approvals[id];

    emit Transfer(from, to, id);
}

function safeTransferFrom(address from, address to, uint256 id) external {
```

```
    transferFrom(from, to, id);

    require(
        to.code.length == 0
            || IERC721Receiver(to).onERC721Received(msg.sender, from, id,"")
                == IERC721Receiver.onERC721Received.selector,
        "unsafe recipient"
    );
}

function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    bytes calldata data
) external {
    transferFrom(from, to, id);

    require(
        to.code.length == 0
            || IERC721Receiver(to).onERC721Received(msg.sender,from,id,data)
                == IERC721Receiver.onERC721Received.selector,
        "unsafe recipient"
    );
}

function _mint(address to, uint256 id) internal {
    require(to != address(0), "mint to zero address");
    require(_ownerOf[id] == address(0), "already minted");

    _balanceOf[to]++;
    _ownerOf[id] = to;

    emit Transfer(address(0), to, id);
}

function _burn(uint256 id) internal {
    address owner = _ownerOf[id];
    require(owner != address(0), "not minted");

    _balanceOf[owner] -= 1;

    delete _ownerOf[id];
    delete _approvals[id];

    emit Transfer(owner, address(0), id);
}
```

```
}

contract MyNFT is ERC721 {
    function mint(address to, uint256 id) external {
        _mint(to, id);
    }

    function burn(uint256 id) external {
        require(msg.sender == _ownerOf[id], "not owner");
        _burn(id);
    }
}
```

Result:

Exp. No.	**11.Creating and Deploying an ERC20 Token with Token Swap Functionality**
Date:	

Aim:

The aim of this experiment is to create an ERC20 token contract, deploy it, and implement a token swapping contract to facilitate the exchange of one ERC20 token for another.

Algorithm:

1. **Creating an ERC20 Token**:
    - Define an interface IERC20 with necessary ERC20 functions like transfer, approve, transferFrom, etc.
    - Implement an ERC20 contract that includes these functions, managing the total supply, balances, and approvals.
    - Provide mint and burn functionalities to issue or destroy tokens.
2. **Creating Token Swap Contract**:
    - Define the TokenSwap contract with two ERC20 tokens.
    - Initialize the contract with the token addresses, owners, and swap amounts.
    - Implement a swap function that allows the swapping of tokens between two users.
    - Ensure that both parties approve the contract to transfer their tokens before swapping.
    - Transfer tokens securely using transferFrom from one user to another.

Code:

ERC20 Token Interface and Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address recipient, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);
}

contract ERC20 is IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
    string public name;
    string public symbol;
    uint8 public decimals;

    constructor(string memory _name, string memory _symbol, uint8 _decimals) {
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
    }

    function transfer(address recipient, uint256 amount) external returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }

    function approve(address spender, uint256 amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
```

```
    }


  function transferFrom(address sender, address recipient, uint256 amount) external returns (bool) {
        allowance[sender][msg.sender] -= amount;
        balanceOf[sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(sender, recipient, amount);
        return true;
    }

    function _mint(address to, uint256 amount) internal {
        balanceOf[to] += amount;
        totalSupply += amount;
        emit Transfer(address(0), to, amount);
    }

    function _burn(address from, uint256 amount) internal {
        balanceOf[from] -= amount;
        totalSupply -= amount;
        emit Transfer(from, address(0), amount);
    }

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) external {
        _burn(from, amount);
    }
}
```

Token Swap Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "./IERC20.sol";

contract TokenSwap {
    IERC20 public token1;
    address public owner1;
    uint256 public amount1;
    IERC20 public token2;
    address public owner2;
    uint256 public amount2;

   constructor(
        address _token1,
        address _owner1,
        uint256 _amount1,
        address _token2,
        address _owner2,
        uint256 _amount2
    ) {
        token1 = IERC20(_token1);
        owner1 = _owner1;
        amount1 = _amount1;
        token2 = IERC20(_token2);
        owner2 = _owner2;
        amount2 = _amount2;
    }

    function swap() public {
        require(msg.sender == owner1 || msg.sender == owner2, "Not authorized");
        require(
            token1.allowance(owner1, address(this)) >= amount1,
            "Token 1 allowance too low"
        );
        require(
            token2.allowance(owner2, address(this)) >= amount2,
            "Token 2 allowance too low"
        );

        _safeTransferFrom(token1, owner1, owner2, amount1);
        _safeTransferFrom(token2, owner2, owner1, amount2);
```

```
    }



    function _safeTransferFrom(
        IERC20 token,
        address sender,
        address recipient,
        uint256 amount
    ) private {
        bool sent = token.transferFrom(sender, recipient, amount);
        require(sent, "Token transfer failed");
    }
}
```

Result :

Exp. No.	**12.Develop and Connect Your Decentralized Application (dApp) with MetaMask Using Ethers.js**
Date:	

Objective:
To create a decentralized application (dApp) with a React frontend that connects to MetaMask using Ethers.js, enabling users to view their Ethereum wallet address and balance.

Overview:
This experiment will guide you through creating a dApp with a simple React interface that prompts the user to connect to MetaMask. If the user does not have a MetaMask account, they can create one or log in to an existing account. Upon connecting, the dApp will display the user's wallet address and balance using Ethers.js to interact with the Ethereum blockchain.

Hardware Requirements:

- Computer with internet access
- Browser with MetaMask extension installed

Software Requirements:

- MetaMask extension
- Node.js (for npm)
- React framework
- Ethers.js library (v5.7)

What are dApps

A dApp is an application that builds on a decentralized network. It has a user interface and a smart contract.

DApps run on blockchains that are open-source, decentralized environments free from the control of any single actor. For example, a developer can create a Facebook-like app and run it on the Ethereum blockchain, and when any user makes a post, no one can delete it. On decentralized apps, no one person is fully managing user data.

A dApp can have a frontend that is built using any language. It can run on decentralized servers like IPFS or Swarm.

Exp. No.	
Date:	

What is MetaMask

MetaMask is a popular browser extension that serves as a cryptocurrency wallet that connects to the Ethereum blockchain. MetaMask is a secure wallet that enables users to interact with Ethereum's dApps. It allows users to store Ether and other ERC-20 tokens. Users can then spend these tokens on games, stake them on DeFi apps, and trade them on exchanges. MetaMask does not store only ERC-20 tokens but can also store ERC-721 tokens.
In this article, we will be connecting our dApp with MetaMask so users can connect to the Ethereum blockchain.

Install MetaMask

We will need to install the MetaMask extension in our browser to complete this tutorial.
To add MetaMask to your browser for Chrome, follow this link, but if you are using a different browser than Chrome, you can search for the extension for the browser you are using.

Why Ethers.js

Ethers.js is a lightweight JavaScript library that allows developers to connect and interact with the Ethereum blockchain. The library includes utility functions and has all the capabilities of an Ethereum wallet.

Ethers.js makes it easier to connect to the Ethereum blockchain with just a few lines of code.

With that bit of context, you are now armed with the knowledge of *why* we are using the technologies we are. You are now prepared with everything you need to start coding!

Exp. No.	
Date:	

Build a dApp frontend using React

With this application, we will explore the basic concepts of the react framework and the blockchain. We will create paths to communicate with the Ethereum blockchain, query it and fetch data to render in our dApp.
At the end of this tutorial, we will have a working webpage that interacts with Ethereum.

Run the command npx create-react-app ethersjs_meta to create a new react app named ethersjs_meta.

Next, run the command:
npm install ethers@5.7

This will install the Ethers.js (version 5.7) library.

Next, delete the dummy code in the App.js file, and develop the code below.

```
function App() {
 return (
   <div className="App-header">
     <div className="centerCard">
       <div className = "card">
         <div className="App">
           <WalletCard/>
         </div>
       </div>
     </div>
   </div>
 );
}
export default App;
```

Exp. No.	
Date:	

Next, we want to create a new file called WalletCard.js in our src folder. WalletCard.js will contain all of the UI and logic for our project.

```
import React, { useState } from 'react';
import { Button } from '@material-ui/core';
import Ethereum from './Ethereum.png'
import { ethers } from 'ethers';
const provider = new ethers.providers.Web3Provider(window.Ethereum)
const WalletCard = () => {
  const [errorMessage, setErrorMessage] = useState(null);
  const [defaultAccount, setDefaultAccount] = useState(null);
  const [userBalance, setUserBalance] = useState(null);
  const connectwalletHandler = () => {
    if (window.Ethereum) {
      provider.send("eth_requestAccounts", []).then(async () => {
        await accountChangedHandler(provider.getSigner());
      })
   } else {
      setErrorMessage("Please Install Metamask!!!");
    }
  }
  const accountChangedHandler = async (newAccount) => {
    const address = await newAccount.getAddress();
    setDefaultAccount(address);
    const balance = await newAccount.getBalance()
    setUserBalance(ethers.utils.formatEther(balance));
    await getuserBalance(address)
  }
  const getuserBalance = async (address) => {
    const balance = await provider.getBalance(address, "latest")
  }
  return (
    <div className="WalletCard">
      <img src={Ethereum} className="App-logo" alt="logo" />
      <h3 className="h4">
        Welcome to a decentralized Application
      </h3>
```

```
        <Button
          style={{ background: defaultAccount ? "#A5CC82" : "white" }}
          onClick={connectwalletHandler}>
          {defaultAccount ? "Connected!!" : "Connect"}
        </Button>
        <div className="displayAccount">
          <h4 className="walletAddress">Address:{defaultAccount}</h4>
          <div className="balanceDisplay">
            <h3>
              Wallet Amount: {userBalance}
            </h3>
          </div>
        </div>
        {errorMessage}
      </div>
  )
}
export default WalletCard;
```

Created a constant variable called provider and assigned it to new *ethers.providers.Web3Provider(window.ethereum)*, which is a read-only abstraction for accessing blockchain data.
We can access the whole Ethereum API using *window.ethereum,* which is why we passed it as a parameter to Ethers web3 provider.

Inside the WalletCard component, we created three states to mutate the app. We change the state when an error occurs, the account display changes, or the user balance changes.

```
const [errorMessage, setErrorMessage] = useState(null);
const [defaultAccount, setDefaultAccount] = useState(null);
const [userBalance, setUserBalance] = useState(null);
```

Blockchain Application and smart contract dBlockchain Application and smart contract dBlockchain Application and smart contract Create a connection with MetaMask

create a function called connectwalletHandler that will contain a request for permission to connect to MetaMask. The function includes the following.

```
const connectwalletHandler = () => {
  if (window.ethereum) {
     provider.send("eth_requestAccounts", [])
        .then(async () => {
           await accountChangedHandler(provider.getSigner());
        })
  } else {
     setErrorMessage("Please Install MetaMask!!!");
  }
}
```

Check to assert that the user has MetaMask installed; if the user does not have it installed, it should ask the user to install MetaMask. If it is installed, it should connect to MetaMask.

Build a connect Button: Create a button that allows a user to connect to Ethereum. This button will call the function onClick. When this button is clicked, we will connect to MetaMask.

```
<Button
 style={{ background: defaultAccount ? "#A5CC82" : "white" }}
 onClick={connectwalletHandler}>
   {defaultAccount ? "Connected!!" : "Connect"}
</Button>
```

The *defaultAccount* displays the wallet address, and we use the wallet address to show connected when the address is available. Otherwise, it should just show connect. Login to MetaMask : After connecting to MetaMask, we are prompted to log in.

Display User Wallet Balance

To get the user balance from the blockchain.

```
const accountChangedHandler = async (newAccount) => {
      const address = await newAccount.getAddress();
      setDefaultAccount(address);
      const balance = await newAccount.getBalance()
      setUserBalance(ethers.utils.formatEther(balance));
      await getuserBalance(address)
  }
```

Access the balance of the wallet using the provider.signer() passed as newAccount , using the getBalance() function but this would return a hexadecimal value and that would be very large and unreadable, so we have to covert it from Wei to Ether (Wei is the smallest denomination of Ether). To do this, we will do the following

```
setUserBalance(ethers.utils.formatEther(balance));
```

Next, Pass the address into the getuserBalance() to be specific as to what wallet we will be needing its balance.

```
const getuserBalance = async (address) => {
      const balance = await provider.getBalance(address, "latest")
  }
```

Inside the getuserBalance() function, we will pass the address to the provider.getBalance() that takes the address and blockTag as parameters. To render it in our UI, use the code below

```
<div className="balanceDisplay">
 <h3>
   Wallet Amount: {userBalance}
 </h3>
</div>
```

Result:

.

Exp. No.	
Date:	