

Please use the following QR code to check in and record your attendance.

CS 1027

Fundamentals of Computer  
Science II

# Abstract Data Types

---

Ahmed Ibrahim



# Collections

---

- **Collection:** a group of items we wish to treat as a **conceptual unit**.
  - In programming, a conceptual unit refers to a single, unified entity that groups multiple items or elements together, treating them as one coherent object or collection.
- In Java, the **Collections Framework** provides a set of interfaces and classes for storing and manipulating groups of data as a **single unit**.
- Choosing the **right** collection type can significantly impact a solution's performance and clarity. Properly choosing a collection for a given problem can improve the solution's efficiency and simplicity.

# Example

---

- Imagine you're designing a system that tracks unique customer IDs for a **rewards program**.  
You need to:
  - Ensure each customer ID is unique.
  - Frequently check if a particular customer ID is already part of the program.
  - Occasionally, iterate over all customer IDs in the order they were added.
- Ideas:
  - **Set** - A collection that cannot contain duplicate elements.
  - **Queue** - A collection for holding elements before processing.
  - And much more ...
- But what exactly defines these choices? Behind each of these **collection types** is a concept known as an Abstract Data Type (**ADT**).

# Abstract Data Type (ADT)

---

- ADTs are theoretical models that define the behaviour and properties of a data structure, focusing on **what** operations it should support.
- It's like a **blueprint** that helps us determine the most efficient way to organize and access data.
- By understanding the ADTs behind collections — like sets, queues or more — you can more easily select the **right** structure to meet your program's requirements and ensure efficient and effective data management.

# Abstract Data Type (ADT) cont.

---

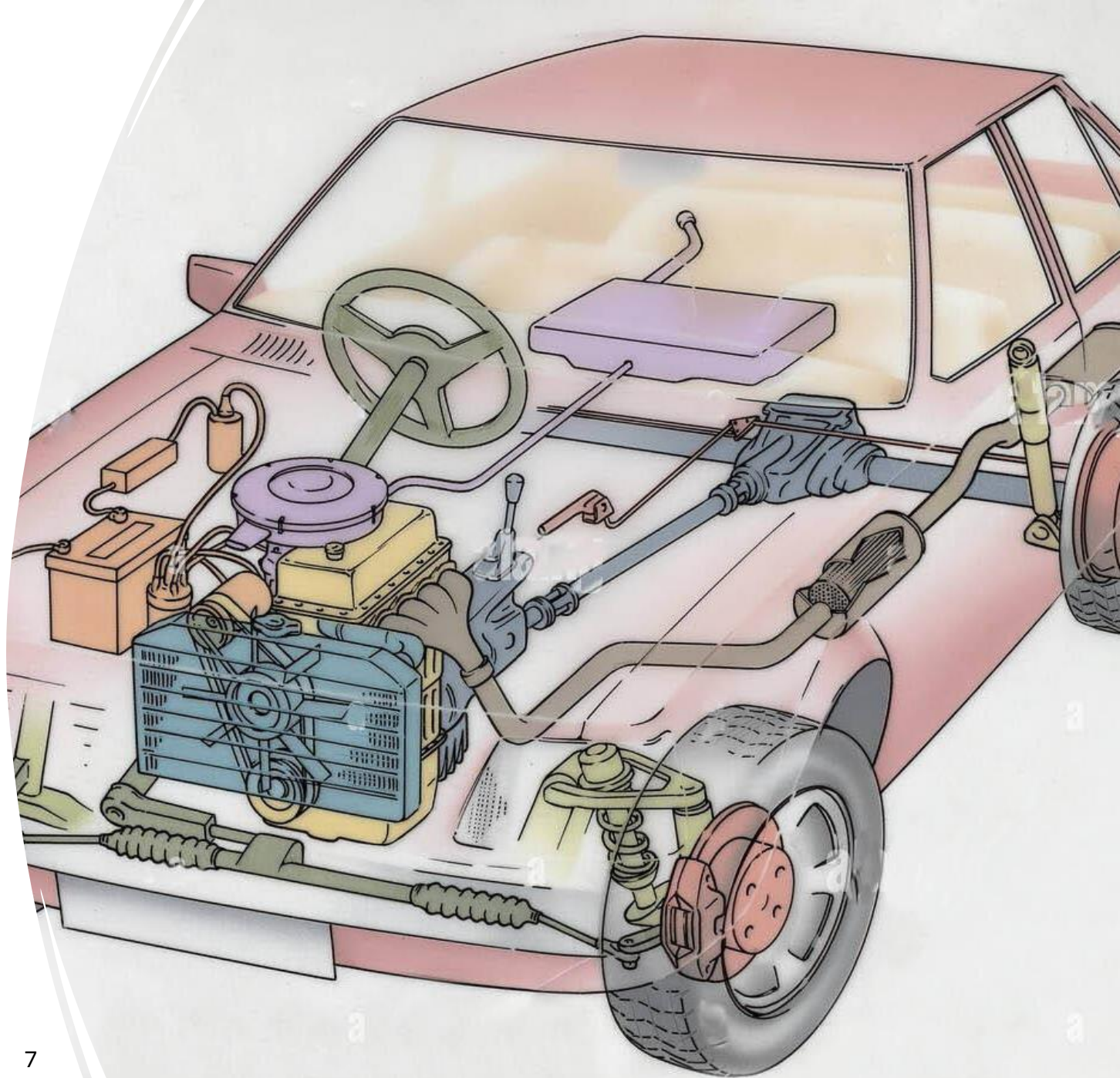
- For example, an ADT might define operations for **adding** and **removing** elements from a collection. The ADT specifies **what** these operations do—adding or removing an element from the collection—but does not specify **how** they are performed.
- It leaves out implementation details, such as:
  - **Which data structure** is used to store the elements?
  - **Where** a new element is added to the collection, and **how** elements are reorganized when one is removed.
- This **abstraction** allows ADTs to focus on the collection's **behaviour** and **properties** while leaving flexibility for different implementations that best suit particular use cases or performance requirements.

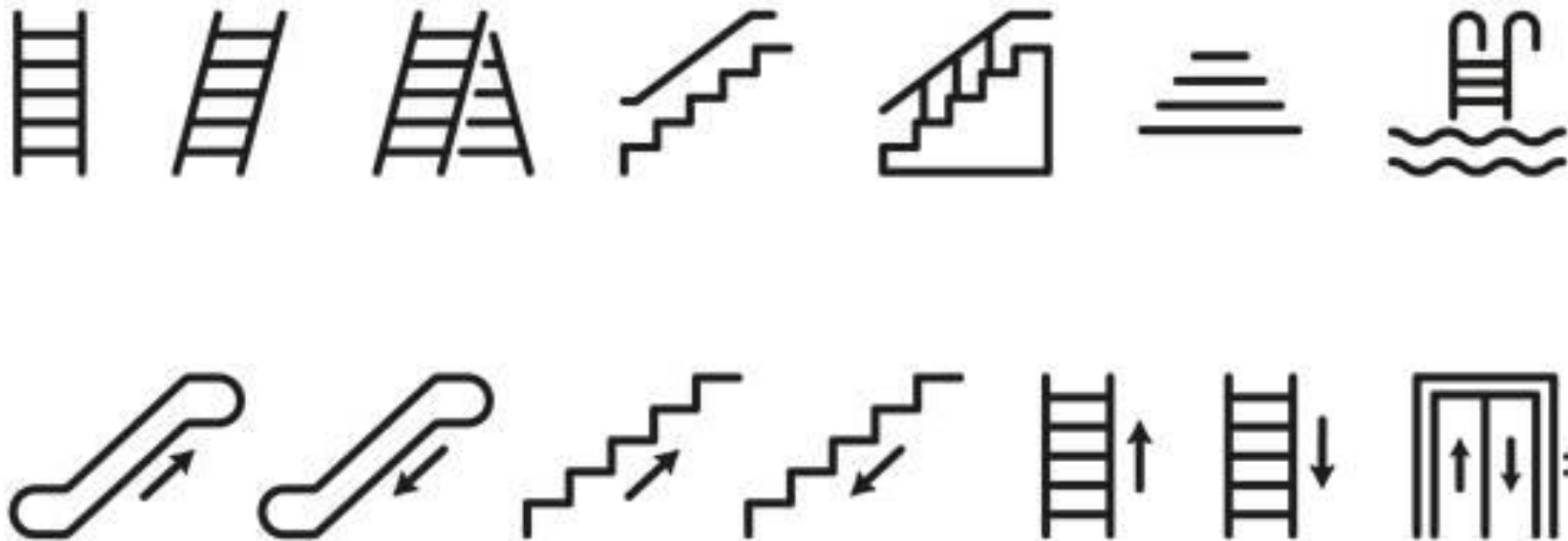


# Abstraction

---

- Abstraction separates the purpose of an entity from its implementation or how it works
- Example in real life: a car (**we do not have to know how an engine works to drive a car**)





Abstraction in design focuses on highlighting essential elements and relevant details while omitting unnecessary specifics,

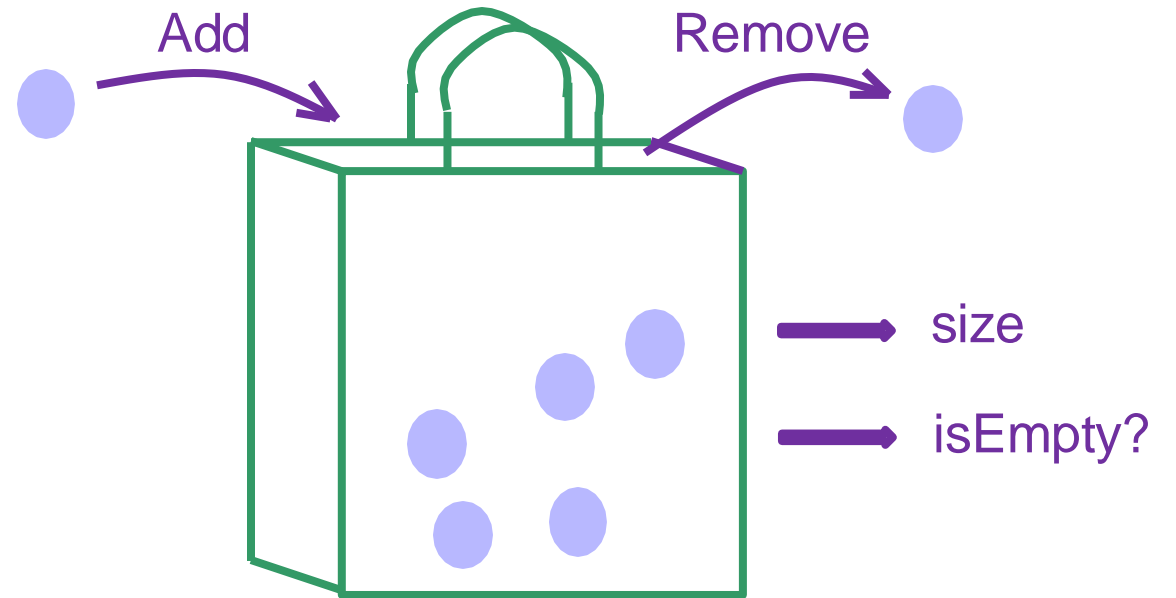


# Designing an Abstract Data Type (ADT)

Designing an Abstract Data Type (ADT) involves defining its **behaviour** through a **set of operations** and describing its properties without specifying its implementation details.

# Example: The Bag ADT

- Here's an example of an ADT design process for a **Bag ADT**.
- A **bag** is a collection that holds multiple data items with **no specific order** or **unique constraints**. It provides the following **operations**:
  - **Add**: Insert a new data item into the bag.
  - **Remove**: Delete a specified data item from the bag.
  - **Count**: Returns the number of occurrences of the element x in the bag.



# The Bag ADT: Usage Scenarios

- Bags are used in various applications where **duplicates** are allowed, and the order of elements does not matter. Some examples include:
  - **Inventory Management:** In a game or store inventory system, a bag can hold multiple instances of the same item (e.g., multiple potions or books).
  - **Word Frequency Counting:** A bag can be used to count **occurrences** of words in a document without caring about the order.

# Common ADTs and Their Use Cases

---

ADT	Characteristics	When to Use
<b>List</b>	Ordered, allows duplicates	When order matters or when accessing by index
<b>Set</b>	Unordered, no duplicates	When uniqueness of items is required
<b>Queue</b>	FIFO (First-In-First-Out)	For tasks that need sequential processing
<b>Stack</b>	LIFO (Last-In-First-Out)	For tasks that require reversing or backtracking
<b>Map</b>	Key-value pairs, unique keys	When mapping relationships or fast lookups are needed
<b>Bag</b>	Unordered, allows duplicates	When collecting items with no order or uniqueness requirements

# Choosing the Right ADT for a Problem

- **Efficiency:** The choice of Abstract Data Type (ADT) can significantly impact a solution's performance. Selecting the right ADT ensures efficient use of memory and processing power.
- **Simplicity:** An appropriate ADT can make code easier to read, maintain, and extend by aligning data organization with problem requirements.

# Question!

---

You are designing a software system for a library to manage book inventory and user requests. The system must meet the following requirements:

1. Track each book's unique ISBN number and allow quick lookups by ISBN.
2. Allow multiple copies of the same book in the inventory.
3. Process book requests from users in the order they are received.

Which combination of ADTs would be most suitable to meet these requirements?

- A) Use a List to store all books and a Queue to process user requests.
- B) Use a Set to store unique ISBNs and a Queue to manage user requests.
- C) Use a Map for ISBN lookups, a Bag for multiple copies, and a Queue for user requests.
- D) Use a Stack to manage book inventory and a Queue to handle user requests.

# Implementing ADT using Interfaces



---

- In Java, an **interface** is a programming construct used to define the operations of an **Abstract Data Type (ADT)**.
- An **interface** specifies a collection of **abstract methods** (method signatures without implementation) and **constants** that a class must implement to adhere to a particular behaviour.
- Key characteristics of a Java interface include:
  - It serves as a **blueprint** for classes, defining method signatures that any implementing class must provide.
  - Java interfaces are public by default, ensuring they are accessible to all classes within and outside the package.
  - Constants within an interface must be declared **static final**, meaning their values are fixed and cannot be changed.



# Interfaces

---

- Interfaces cannot be instantiated; This means that you cannot create an object of an interface type using **new InterfaceName()** because interfaces are **abstract** by nature and do not have any **concrete implementation**.
- Instead, to use an interface, you must create an instance of a class that implements that interface.
- When a class (like **MyClass**) implements an interface (such as **MyInterface**), it provides concrete implementations for all the abstract methods defined in the interface. Only then can you instantiate the class and assign it to a variable of the interface type, as in:
- **MyInterface** data = **new** MyInterface();  Invalid.
- **MyInterface** data = **new** MyClass();  Valid, if **MyClass** implements **MyInterface**

# Implementing an Interface

---

- A class must use the `implements` keyword to implement an interface in Java. This establishes a contract where the class agrees to provide **concrete implementations** for all methods defined in the interface. For example:

```
public class IntBag implements BagADT {  
    private int[] array; // Array to store data items  
    private int count;   // Tracks the number of items currently in the bag  
  
    // Here, we must implement all methods specified by the BagADT interface.  
    // These might include methods like add, remove, and getCount to manage items in the bag.  
}
```

- Within the `IntBag` class:
  - The array variable serves as the **underlying storage** for data items.
  - The **count** variable keeps track of the number of items in the bag.

# IntBag Class

- By implementing BagADT, the `IntBag` class could provide methods like **`add()`**, **`remove()`**, and **`getCount()`**, each following the guidelines defined in the interface.
- This setup allows `IntBag` to be used in any context where BagADT is expected, regardless of its specific internal implementation.

# Generic Types

---

- The `IntBag` class is limited to storing `int` values. However, creating a separate class for each data type (e.g., `String`, `Double`) would be inefficient and result in redundant code.
- A more flexible and efficient solution is to use **generic types**.
- By making the bag class generic, we can allow it to store any data type, providing greater versatility without duplicating code.

```
public class BagADT <T> {  
    // Array to store items of any type  
    private T[] items;  
    // Number of items in the bag  
    private int count;  
  
    // Here, we must implement all methods  
    // specified by the BagADT interface.  
    // These might include methods like add,  
    // remove, and getCount to manage items  
    // in the bag.  
}
```

# Generic Types and ADT

---

- Simply adding a class parameter `<T>` does not automatically make a class capable of handling any data type. We also need to design the methods to work with the **generic type**.
- This means replacing specific data types with the generic type parameter `T` wherever the type is used. For instance, a method like `public void add(int dataItem);`
- It would need to be changed to `public void add(T dataItem);`
- This way, the method can handle data of any type specified by `T`, making the class **fully generic** and adaptable to various data types.

# Java Interface for a Bag of ints

```
public interface BagADT {
    // Array to store items of any type
    private int[] items;
    // Number of items in the bag
    private int count;

    // Add a new data item to the bag
    public void add (int dataItem);

    // Returns true if dataItem was removed from the bag;
    // returns false if dataItem was not in the bag
    public boolean remove (int dataItem);

    // Returns the number of data items in the bag
    public int size ( );
}
```

```
public interface BagADT<T> {
    // Array to store items of any type
    private T[] items;
    // Number of items in the bag
    private int count;

    // Add a new data item to the bag
    public void add (T dataItem);

    // Returns true if dataItem was removed from the bag;
    // returns false if dataItem was not in the bag
    public boolean remove (T dataItem);

    // Returns the number of data items in the bag
    public int size ( );
}
```

Does this 'int' have to change to 'T'? Why or why not?

# Challenges with Generics and Arrays in Java

---

- In Java, **generics** are implemented with **type erasure**, meaning generic type information is removed at runtime.
- Compilation error:
  - Due to this, Java cannot directly create arrays of a generic type like `T[]`.
  - Attempting `new T[initCapacity]` will result in a compilation error.
- Using **Object Arrays**
  - We can use an `Object[]` to store items since `Object` can represent **any type**.
  - By **casting** `Object[]` to `T[]`, we create a **generic array** while handling Java's type system limitations.



# Example

---

```
public class Bag<T> implements BagADT<T>
{
    // Use an array to store items
    private T[] array;
    // Number of data items
    private int count;

    public Bag (int initCapacity) {
        count = 0;
        array = new T[initCapacity];
    }
```

Why is this wrong?

```
public class Bag<T> implements BagADT<T>
{
    // Use an array to store items
    private T[] array;
    // Number of data items
    private int count;
    public Bag (int initCapacity) {
        count = 0;
        array = new Object[initCapacity];
    }
```

Why is this wrong?

# IntBag Class

```
public class Bag<T> implements BagADT<T> {  
    // Use an array to store items  
    private T[] array;  
    // Number of data items  
    private int count;  
    public Bag (int initCapacity) {  
        count = 0;  
        array = (T[])(new Object[initCapacity]);  
    }
```

Casting



- While this approach generates an unchecked cast warning, it's commonly used as the best workaround to implement generic arrays.

# Question!

---

Consider the following Java code:

```
public class BagTest {  
    public static void main(String[] args) {  
        Bag<String> stringBag = new Bag<>(3);  
        stringBag.add("apple");  
        stringBag.add("banana");  
        stringBag.add("cherry");  
        System.out.println(stringBag.get(1));  
    }  
}
```

If this code is executed, what will be printed?

- A) apple
- B) banana
- C) cherry
- D) null

```
public class Bag<T> {  
    private Object[] items; // Use Object array  
    private int count;  
    public Bag(int size) {  
        items = new Object[size]; // Create an Object array  
        count = 0;  
    }  
    public void add(T item) {  
        if (count < items.length) { // Check for capacity  
            items[count++] = item;  
        } else {  
            throw new IllegalStateException("Bag is full");  
        }  
    }  
    public T get(int index) {  
        if (index >= 0 && index < count) { // Check for valid index  
            return (T) items[index]; // Cast to T when retrieving  
        } else {  
            throw new IndexOutOfBoundsException("Invalid index");  
        }  
    }  
}
```

# Data Structures

- Implementing collections in programming relies on the use of data structures.
- Common examples include:
  - **Arrays:** A contiguous block of memory that stores elements of the same type, allowing fast access by index.
  - **Linked Lists:** A sequence of nodes, each containing data and a reference to the next node, which provides flexibility in memory allocation and efficient insertions/deletions.

# Mapping ADTs to Data Structures

- **Performance Optimization:** The underlying data structure determines the efficiency of operations like insertion, deletion, and lookup for an ADT.
- **Aligning Characteristics:** Choosing the right data structure for an ADT aligns with the desired behaviour (e.g., **order**, **uniqueness**), making solutions more efficient and effective.

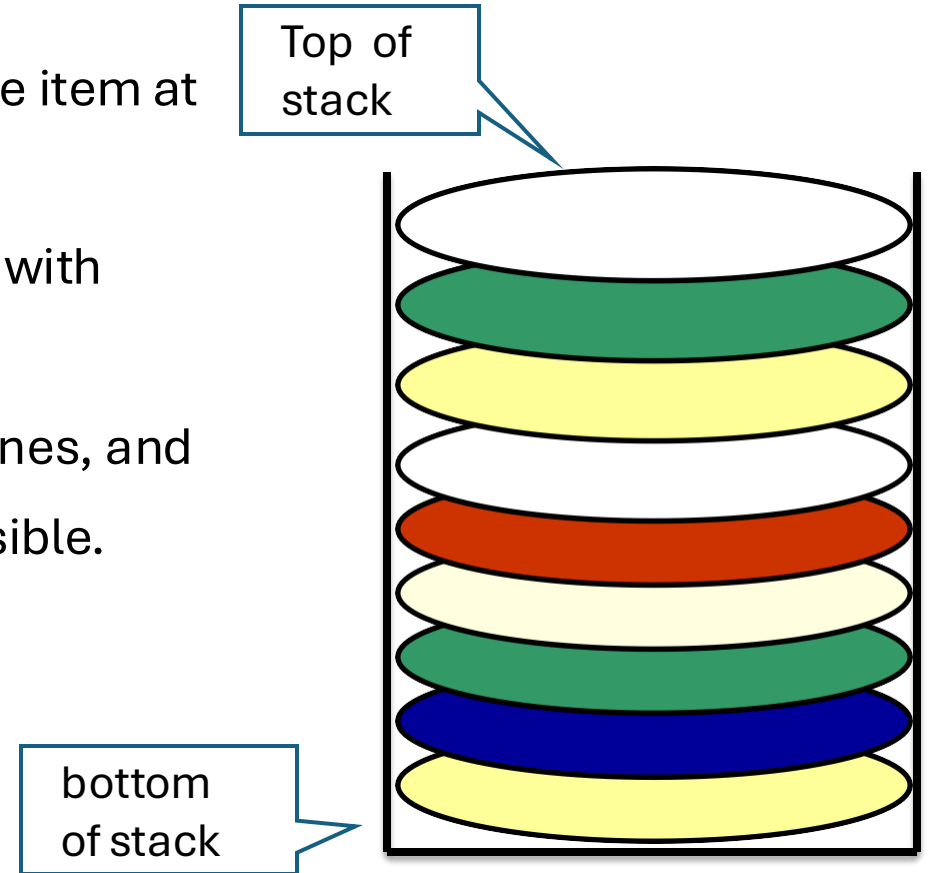
# Stack ADT



# The Stack ADT

---

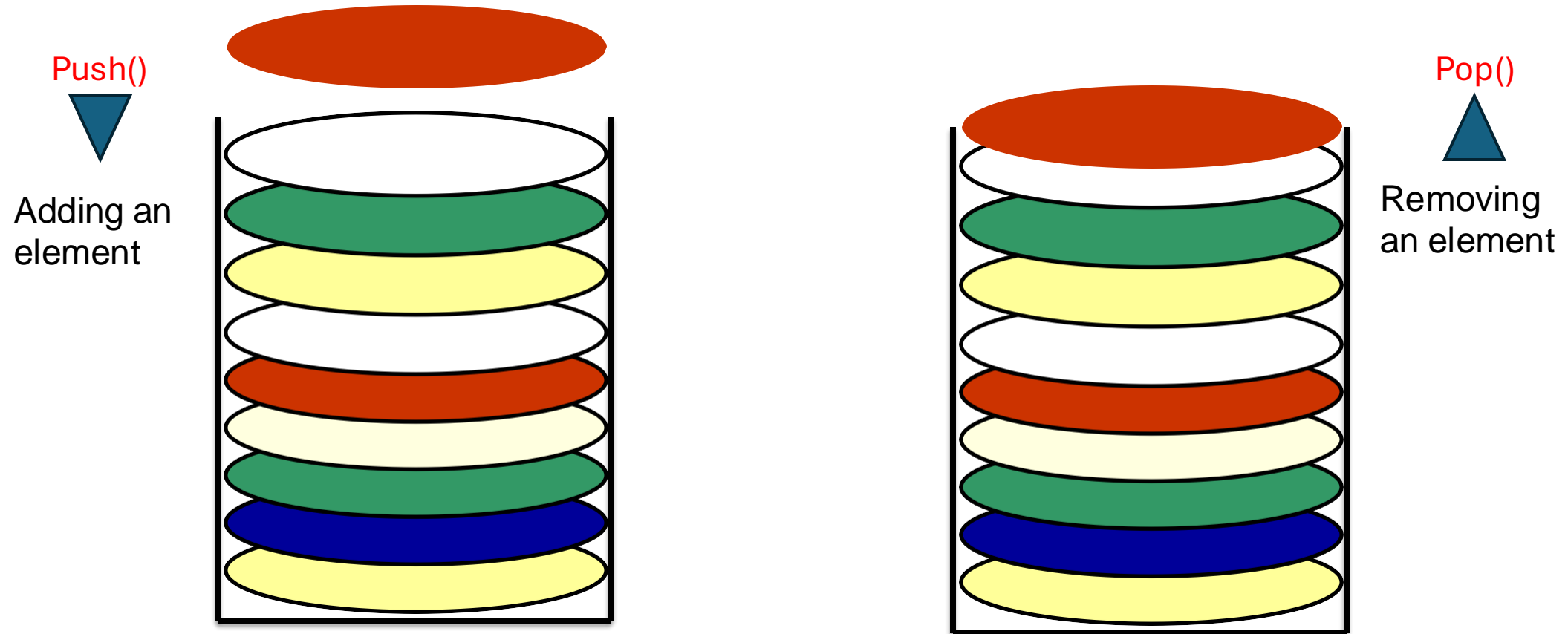
- A stack is a collection of items organized so that only the item at the top is accessible.
- You can visualize it as a container open at just one end, with items stacked on top of each other.
- Each time you add an item, it is added to the previous ones, and when you remove an item, only the one on top is accessible.





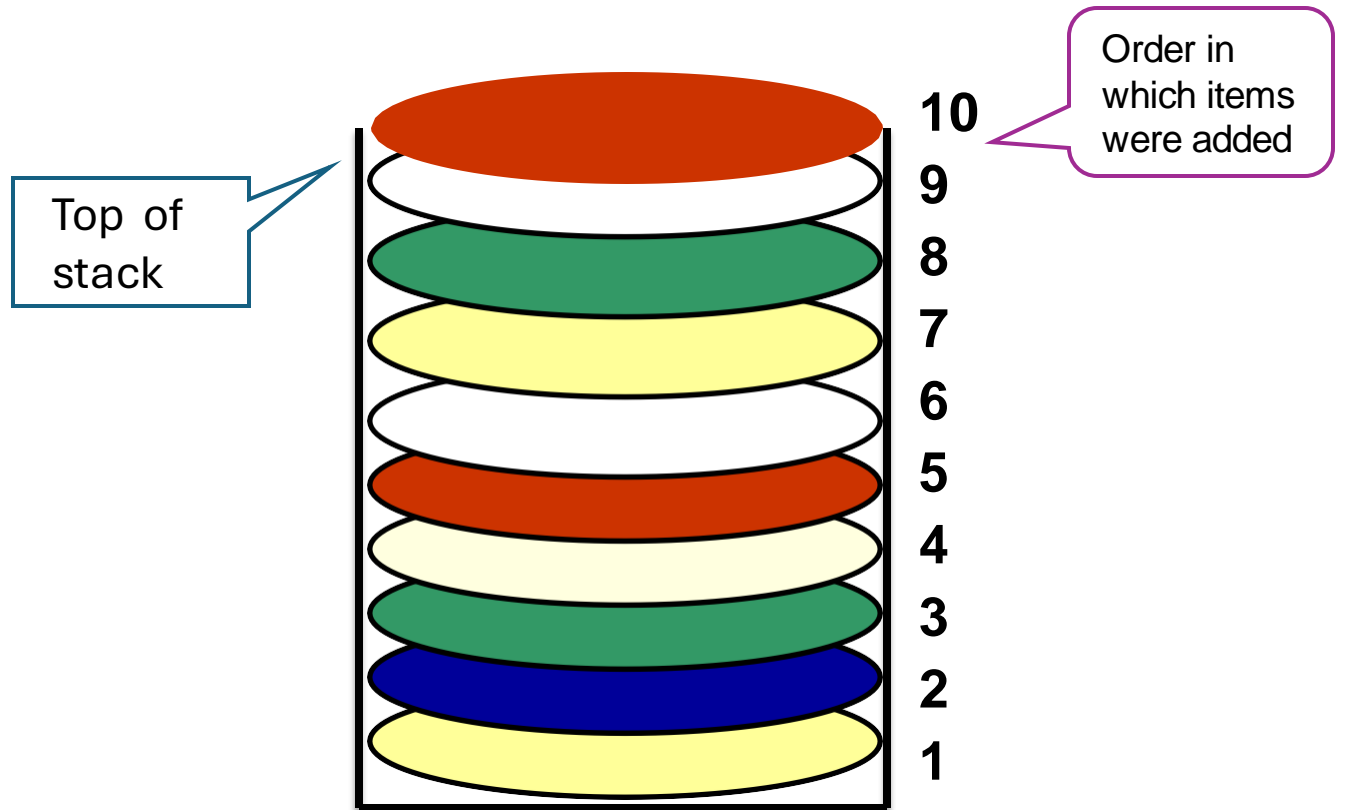
# The Stack ADT Operations

---



# A Stack is a LIFO structure

- A stack is a collection whose elements are added and removed from one end, called the top of the stack
- Stack is a LIFO (Last In, First Out) structure



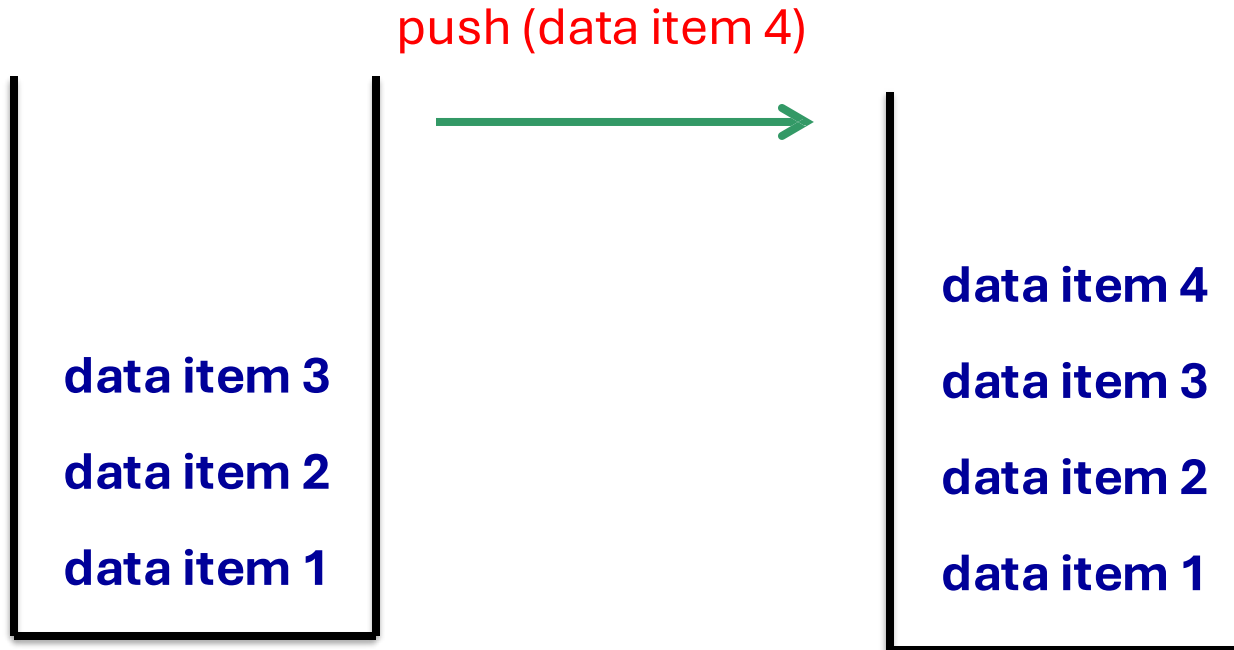
# Stack Operations

A thick, hand-drawn style orange line that underlines the title "Stack Operations". It starts under the 'S' and ends under the 's', with a slightly wavy, irregular edge.

# Stack Operations: Push

---

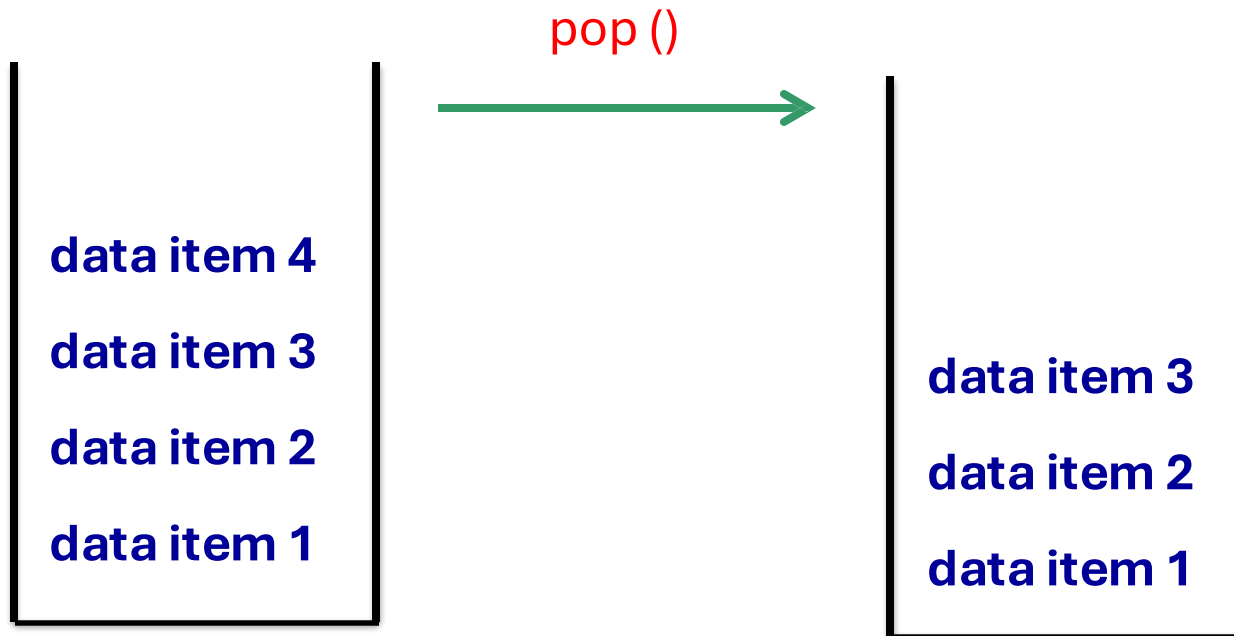
- **Push**: adds an element at the top of the stack



# Stack Operations: Peek

---

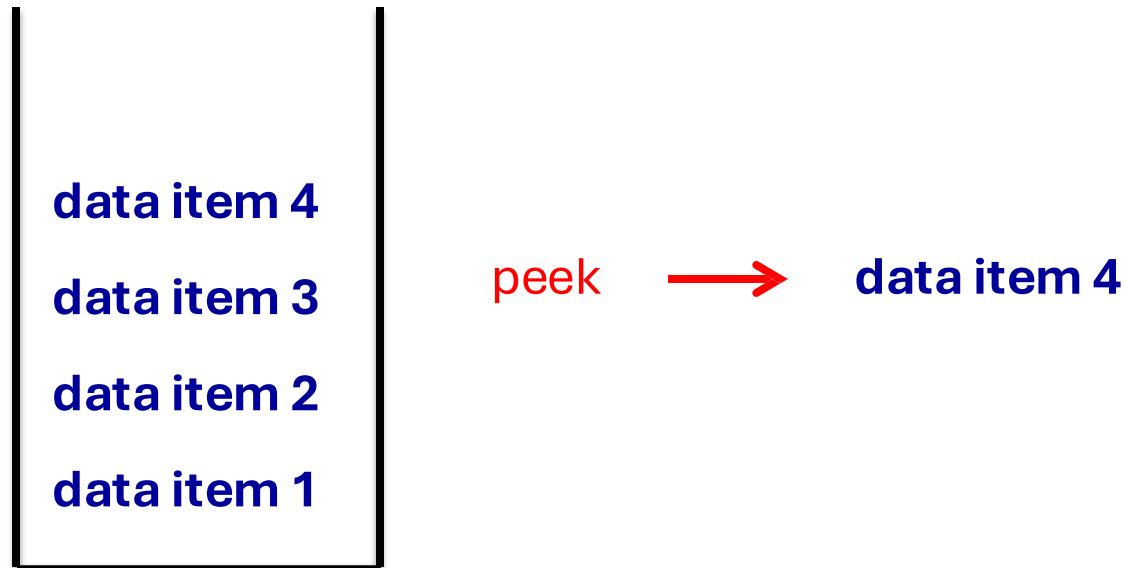
- **Peek:** returns the element at the top of the stack without removing it



# Stack Operations: Pop

---

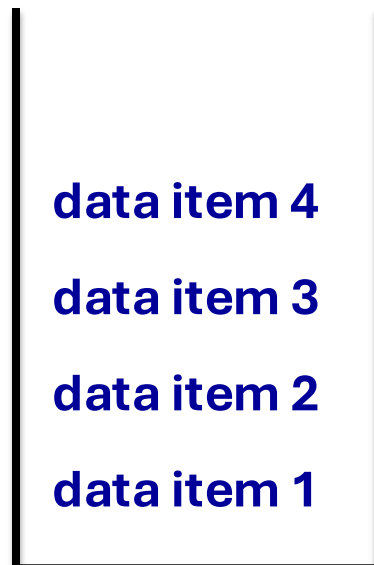
- **Pop**: removes the element at the top of the stack



# More Stack Operations

---

- **size**: number of elements in the stack
- **isEmpty**: true if the stack is empty
- **toString**: string representation of stack



**size** → 4  
**isEmpty** → false  
**toString** → "Stack: data item 4  
data item 3 data item2 data item  
1"



# Definition of the Stack ADT

---

- Stack Abstract Data Type (Stack ADT)
- It is a **collection of data** together with the following **operations**:
  - push
  - pop
  - peek
  - Size
  - isEmpty
  - toString

```
public interface StackADT<T>
{
    public void push (T dataItem); // Adds one // data item to the
    // top of this stack

    public T pop( ); // Removes and returns the
    // top data item of this stack

    public T peek( ); // Returns the top data
    // item of this stack

    public boolean isEmpty( ); // Returns true
    // if this stack is empty

    public int size( ); // Returns the number of data items in this
    stack

    public String toString( ); // Returns a
    // string representation of this stack
}
```

# Implementing an Interface

- Remember that we cannot create an object of type StackADT because it is an interface.
- To be able to initialize stack objects, we first need to create a class that implements the interface by providing the code for each of the abstract methods
- Example:

```
StackADT<Integer> S; // valid
```

only declaring a reference variable of type

```
S = new StackADT<Integer>(); // invalid
```

compilation error

```
S = new ClassThatImplementsStackADT<Integer>(); // valid
```

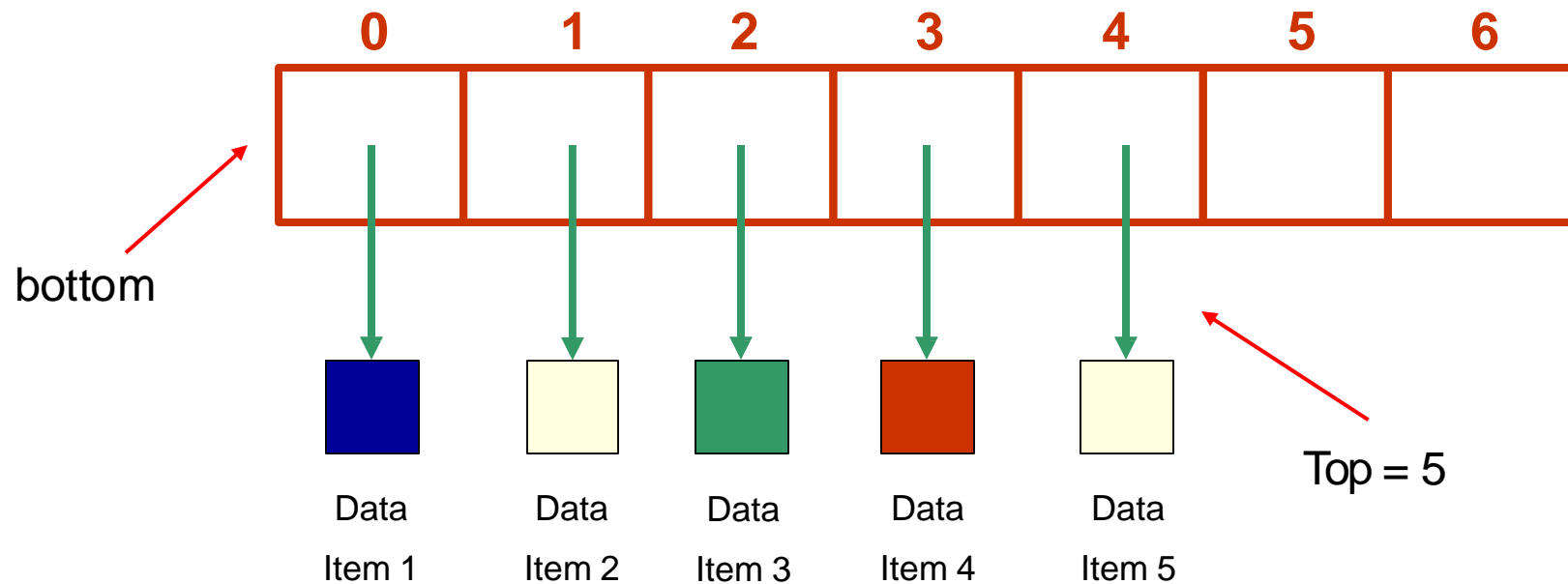
point to an actual object of a class that implements StackADT

# Stack Implementation Issues

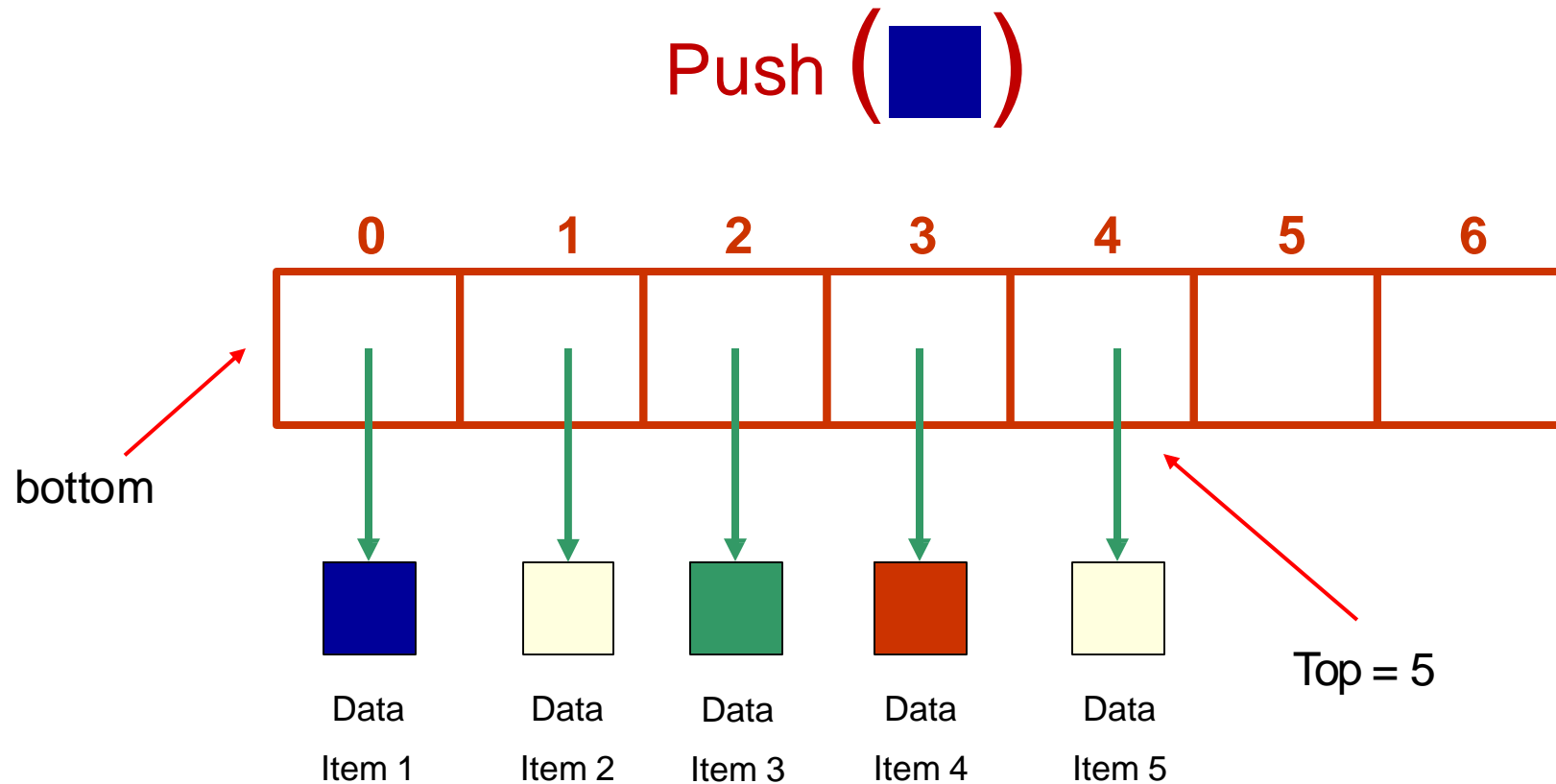
- What do we need to implement a stack?
- A data structure to store the data items.
- For example:
  - Arrays
  - Linked Lists
- A way to indicate the top and bottom of the stack

# Array Implementation of a Stack

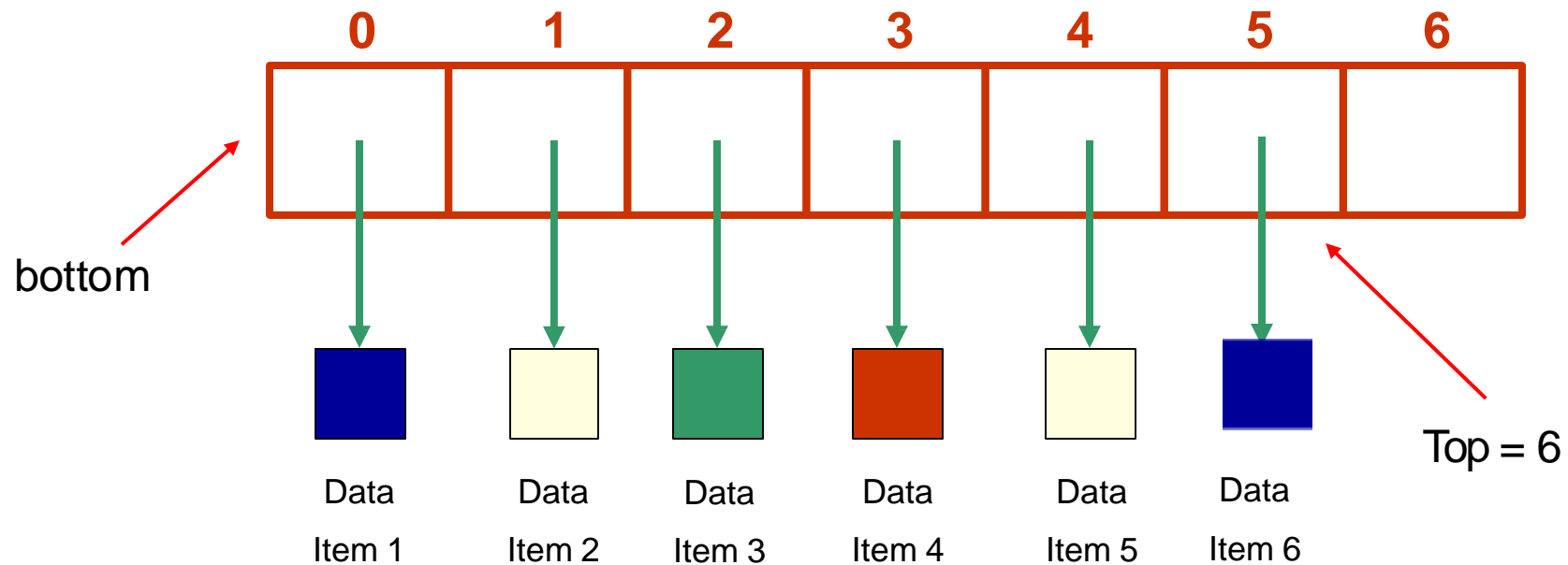
- We do not need to keep track of the bottom, as the bottom of the stack is always at index 0



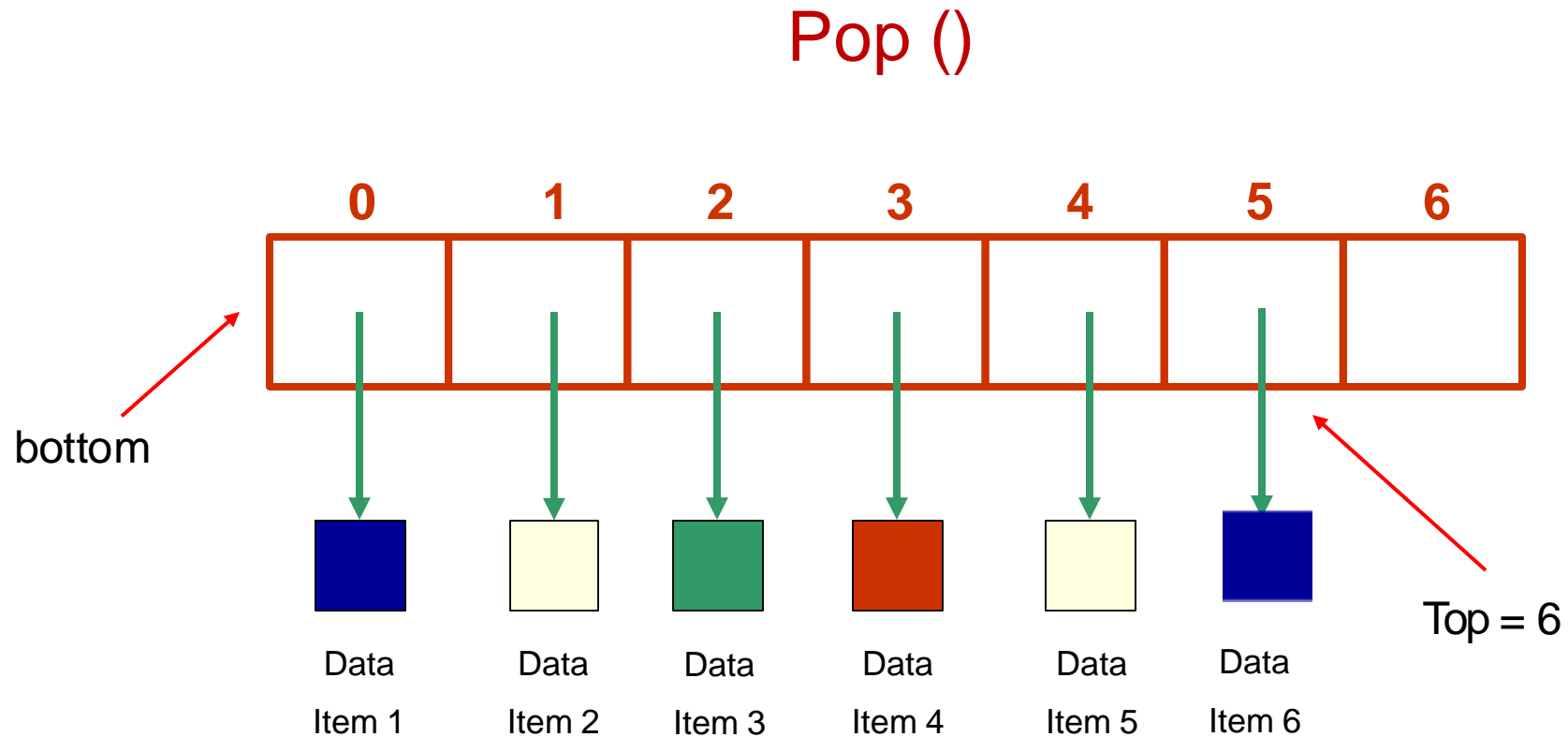
# Push New Data Element



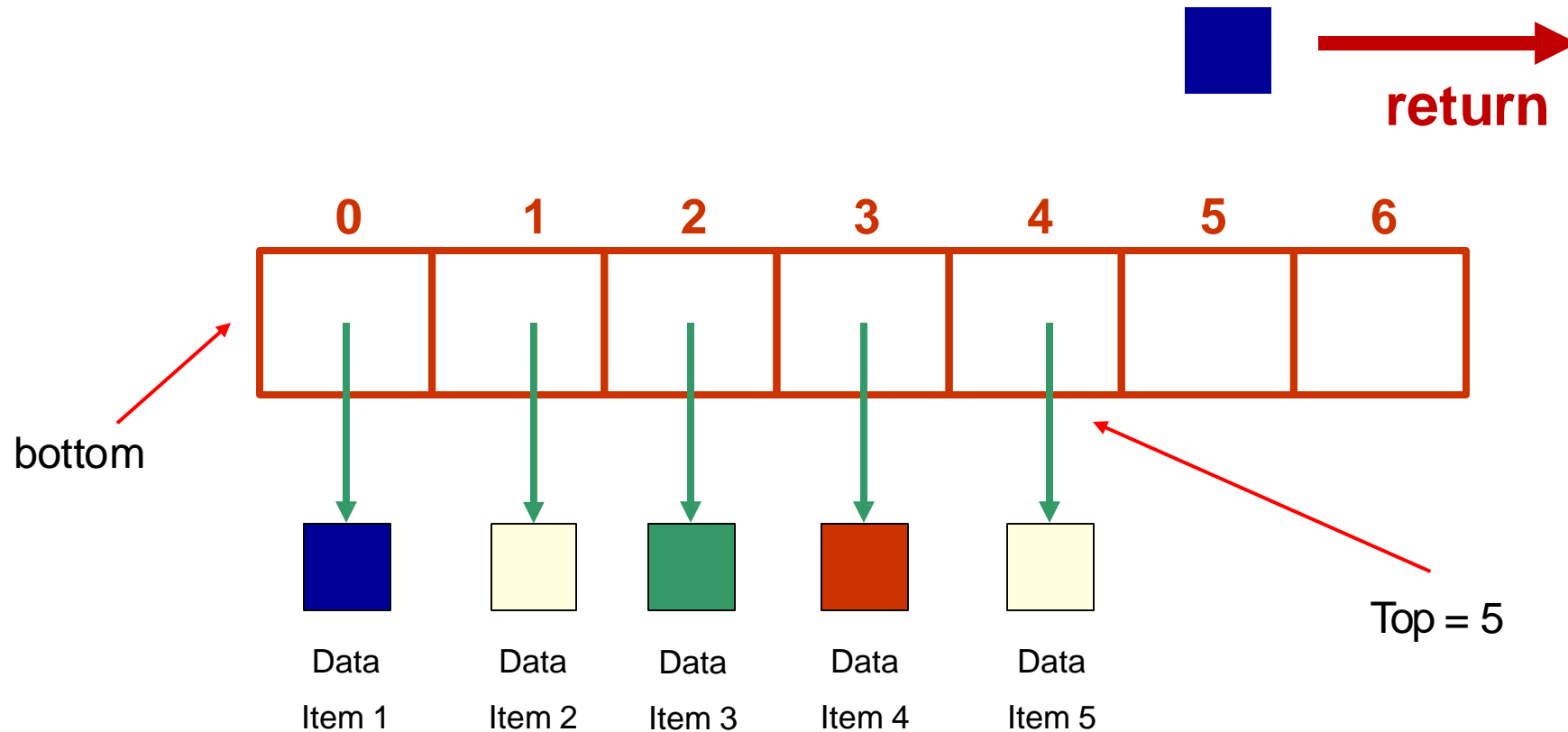
# Push New Data Element cont.



# Pop a Data Element



# Pop a Data Element cont.





# Array Implementation of a Stack cont.

---

- An interface is implemented by a Java class using the keyword implements.

```
public class ArrayStack<T> implements StackADT<T> {  
    private T[ ] stack; // Array for the data  
    private int top; // Top of stack  
    private final int DEFAULT_CAPACITY=100;
```

- Since type **T** is unknown at compile time, how do we allocate memory to the array **stack**?

# Array Implementation of a Stack cont.

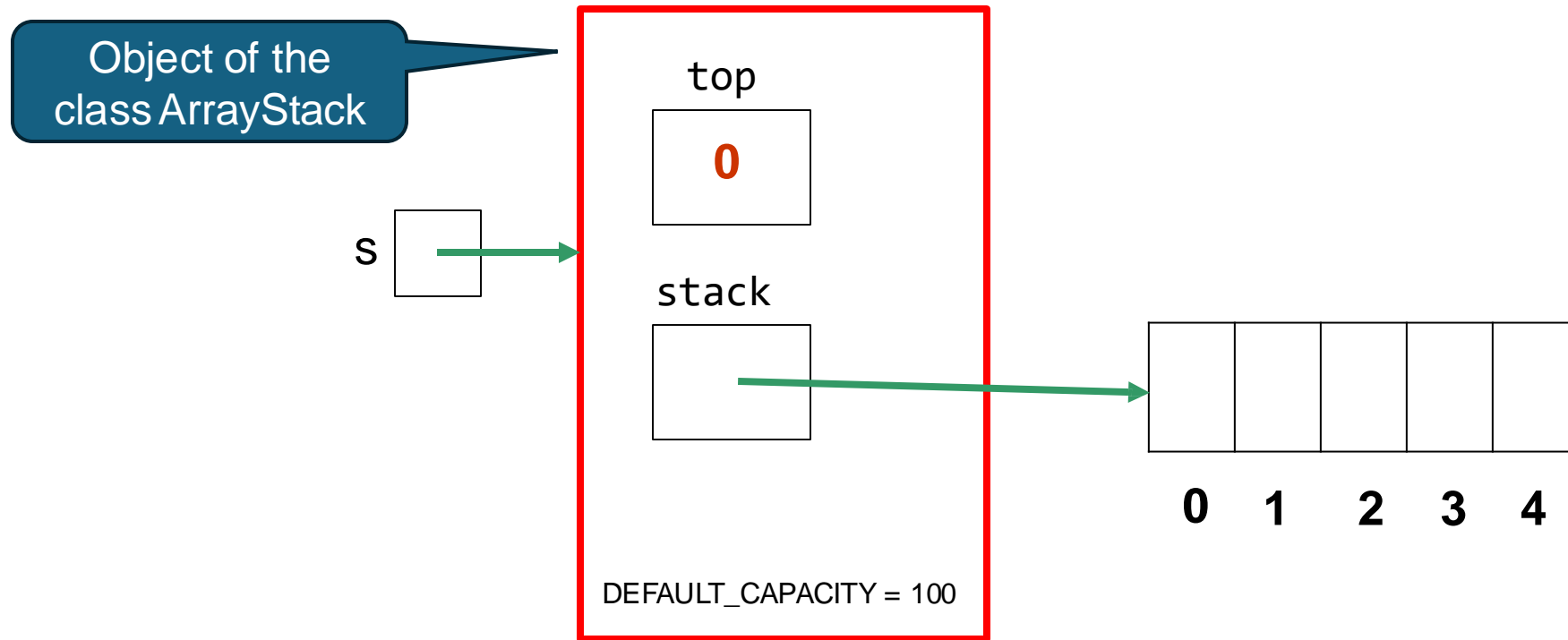
---

- An interface is implemented by a Java class using the keyword implements.

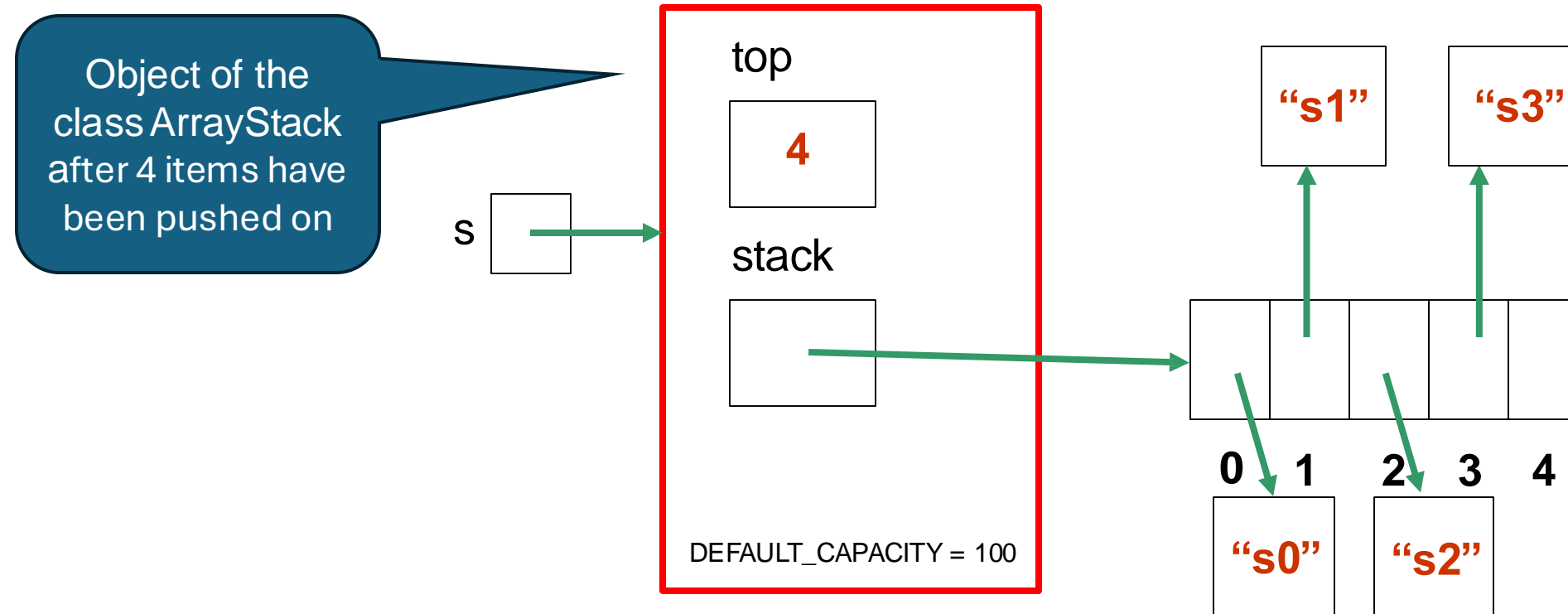
```
public class ArrayStack<T> implements StackADT<T> {  
    private T[ ] stack; // Array for the data  
    private int top; // Top of stack  
    private final int DEFAULT_CAPACITY=100;  
    public ArrayStack( ) {  
        top = 0;  
        stack = (T[]) (new Object[DEFAULT_CAPACITY]);  
    }  
    public ArrayStack (int initialCapacity) {  
        top = 0;  
        stack = (T[]) (new Object[initialCapacity]);  
    }  
}
```

# Example of Creating a Stack of Strings

```
ArrayStack<String> s = new ArrayStack<String>(5);
```



# Example of Creating a Stack of Strings



# Implementation of Push & Pop Functions

---

```
public class ArrayStack<T> implements
StackADT<T> {
    private T[ ] stack; // Array for the data
    private int top; // Top of stack
    private final int DEFAULT_CAPACITY=100;
    public ArrayStack( ) {
        top = 0;
        stack = (T[]) (new
            Object[DEFAULT_CAPACITY]);
    }
    public ArrayStack (int initialCapacity) {
        top = 0;
        stack = (T[ ]) (new
            Object[initialCapacity]);
    }
}
```

```
// Adds the specified data item to the
// top of the stack, expanding the
// capacity of the stack array if
// necessary
public void push (T dataItem) {
    if (top == stack.length) expandCapacity( );
    stack[top] = dataItem;
    top++;
}
public T pop() {
    top--;
    T item = stack[top];
    stack[top] = null;
    // Clear the reference for garbage collection
    return item;
}
```

# More Implementation

---

```
// Returns the data item at the top of the stack. Throws an
// EmptyCollectionException if the stack is empty.
// Returns the data item at the top of the stack without removing it
public T peek() throws EmptyCollectionException {
    if (top == 0) throw new EmptyCollectionException("Empty stack");
    return stack[top - 1];
}

// Returns the number of data items in the stack
public int size() {return top;}

// Returns true if the stack is empty and false otherwise
public boolean isEmpty() { return (top == 0);}
```

# More Implementation

---

```
// Returns a string representation of this stack
public String toString() {
    String result = "Stack:\n";
    for (int index = 0; index < top; index++) {
        result = result + stack[index].toString() + "\n";
    }
    return result;
}

// Expands the capacity of the stack array if necessary
private void expandCapacity() {
    T[] larger = (T[]) (new Object[stack.length * 2]);
    System.arraycopy(stack, 0, larger, 0, stack.length);
    stack = larger;
}
```



Thank you