ADT and Array

Abstract Data Type (ADT)

Abstract Data Type (ADT) is a model for a data structure that defines the behavior of the data type without specifying its implementation. It focuses on *what operations* a data type can perform rather than *how it performs* those operations.

Key Features of ADT:

1. **Encapsulation of Data and Operations:**
    - ADTs hide the internal representation of data. Users interact with the data only through a set of well-defined operations, which ensures that the data is safe from unintended interference.
2. **Separation of Interface and Implementation:**
    - The ADT defines an interface (a set of operations), but how those operations are implemented is not exposed. This allows for flexibility in implementation.
3. **Reusability:**
    - Since the implementation is hidden, ADTs can be easily reused or changed without affecting the rest of the code that relies on them.

Example of ADT: Stack

A **stack** is a classic example of an ADT that operates on the *Last In, First Out (LIFO)* principle.

1. **Operations:**
    - `push(item)`: Add an item to the top of the stack.
    - `pop()`: Remove and return the item from the top of the stack.
    - `peek()`: Return the item at the top of the stack without removing it.
    - `isEmpty()`: Check if the stack is empty.

    - `size()`: Return the number of items in the stack.

2. **Implementation:**
    - The implementation can vary: a stack can be implemented using arrays, linked lists, or any other data structure, but this is hidden from the user.

Example Code (Java):

```
// Stack ADT Interface
public interface StackADT<T> {
    void push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    int size();
}

// Array-based Stack Implementation
public class ArrayStack<T> implements StackADT<T> {
    private T[] stack;
    private int top;
    private int capacity;

    @SuppressWarnings("unchecked")
    public ArrayStack(int capacity) {
        this.capacity = capacity;
        stack = (T[]) new Object[capacity];
        top = -1;
    }

    @Override
    public void push(T item) {
        if (top == capacity - 1) {
            System.out.println("Stack is full");
            return;
        }
        stack[++top] = item;
    }

    @Override
    public T pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return null;
        }
        return stack[top--];
    }

    @Override
    public T peek() {
        if (isEmpty()) {
            return null;
        }
        return stack[top];
    }

    @Override
    public boolean isEmpty() {
        return top == -1;
    }

    @Override
    public int size() {
        return top + 1;
    }
}
```

In the above example, the `StackADT` defines the interface (the ADT), and the `ArrayStack` class provides one possible implementation of that ADT using an array.

Array

An **array** is a data structure that stores a fixed-size, sequential collection of elements of the same data type. Arrays are one of the simplest data structures and are widely used because of their efficiency in accessing elements.

Key Features of Arrays:

1. **Fixed Size:**
    - The size of an array is determined at the time of creation and cannot be changed. This is known as a static data structure.
2. **Homogeneous Elements:**
    - All elements in an array must be of the same data type (e.g., all integers, all strings).
3. **Index-Based Access:**
    - Each element in an array can be directly accessed using its index. The index starts from 0 (first element) to `n-1` (last element), where `n` is the size of the array.

Array Operations:

1. **Access (Retrieve/Read):**
    - You can directly access any element using its index.
    - Example: `arr[2]` retrieves the element at index 2.
2. **Insert (Add):**
    - Insert a value at a specific index (if the array has space available).
    - Example: `arr[0] = 10;`
3. **Update (Modify):**
    - Change the value at a specific index.
    - Example: `arr[2] = 15;`
4. **Delete:**

- Removing elements is not straightforward since arrays have a fixed size. You can set the element to `null` or `0`.
- Example: `arr[3] = 0;`

5. **Traverse:**
   - Iterate over all elements in the array using a loop.
   - Example:

```
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

Example Code (Java):

```
public class ArrayExample {
    public static void main(String[] args) {
        // Creating an array of integers with size 5
        int[] arr = new int[5];

        // Inserting values into the array
        arr[0] = 10;
        arr[1] = 20;
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;

        // Accessing and printing a specific element
        System.out.println("Element at index 2: " + arr[2]);
// Output: 30

        // Updating an element at a specific index
        arr[2] = 35;

        // Printing all elements (Traverse)
        System.out.println("Array Elements:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}
```

Output:

```
Element at index 2: 30
Array Elements:
10
20
35
40
50
```

Relationship Between ADT and Array

1. **ADT Perspective:**
    - When we think of arrays as an ADT, we focus on the operations (e.g., insert, update, retrieve) that we can perform on the array. The actual implementation (how the elements are stored in memory) is hidden.
2. **Implementation Perspective:**
    - In programming, arrays are implemented at a low level where each element is stored in contiguous memory locations. This allows for efficient access using the index.