Topic 3

The Stack ADT

Objectives

- Define a stack collection
- Use a stack to solve a problem
- Examine an array implementation of a stack

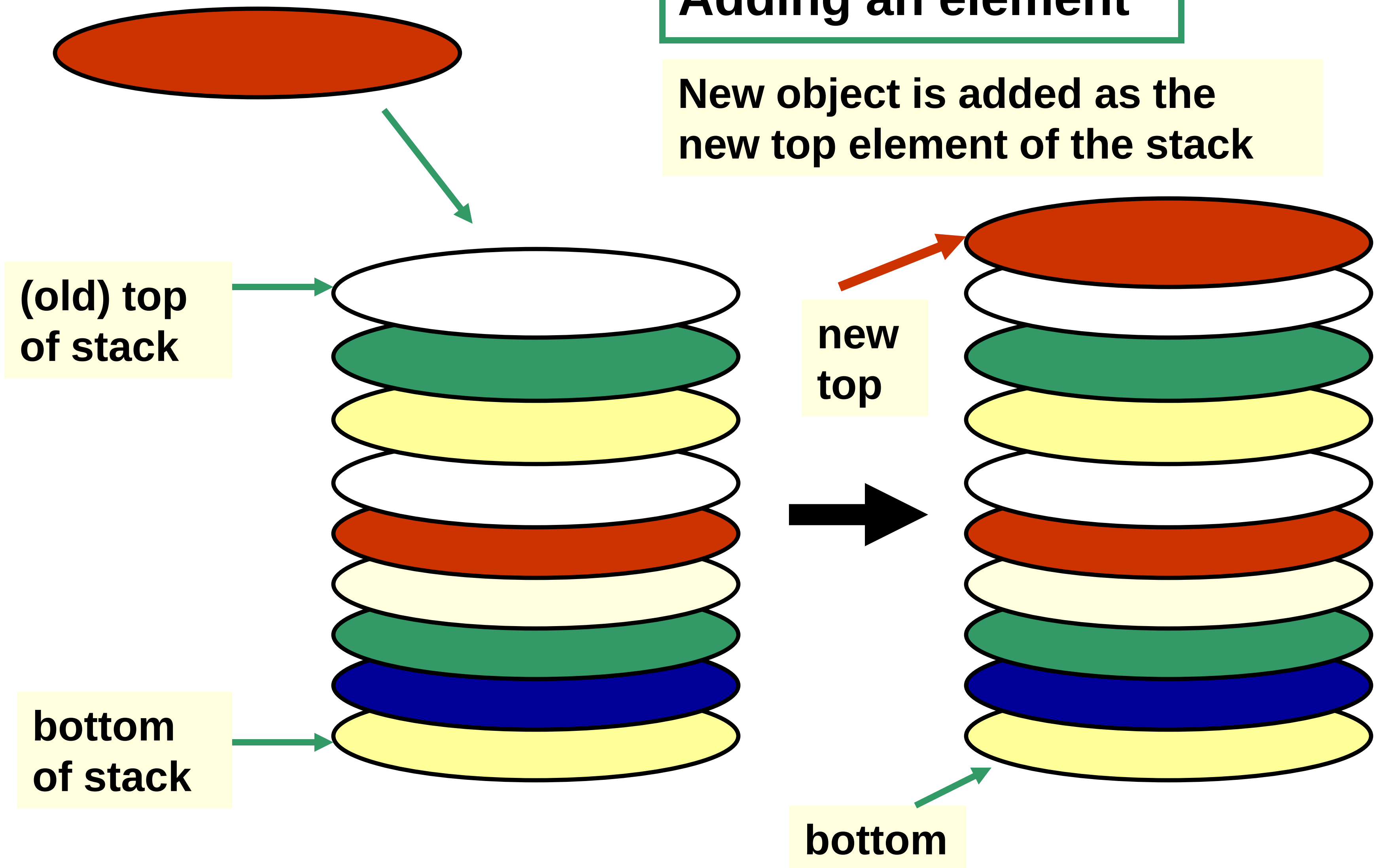
Stacks

- ***Stack*** **:** 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) data structure
- ***Examples***:
  - A stack of plates – what can we do with the elements of this collection?
  - Other real-world examples of stacks?

Conceptual View of a Stack

Adding an element

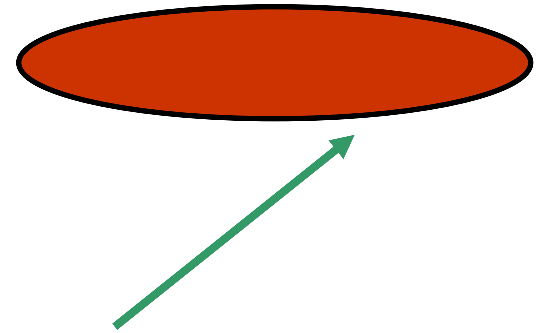
New object is added as the new top element of the stack

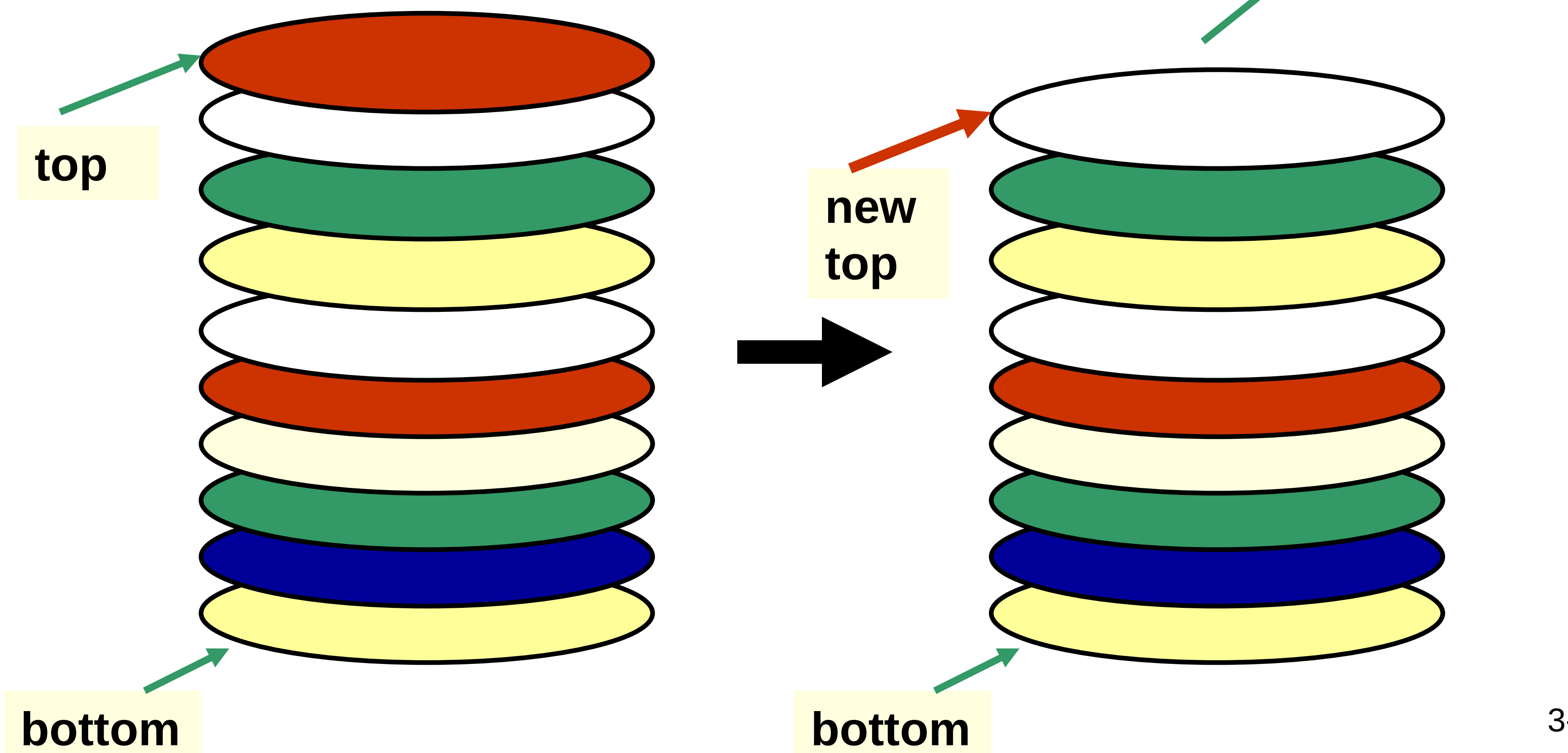


Conceptual View of a Stack

Removing an element

Object is removed from the top of the stack





Uses of Stacks in Computing

Useful for any kind of problem involving ***LIFO*** data

- ***Backtracking***: in puzzles and games
- ***Browsers***
  - To keep track of pages visited in a browser tab

Uses of Stacks in Computing

- ***Word Processors, editors***
  - To check expressions or strings of text for matching parentheses / brackets
    e.g. if (a == b)
    { c = (d + e) * f;
    }
  - To implement *undo* operations
    - Keeps track of the most recent operations
- ***Markup languages*** (***e.g.*** HTML, XML): have formatting information (***tags***) as well as raw text
  - To check for matching tags
    e.g. **<HEAD>**
    <TITLE>Computer Science 1027a</TITLE>
    </HEAD>

Uses of Stacks in Computing

- ***Stack Calculators***
  - To convert an ***infix*** expression to ***postfix***, to make evaluation easier (more on this later)

    Infix expression: a * b + c

    Postfix expression: a b * c +
  - To evaluate postfix expressions (ditto)
- ***Compilers***
  - To convert infix expressions to postfix, to make translation of a high-level language such as Java or C to a lower level language easier

Uses of Stacks in Computing

- ***Call stack (Runtime stack)***
  - Used by runtime system when methods are invoked, for method call / return processing (more on this later)
    - e.g. main calls method1
      method1 calls method 2
      method 2 returns …
  - Holds "*call frame*" containing local variables, parameters, etc.
  - Why is a stack structure used for this?

Operations on a Collection

- Every collection has a set of operations that define how we interact with it, for example:
  - Add elements
  - Remove elements
  - Determine if the collection is empty
  - Determine the collection's size

Stack Operations

- ***push***: add an element at the top of the stack
- ***pop***: remove the element at the top of the stack
- ***peek***: examine the element at the top of the stack

- It is ***not*** legal to access any element other than the one that is at the top of the stack!

Operations on a Stack

Operation	Description
push	Adds an element to the top of the stack
pop	Removes an element from the top of the stack
peek	Examines the element at the top of the stack
isEmpty	Determines whether the stack is empty
size	Determines the number of elements in the stack
toString	Returns a string representation of the stack

Discussion

- Do the operations defined for the stack have anything to do with Java?
- Do they say what the stack is used for?
- Do they say how the stack is stored in a computer?
- Do they say how the operations are implemented?

Stack ADT

- ***Stack Abstract Data Type (Stack ADT)***
  - It is a ***collection*** of data
  - Together with the ***operations*** on that data
    - We just discussed the operations

Java Interfaces

- Java has a ***programming construct*** called an ***interface*** that we use to *formally* define what the operations on a collection are in Java
- ***Java interface***: a list of ***abstract methods*** and constants
  - Must be public
  - Constants must be declared as **final static**

Java Interfaces

- ***Abstract method*** : a method that does not have an implementation, i.e. it just consists of the *header* of the method:

  return type **method name** (parameter list)

Java Interface for Stack ADT

```
public interface StackADT<T>
{
  //  Adds one element to the top of this stack
  public void push (T element);
  //  Removes and returns the top element from this stack
  public T pop( );
  //  Returns without removing the top element of this stack
  public T peek( );
  //  Returns true if this stack contains no elements
  public boolean isEmpty( );
  //  Returns the number of elements in this stack
  public int size( );
  //  Returns a string representation of this stack
  public String toString( );
}
```

Generic Types

What is this <T> in the interface definition?

- It represents a ***generic type***
  - For generality, we can define a class (or interface) based on a generic type rather than an actual type
  - Example: we define a Stack for objects of type T
- The ***actual type*** is known only when an application program creates an object of that class
  - Examples:
    - in a card game: a Stack of Card objects
    - in checking HTML tags: a Stack of Tag objects

Generic Types

- Note: it is merely a convention to use **T** to represent the generic type
- In the class definition, we enclose the generic type in angle brackets: **< T >**

Implementing an Interface

- One or more classes can ***implement an interface***, perhaps differently
  - A class ***implements the interface*** by providing the implementations (bodies) for each of the abstract methods
  - Uses the reserved word **implements** followed by the interface name
- We will see Stack ADT *implementation* examples soon … but first we will look at *using* a stack

Using a Stack: Postfix Expressions

- Normally, we write expressions using ***infix notation***:
  - Operators are between operands: 3 + 4 * 2
  - Parentheses force precedence: (3 + 4) * 2
- In a ***postfix expression***, the operator comes ***after*** its two operands
  - Examples above would be written as:

    3 4 2 * +

    3 4 + 2 *
- What is the advantage of writing expressions in postfix form?

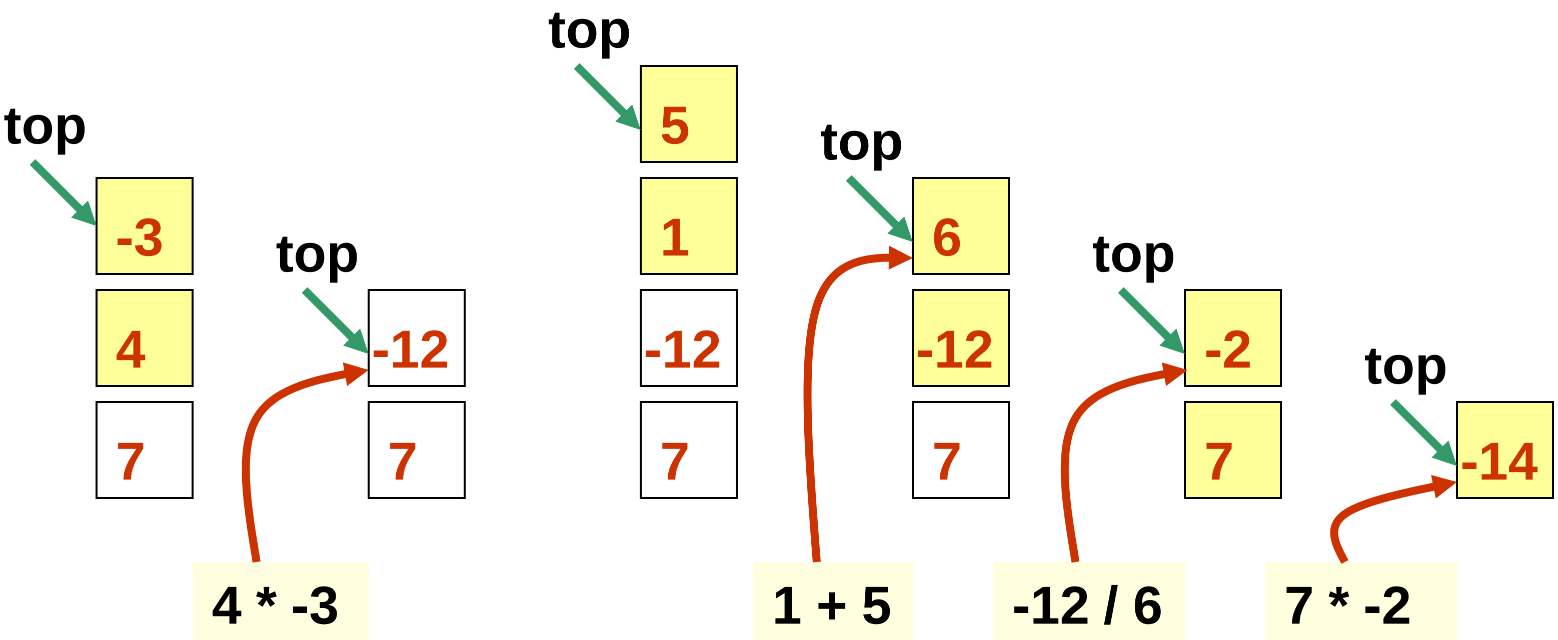
Evaluating Postfix Expressions

- *Algorithm to evaluate a postfix expression*:
  - Scan from left to right, determining if the next token or symbol is an operator or operand
  - If it is an operand, push it on the stack
  - If it is an operator, pop the stack twice to get the two operands, perform the operation, and push the result back onto the stack
- Try the algorithm on our examples …
- At the end, there will be one value on the stack – what is it?

Using a Stack to Evaluate a Postfix Expression

Evaluation of

7 4 -3 * 1 5 + / *



At end of evaluation, the result is the only item on the stack

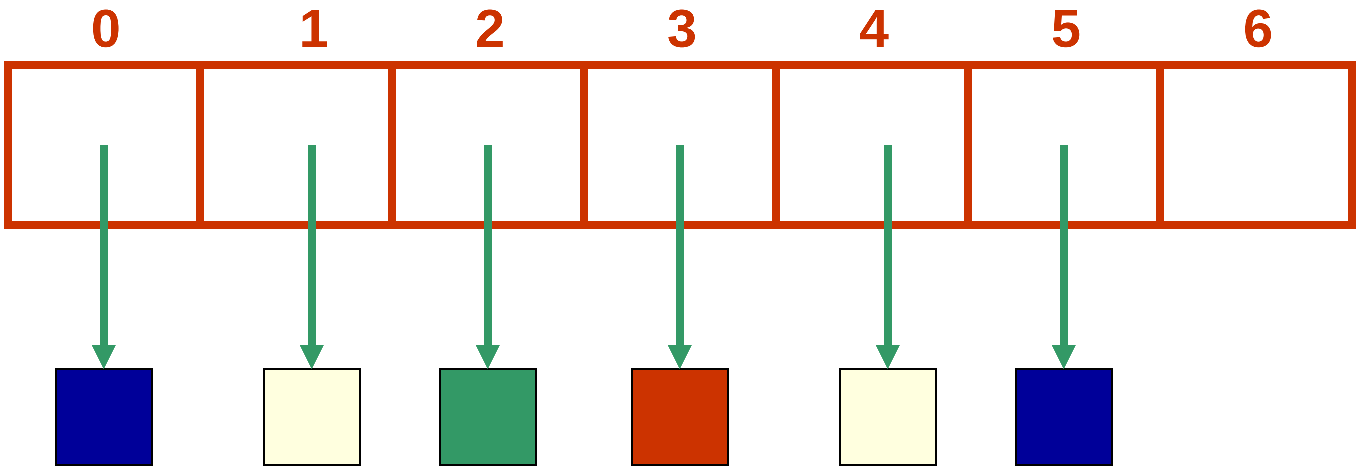
Java Code to Evaluate Postfix Expressions

- For simplicity, assume the operands in the expressions are integer
- See ***Postfix.java***
  - Reads postfix expressions and evaluates them
- See ***PostfixEvaluator.java***
  - The postfix expression evaluator
  - Note that it uses a class called **ArrayStack**, which is an implementation of the Stack ADT that we will now examine
    - We will see later that it could just as well have used a different implementation of the Stack ADT!

Implementing a Stack

- Does an application program need to know ***how*** the Stack collection is implemented?
  - No - we are using the Stack collection for its *functionality (**what**);* how it is implemented is not relevant
- The Stack collection could be implemented in various ways; let's first examine how we can use an ***array***

An Array of Object References



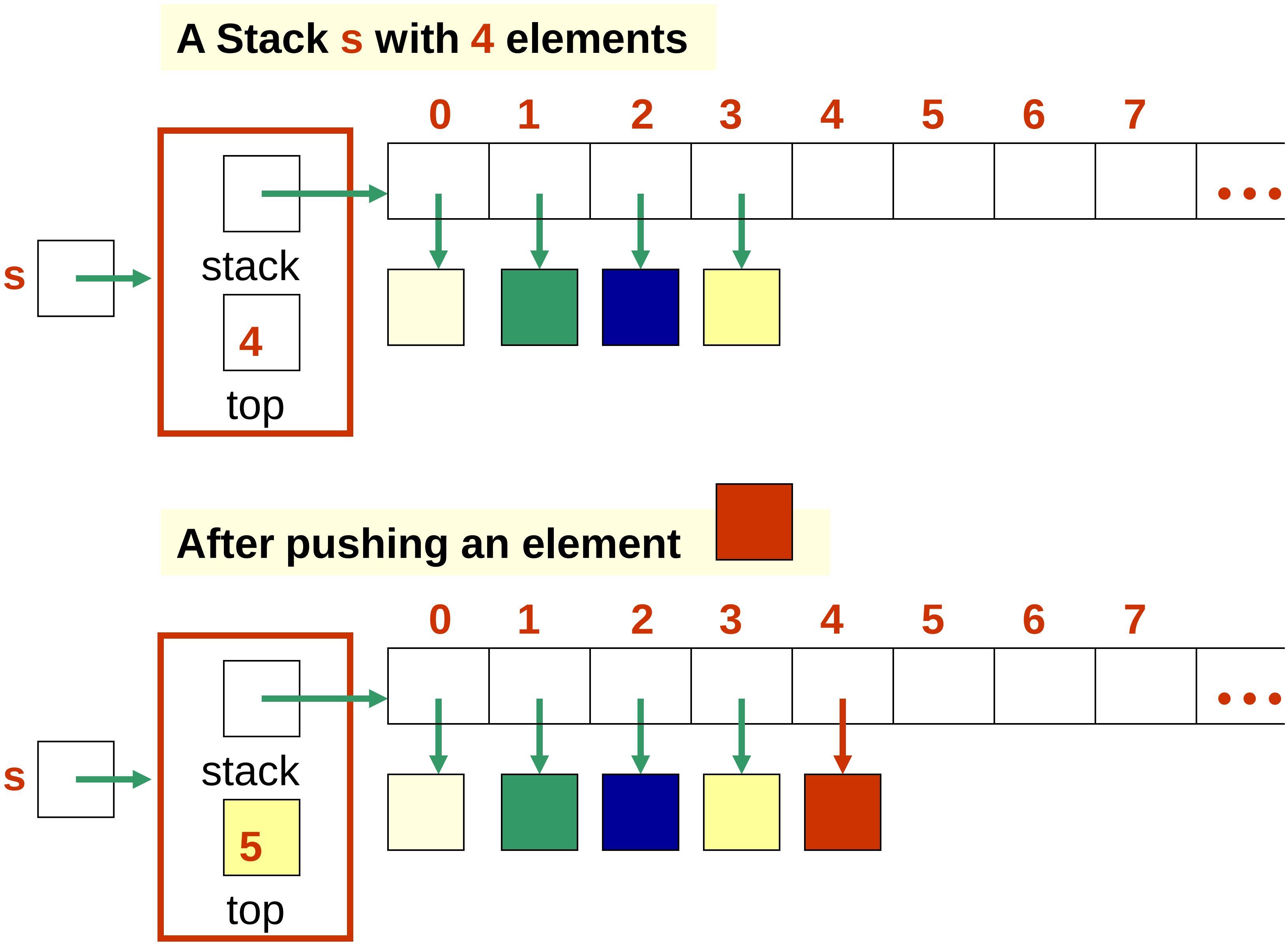
Stack Implementation Issues

- What do we need to implement a stack?
  - A data structure (***container***) to hold the data elements
  - Something to indicate the ***top*** of the stack

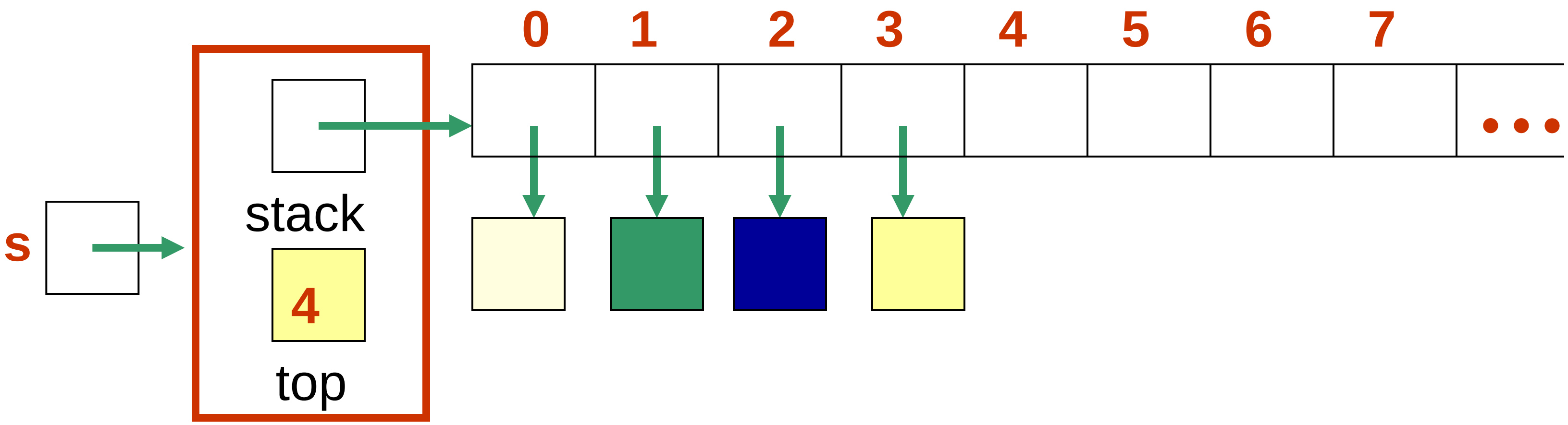
Array Implementation of a Stack

- Our container will be an ***array*** to hold the data elements
  - Data elements are kept contiguously at one end of the array
- The top of the stack will be indicated by its position in the array (***index***)
  - Let's assume that the bottom of the stack is at index 0
  - The top of the stack will be represented by an integer variable that is the index of the ***next available slot*** in the array

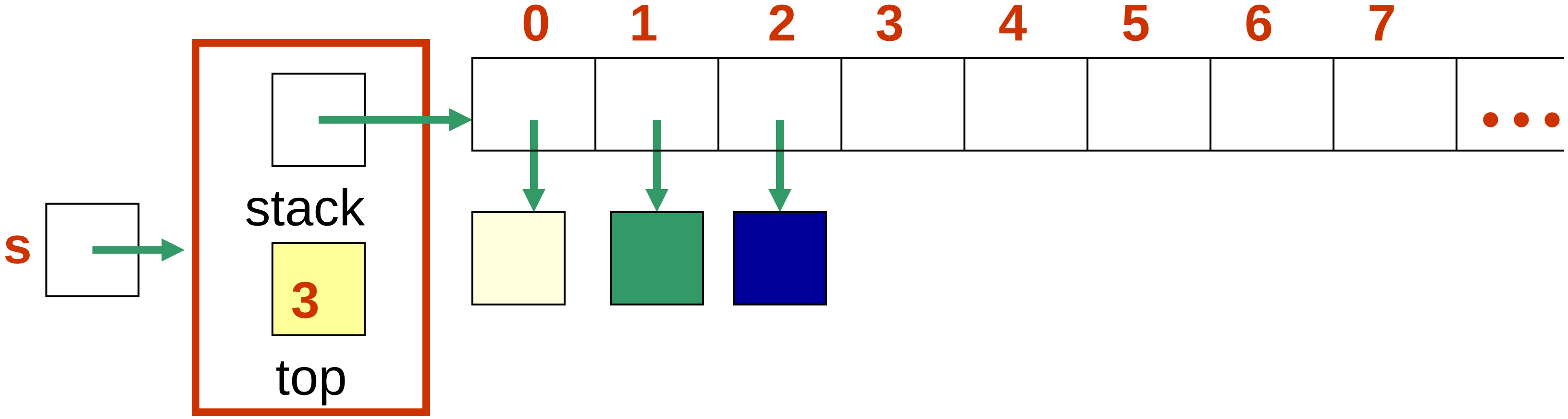
Array Implementation of a Stack



After popping one element



After popping a second element



Java Implementation

- The array variable **stack** holds references to objects
  - Their type is determined when the stack is instantiated
- The integer variable **top** stores the index of the *next available slot* in the array
  - What else does **top** represent?

The ArrayStack Class

- The class **ArrayStack** implements the **StackADT** interface:

  public class ArrayStack<T> implements StackADT<T>

In the ***Java Collections API***, class names indicate both the underlying data structure and the collection

- We will adopt the same naming convention: the **ArrayStack** class represents an ***array*** implementation of a ***stack*** collection

ArrayStack Data Fields

- ***Attributes*** *(instance variables)*:

  private T[ ] stack;   // the container for the data

  private int top;   // indicates the next open slot

  - Note that these were ***not*** specified in the Java interface for the StackADT (why not?)

- There is also a private constant (see later)
  private final int DEFAULT_CAPACITY=100;

```java
//-----------------------------------------------------------------
//  Creates an empty stack using the default capacity.
//-----------------------------------------------------------------
public ArrayStack( )
{
   top = 0;
   stack = (T[ ]) (new Object[DEFAULT_CAPACITY]);
}
```

ArrayStack constructors

```java
//-----------------------------------------------------------------
//  Creates an empty stack using the specified capacity.
//-----------------------------------------------------------------
public ArrayStack (int initialCapacity)
{
   top = 0;
   stack = (T[ ]) (new Object[initialCapacity]);
}
```

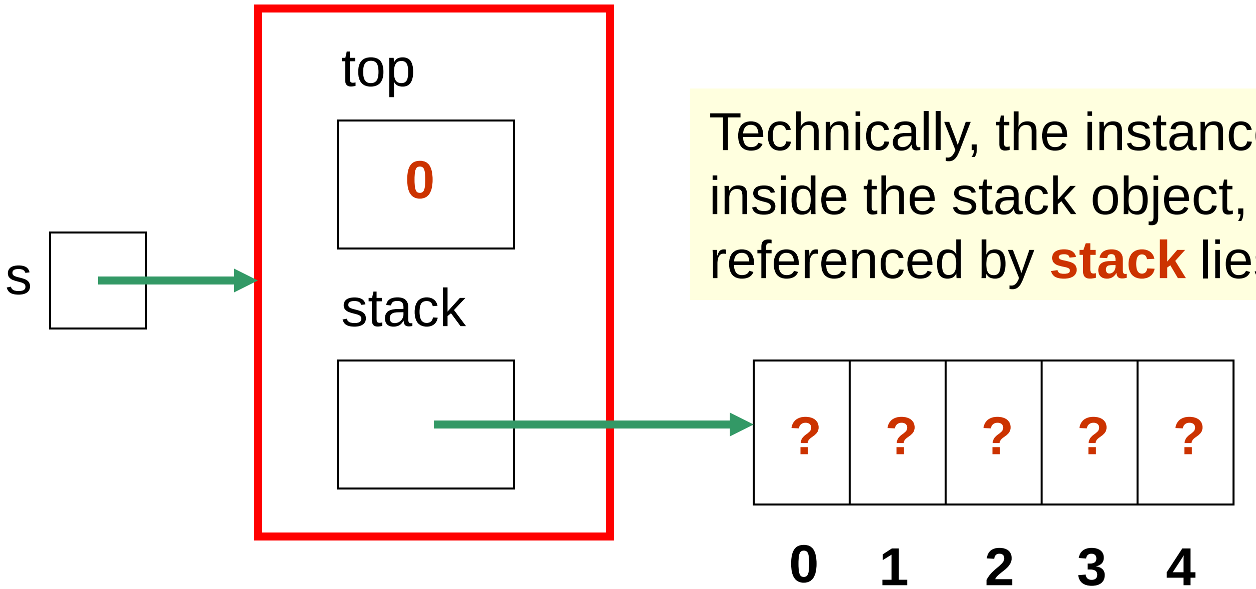
ArrayStack Constructors

- ***Note:*** constructors are ***not*** specified in the Java interface for the StackADT (why not?)
- What is the purpose of **(T[ ])** ?
  - The elements of the **stack** array are of generic type **T**
    - ***Recall:*** we can't instantiate anything of a generic type
    - So, we need to instantiate an element of type **Object** and cast it into the type **T**
  - Specifically, we are ***casting*** an array of **Object** objects into an array of type **T**

Example of using Constructor to create a Stack of Numbers

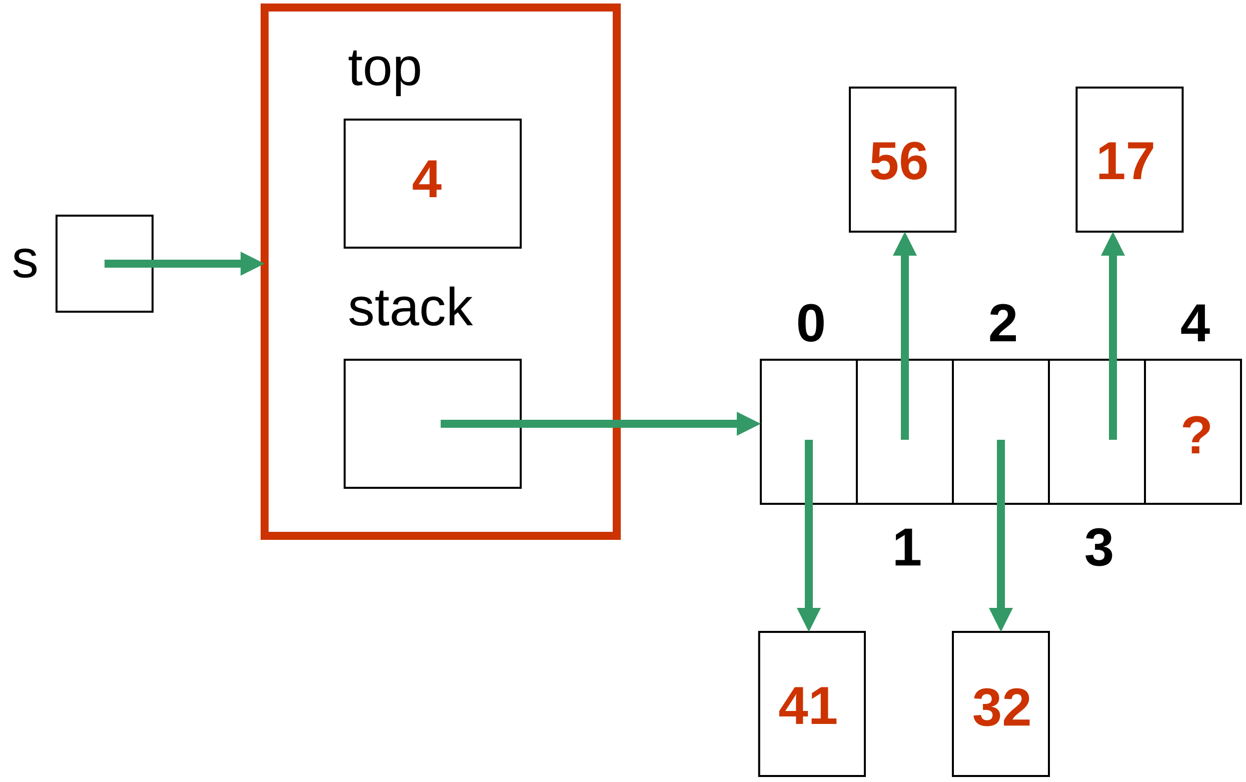
What happens in memory when an ArrayStack object is created using the following statement?

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



Technically, the instance variables lie inside the stack object, but the array referenced by **stack** lies outside it

Example: the same **ArrayStack** object after four items have been pushed on



```
//-----------------------------------------------------------------
//  Adds the specified element to the top of the stack,
//  expanding the capacity of the stack array if necessary
//-----------------------------------------------------------------
public void push (T element)
{
   if (top == stack.length)
      expandCapacity( );

   stack[top] = element;
   top++;
}
```

The push( ) operation

Where in the array is the element added?

Managing Capacity

- An array has a particular number of cells when it is created (its ***capacity***), so the array's capacity is also the stack's capacity
- What happens when we want to push a new element onto a stack that is full, ***i.e.*** add it to an array that is at capacity?

1. The **push** method could throw an exception
2. It could return some kind of status indicator (***e.g.*** a boolean value **true** or **false**, that indicates whether the push was successful or not)
3. It could *automatically* expand the capacity of the array

Discussion

- What are the implications to the class ***using*** the stack, of each of the three solutions?

```
//-----------------------------------------------------------------
//  Helper method to create a new array to store the
// contents of the stack, with twice the capacity
//-----------------------------------------------------------------
private void expandCapacity( )
{
   T[ ] larger = (T[ ]) (new Object[stack.length*2]);

   for (int index=0; index < stack.length; index++)
      larger[index] = stack[index];

   stack = larger;
}
```

The expandCapacity( ) helper method

```java
//-----------------------------------------------------------------
//  Removes the element at the top of the stack and returns a
//  reference to it. Throws an EmptyCollectionException if the
//  stack is empty.
//-----------------------------------------------------------------
public T pop( ) throws EmptyCollectionException
{
   if (isEmpty( ))
      throw new EmptyCollectionException("Stack" );
   top--;
   T result = stack[top];
   stack[top] = null;
   return result;
}
```

The pop( ) operation

Note the order: decrement top before getting element from array (why?)

Stack Exceptions

- What happens if the user of the Stack collection attempts to pop an element from an empty stack?
  - The designer of the implementation determines how it will be handled:
    - The ***user*** of the stack could check beforehand, using the **isEmpty** method
    - Here, the **pop** method throws an ***exception*** if the stack is empty
      - In this case the user of the stack can deal with the problem (using a ***try/catch***)

```java
//---------------------------------------------------------------------
//  Returns a string representation of this stack.
//---------------------------------------------------------------------
public String toString( )
{
   String result = "";

   for (int index=0; index < top; index++)
      result = result + stack[index].toString( ) + "\n";

   return result;
}
```

The toString( ) operation

```java
//-----------------------------------------------------------------------
//  Returns the number of elements in the stack
//-----------------------------------------------------------------------
public int size( )
{
   return top;
}

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

The size( ) operation

The isEmpty( ) operation

Exercise

- Fill in the code for the ***peek*** operation on the next slide

```
//-----------------------------------------------------------------
//  Returns a reference to the element at the top of the stack.
//  The element is not removed from the stack.
//  Throws an EmptyCollectionException if the stack is empty.
//-----------------------------------------------------------------
public T peek( ) throws EmptyCollectionException
{



}
```

The peek( ) operation

Discussion

- At any point, how many elements are there on the stack?
- What is the advantage of having the bottom (rather than the top) of the stack be at ***index 0*** ?
- Can the stack be full?

The **java.util.Stack** Class

- The Java Collections API contains an implementation of a **Stack** class with similar operations
  - It has some additional operations (e.g. **search**, which returns distance from top of stack)
- **Stack** class is derived from the **Vector** class, which has a dynamically "***growable***" array
  - So it has some characteristics that are ***not*** appropriate for a pure stack (e.g. inherited method to ***add item in middle***)