

BITS, Pilani – Pilani Campus  
CS F452 (Blockchain Technology)  
Laboratory 3 (Week-3)

---

**AIM:**

To deepen understanding of advanced Solidity concepts, essential for building complex smart contracts on the Ethereum blockchain.

**TOPICS TO BE COVERED:**

1. Arrays
2. Mappings
3. Structure(Struct)
4. Constructors
5. Modifiers
6. Events
7. Inheritance
8. Payable Function

**Arrays:**

Arrays are a data structure used to store a fixed-size sequential collection of elements of the same type. They allow you to group multiple values together in a single variable, making it easier to manage collections of data.

**Types of Arrays in Solidity:**

1. **Fixed-size Arrays:** The size of the array is determined at the time of declaration and cannot be changed.
2. **Dynamic Arrays:** The size of the array can be changed during runtime, allowing for more flexibility.

## Built-in Features for Arrays in Solidity:

1. **".length" Property:** Returns the current number of elements in the array. For dynamic arrays (arrays without a pre-set length), Solidity offers additional functionality:

### Resizing with .length:

In earlier Solidity versions, you could resize arrays by setting a new length. However, from Solidity 0.6.0 onwards, directly modifying `.length` is deprecated. Instead, you add or remove elements with `.push()` and `.pop()`.

2. **.push() Function:** Appends a new element to the end of a dynamic array and increases the length by 1, it doesn't return any value.
3. **.pop() Function:** Removes the last element of a **dynamic array**, reducing its length by 1.

(💡 `Push()` and `Pop()` Works only for dynamic arrays not for fixed arrays)

Eg: This code snippet will teach us how to initialize the arrays of both types using all built in functions.

```
// Fixed-size array
uint[3] fixedArray = [1, 2, 3]; // Direct initialization

// Function to initialize fixed array if needed with a loop
function initializeFixedArray() public {
    for (uint i = 0; i < fixedArray.length; i++) {
        fixedArray[i] = (i + 1) * 10; // This will set fixedArray to [10, 20, 30]
    }
}

// Dynamic array
uint[] dynamicArray;

function initializeDynamicArray() public {
    for (uint i = 0; i < 3; i++) {
        dynamicArray.push((i + 1) * 10); // Adds 10, 20, 30 to the dynamic array
    }
}

// Accessing dynamicArray length and adding more elements
function extendDynamicArray() public {
    dynamicArray.push(40);
    // Adds 40 to the dynamic array
    // Now dynamicArray contains [10, 20, 30, 40]
    dynamicArray.pop();
    // Now dynamicArray contains [10, 20, 30]
}
```

4. **.delete Keyword:** Sets an array element at a specific index to its default value (0 for uint, false for bool, etc.).

```
uint[3] fixedArray = [1, 2, 3];  
delete fixedArray[1]; // fixedArray becomes [1, 0, 3]
```

Data Type	Default Value	Description
uint	0	Unsigned integer defaults to 0
int	0	Signed integer defaults to 0
bool	false	Boolean defaults to false
address	0x00	Empty address, all bits zero
bytes	0x (empty bytes)	Empty byte array
string	"" (empty string)	Empty string

Default value table for each data type

**Structure(Struct):**In Solidity, a struct is a user-defined data type that allows you to group multiple variables of different types into a single entity. Structs are particularly useful for modeling complex data structures. They help in organizing data and improving the readability and maintainability of smart contracts.

```
// put your pragma solidity ^0.8.0;  
  
struct Person {  
    string name; // Dynamic size  
    uint age;     // Fixed size  
    address wallet; // Fixed size  
}
```

## Definition:

They can be stored in both storage (permanent) and memory (temporary).

**Storage (Permanent)** :When a struct is declared as a state variable in a contract, it is stored in the blockchain's storage. This means that the data persists between function calls and transactions.

**Memory (Temporary)**:When a struct is created as a local variable inside a function, it is stored in memory. This means that the data only exists for the duration of the function call and is cleared once the function execution is complete.

```
// put your pragma solidity ^0.8.0;

contract Example {
    // State variable in storage
    struct Person {
        string name;
        uint age;
        address wallet;
    }

    Person public storedPerson; // This is in storage

    // Function to create a person in memory and return it
    function createPerson(string memory _name, uint _age, address _wallet) public pure returns (Person memory) {
        Person memory newPerson = Person(_name, _age, _wallet); // This is in memory
        return newPerson; // Return a memory struct
    }

    // Function to set the storage struct
    function setStoredPerson(string memory _name, uint _age, address _wallet) public {
        storedPerson = Person(_name, _age, _wallet); // Set the storage struct
    }
}
```

## Mapping:

Mappings in Solidity are a key data structure that allows you to store and retrieve data in an associative array format. They are similar to hash tables or dictionaries in other programming languages. A mapping essentially links a unique key to a specific value.

a key can be any of the built-in data types but reference types are not allowed while the value can be of any type.

```

pragma solidity ^0.8.0;

contract SimpleStorage {
    // Declare a mapping from address to uint
    mapping(address ⇒ uint) public balances;

    // Function to set the balance of a specific address
    function setBalance(uint amount) public {
        balances[msg.sender] = amount; // Set the balance for the sender's address
    }

    // Function to get the balance of a specific address
    function getBalance(address user) public view returns (uint) {
        return balances[user]; // Return the balance for the specified address
    }
}

```

1. Mappings do not have a length property, as they can dynamically grow as new key-value pairs are added.
2. If you try to access a value for a key that doesn't exist in the mapping, it will return the default value for the value type (e.g., `0` for `uint`, `address(0)` for `address`, and `false` for `bool`).

### Nested Mapping:

We'll create a contract called `UserProfileManager` that allows:

1. Adding user profiles, each containing a name and a list of hobbies.
2. Updating hobbies for a user.
3. Retrieving user profiles.

#### 1. Struct Definition:

- We define a struct called `UserProfile` that contains:
  - A `string` for the user's name.
  - An array of strings for the user's hobbies.

#### 2. Mapping:

- A mapping profile links an Ethereum address to a `UserProfile` struct. This allows each user to have their profile stored against their address.

- Functions:
  - **setUserProfile:**
    - This function allows a user to set or update their name and hobbies. If the user already has a profile, it will update the existing one. It's kept storage structure because we are storing a record.
  - **addHobby:**
    - This function allows a user to add a single hobby to their existing hobbies list.
  - **getUserProfile:**
- This view function returns the user's name and hobbies, allowing users to retrieve their profile information. It's kept as memory structure because we just want to read the information from record and hence once function is executed we don't need it.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract UserProfileManager {

    // Struct to represent a user's profile
    struct UserProfile {
        string name;           // Name of the user
        string[] hobbies;      // Array to store hobbies of the user
    }

    // Mapping from the user's address to their profile
    mapping(address => UserProfile) public profiles;

    // Function to create or update a user profile
    function setUserProfile(string memory _name, string[] memory _hobbies) public {
        UserProfile storage profile = profiles[msg.sender]; // Get the profile for the sender
        profile.name = _name; // Set the name
        profile.hobbies = _hobbies; // Set the hobbies
    }

    // Function to add a hobby to the user's profile
    function addHobby(string memory _hobby) public {
        profiles[msg.sender].hobbies.push(_hobby); // Add hobby to the user's hobbies array
    }

    // Function to get the user's profile
    function getUserProfile() public view returns (string memory name, string[] memory hobbies) {
        UserProfile memory profile = profiles[msg.sender]; // Get the profile for the sender
        return (profile.name, profile.hobbies); // Return name and hobbies
    }
}
```

With these foundational concepts in mind, we are now prepared to explore more advanced Solidity concepts , which are part of previously studied primitive topics.

## Constructors:

A constructor is a special function in Solidity that runs only once, right when the contract is deployed. It's generally used to set up initial values for state variables or establish ownership.

### Mandatory Components :

1. **Constructor Keyword**(Its required to implement any constructor)
2. **Body** ( its contains code to initialize the state of a constructor)

### Optional Components:

1. **Visibility:** Constructors can be marked as `public` or `internal`. By default, they are `public`, but you can explicitly set `internal` if you want it accessible only within the contract or its derived contracts.
2. **Parameters:** Constructors can take parameters that are provided at deployment time.

```
pragma solidity ^0.8.0;

contract ExampleContract {
    address public owner;

    // Constructor initializes the owner as the deployer's address
    constructor() {
        owner = msg.sender; // Sets the deployer as the owner
    }
}
```

Diagram annotations: A yellow arrow points from the text "Constructor Keyword" to the `constructor()` keyword. A yellow box highlights the code `owner = msg.sender;` inside the constructor, with the text "Constructor Body" pointing to it.

In this example, the `constructor` is used to initialize the `owner` variable with the address that deploys the contract. `msg.sender` is a global variable that represents the address of the account initiating the transaction. The visibility (default `public`) is omitted here, but it can be explicitly stated.

## Modifiers:

Modifiers are optional components that restrict or enforce specific rules on functions. **Modifiers don't run on their own**; they're executed as part of the function's bytecode when the function is called. This allows modifiers to add conditions dynamically but means they're embedded into the function code itself after compilation.

Modifiers are not standalone executable code snippets, they just get added to the function as logic during compilation and get executed during runtime.

```

pragma solidity ^0.8.0;

contract ExampleContract {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    // Modifier with a condition
    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    function restrictedFunction() public onlyOwner {
        // Restricted code
    }
}

```

### Mandatory Components:

1. **modifier** Keyword: The **modifier** keyword is required.
2. **Modifier Name**: A unique name for each modifier.
3. **Modifier Body**: The **{ }** braces, along with the **\_** placeholder, are required.

### Optional Components:

- **Parameters**: Modifiers can accept parameters to customize conditions.

Eg: Code Snippet , showing the use of modifier.

1. When an account calls **updateData()**, the **onlyOwner** modifier is checked first.
2. If **msg.sender** (the caller's address) matches **owner**, the code at **\_** is replaced with the function body of **updateData()**, and it executes.
3. If not, it stops and reverts with the error message **"Not the owner"**.
4. This approach is useful for securing functions that should be accessible only to specific accounts, like the contract owner, in this case.



## EVENT:

An event in Solidity is like a "broadcast" or "announcement" that a smart contract can send out when something specific happens. Other applications (like a dApp) can "listen" for these events and take action when they occur.

These broadcasts are called **“logs”**, event is a special type of log which serves a way for contract to notify external systems

**“emit”** – emit keyword is used to trigger an event within a smart contract function , the listening done by other applications is done once an event is emitted , which allows them to track or react to specific contract actions without constantly calling the contract.

```
// put yo// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract PiggyBank {
    // Declare an event to log each deposit action
    event DepositMade(address indexed depositor, uint256 amount, uint256 timestamp);

    // Function to allow users to deposit ether into the piggy bank
    function deposit() public payable {
        require(msg.value > 0, "Deposit must be greater than 0");

        // Emit the event to log this deposit action
        emit DepositMade(msg.sender, msg.value, block.timestamp);
    }
}
```

**If Alice deposits 1 Ether(Digital Currency in Ethereum Blockchain), here’s what the event log will look like:**

```
Event: DepositMade
- depositor: 0xAliceAddress
- amount: 1 Ether
- timestamp: 1630000000 (example timestamp)
```

**If Bob deposits 2 Ether, the event log will look like:**

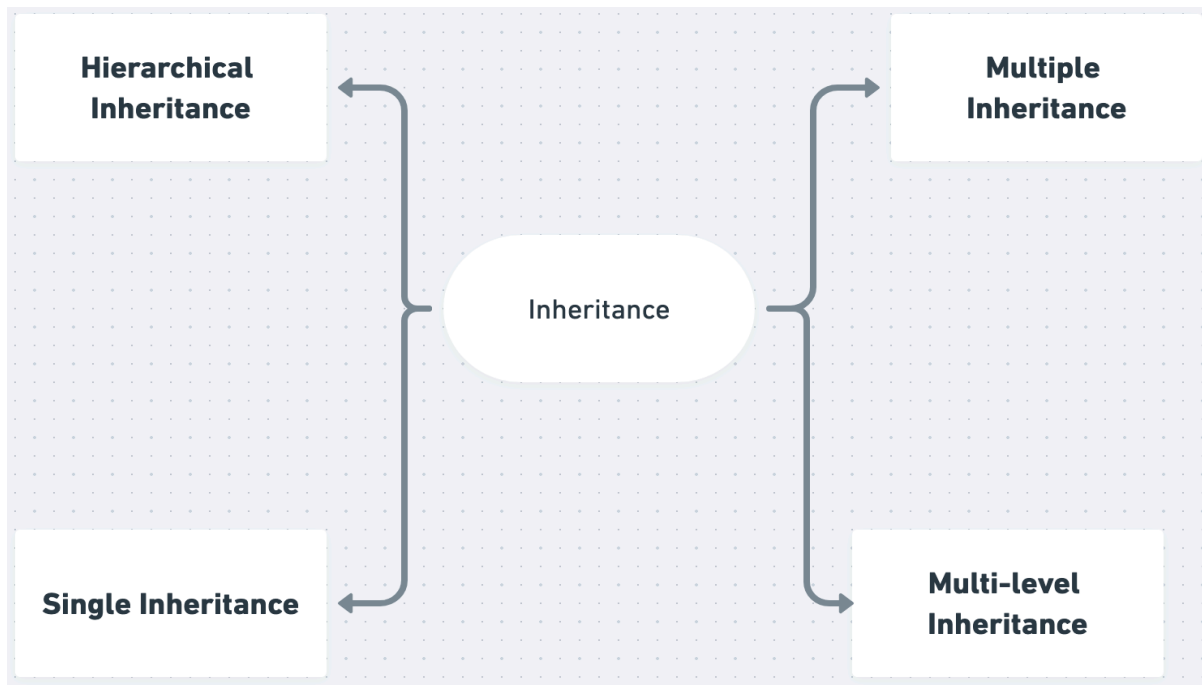
```
Event: DepositMade
- depositor: 0xBobAddress
- amount: 2 Ether
- timestamp: 1630000050)
```

## Concept of Inheritance:

In programming, inheritance lets us create new contracts based on existing ones, reusing and extending their features. Think of it like a family where certain characteristics (like eye color or height) are passed from parents to children. In Solidity, inheritance allows us to build contracts that inherit properties and functions from other contracts, reducing code repetition and organizing code more effectively.

In Solidity:

- The base contract is like the parent, which has properties and functions.
- The derived contract (child) inherits these features from the parent.



## Types of Inheritance

### Single Inheritance

In **Single Inheritance**, one contract (child) inherits features from only one base contract (parent). This is the simplest form of inheritance.

The `Parent` contract has a function `setValue` that calculates a sum. The `Child` contract inherits from `Parent`, so it can use `sum` and `setValue`. The `Child` contract adds its own function, `getValue`, to access `sum`.

```
pragma solidity ≥0.4.22 <0.6.0;

contract Parent {
    uint internal sum;

    function setValue() external {
        uint a = 10;
        uint b = 20;
        sum = a + b;
    }
}

contract Child is Parent {
    function getValue() external view returns (uint) {
        return sum;
    }
}
```

## Multi-Level Inheritance:

In Multi-Level Inheritance, there are multiple levels of inheritance, where a derived contract becomes a parent to another contract.

```
pragma solidity ≥0.4.22 <0.6.0;

contract A {
    string internal x;
    string a = "Hello";
    string b = "World";

    function getA() external {
        x = string(abi.encodePacked(a, b));
    }
}

contract B is A {
    string public y;
    string c = "Ethereum";

    function getB() external {
        y = string(abi.encodePacked(x, c));
    }
}

contract C is B {
    function getC() external view returns (string memory) {
        return y;
    }
}
```

A combines “ Hello” and “World” as string , then B inherits A and adds “Ethereum” , further C inherits B and gets access to everything in A and B.

## Hierarchical Inheritance:

In Hierarchical Inheritance, one base contract is inherited by multiple derived contracts. This is useful when several child contracts need to share common functionality.

```
pragma solidity ≥0.4.22 <0.6.0;

contract A {
    string internal x = "Shared Functionality";

    function setX() external {
        x = "Updated Value";
    }
}

contract B is A {
    function getX() external view returns (string memory) {
        return x;
    }
}

contract C is A {
    function getUpdatedX() external view returns (string memory) {
        return x;
    }
}
```

A contains a variable and function , by inheriting it both B and C can access A's properties and methods.

## Multiple Inheritance:

In Multiple Inheritance, a contract can inherit from more than one parent contract. Solidity supports this, but it requires careful handling to avoid conflicts.

```

pragma solidity ≥0.4.22 <0.6.0;

contract A {
    string internal name = "Contract A";

    function setName() external {
        name = "Updated A";
    }
}

contract B {
    uint internal value;

    function setValue() external {
        value = 100;
    }
}

contract C is A, B {
    function getDetails() external view returns (string memory, uint) {
        return (name, value);
    }
}

```

Contract C inherits both A and B, gaining access to their properties (name from A and value from B). getDetails can return both name and value since C has inherited both.



Access Control: Only **public** and **internal** functions and variables are accessible in derived contracts. Private members cannot be inherited.

Now lets discuss Function Overriding and Super Keyword.

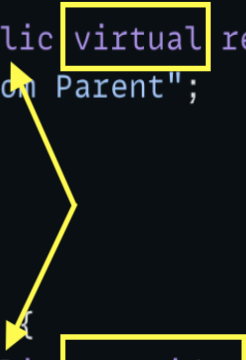
### Function Overriding:

Function Overriding in Solidity allows a derived (child) contract to provide a specific implementation of a function that is already defined in its base (parent) contract. This is essential when the child contract needs to modify or extend the behavior of the parent contract's function.

```
pragma solidity ^0.8.0;

// Base contract
contract Parent {
    function greet() public virtual returns (string memory) {
        return "Hello from Parent";
    }
}

// Derived contract
contract Child is Parent {
    function greet() public override returns (string memory) {
        return "Hello from Child";
    }
}
```



**Virtual Functions:** To enable a function in a base contract to be overridden, it must be marked with the `virtual` keyword.

**Override Keyword:** The overriding function in the child contract must use the `override` keyword to indicate that it is replacing the base contract's function.

**Access Specifiers:** The overriding function must have the same visibility (e.g., `public`, `internal`) as the function it overrides.

## Payable Functions

Payable functions are special functions in Solidity that can receive Ether during a transaction. They are essential for contracts that need to handle funds, such as decentralized applications (dApps) involving payments, crowdfunding, or auctions.

- **payable Keyword:** Functions must be explicitly marked with the **payable** keyword to accept Ether.
- **msg.value:** This global variable holds the amount of Wei (the smallest unit of Ether) sent with the transaction.
- **Receiving Ether:** Besides payable functions, contracts can also receive Ether through the **receive** and **fallback** functions.

```
pragma solidity ^0.8.0;

contract Fundraiser {
    address public owner;
    uint public totalRaised;

    constructor() {
        owner = msg.sender;
    }

    // Payable function to receive donations
    function donate() public payable {
        require(msg.value > 0, "Donation must be greater than zero");
        totalRaised += msg.value;
    }

    // Withdraw function restricted to owner
    function withdraw() public {
        require(msg.sender == owner, "Only owner can withdraw");
        payable(owner).transfer(totalRaised);
        totalRaised = 0;
    }
}
```

# Walkthrough Exercise: Building a Decentralized Library System:

Lets build a Simple Library System using Structure of a Book , Array of Structure to have dynamic record , Maps to link issuer's address to the Book and Record Events.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// Base contract
contract Library {
    struct Book {
        string title;
        string author;
        bool isAvailable;
    }

    Book[] public books;
    mapping(address => Book[]) public borrowedBooks;

    event BookAdded(string title, string author);
    event BookBorrowed(address borrower, string title);
    event BookReturned(address borrower, string title);

    // Add a new book to the library
    function addBook(string memory _title, string memory _author) public {
        books.push(Book({
            title: _title,
            author: _author,
            isAvailable: true
        }));
        emit BookAdded(_title, _author);
    }
}
```



We can extend this contract , to add a function specific to the owner of library. Function removeBook () which will remove the Book from the collection.

```
// Derived contract with restricted access
contract ManagedLibrary is Library {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;
    }

    // Remove a book (only owner can remove)
    function removeBook(uint _index) public onlyOwner {
        require(_index < books.length, "Invalid index");
        books[_index] = books[books.length - 1]; // Replace with last book
        books.pop(); // Remove the last book
    }
}
```

To implement borrowing a book function , we charge a fee to the user and thus we make this function payable. We will override the addBook function to

```
// Final derived contract with borrowing functionality
contract DecentralizedLibrary is ManagedLibrary {
    uint public borrowFee = 0.01 ether;
    uint public addBookFee = 0.005 ether; // Fee for adding a book

    // Override addBook to include payment logic
    function addBook(string memory _title, string memory _author) public payable override onlyOwner {
        require(msg.value ≥ addBookFee, "Insufficient fee to add a book");

        // Call the base addBook function to handle book addition
        super.addBook(_title, _author);
    }

    // Borrow a book
    function borrowBook(uint _index) public payable {
        require(_index < books.length, "Book does not exist");
        Book storage book = books[_index];
        require(book.isAvailable, "Book is not available");
        require(msg.value ≥ borrowFee, "Insufficient fee");

        book.isAvailable = false;
        borrowedBooks[msg.sender].push(book);
        emit BookBorrowed(msg.sender, book.title);
    }

    // Return a book
    function returnBook(uint _index) public {
        require(_index < borrowedBooks[msg.sender].length, "Invalid index");
        Book storage book = borrowedBooks[msg.sender][_index];

        book.isAvailable = true; // Mark the book as available again
        emit BookReturned(msg.sender, book.title);

        // Remove the book from borrowedBooks
        borrowedBooks[msg.sender][_index] = borrowedBooks[msg.sender][borrowedBooks[msg.sender].length - 1];
        borrowedBooks[msg.sender].pop();
    }

    // Withdraw collected fees (only owner)
    function withdrawFees() public onlyOwner {
        payable(owner).transfer(address(this).balance);
    }
}
```

## Excercise:

Design and Implement an advanced voting system using Solidity, applying key concepts such as arrays, mappings, structs, constructors, modifiers, events, inheritance, and payable functions. The system must enable users to create proposals, cast votes, and view results while maintaining the integrity and transparency of the voting process.

As a base, the system should include functionality to **manage proposals, track votes, and emit appropriate events** for key actions like proposal creation and voting.

A derived contract should **introduce access control**, restricting administrative tasks to a designated admin.

Furthermore incorporate advanced features such as **requiring a fee for proposal creation, securely handling Ether transactions**, and **implementing mechanisms to conclude voting on proposals and withdraw fees**.

Implement logical constraints to ensure fairness (e.g., **preventing double voting**), and adhere to security best practices for handling payments.

Design the system with proper event logging for transparency and deploy and test the contract to ensure it functions as intended. Deliverables include the fully implemented Solidity code and evidence of successful deployment and testing.