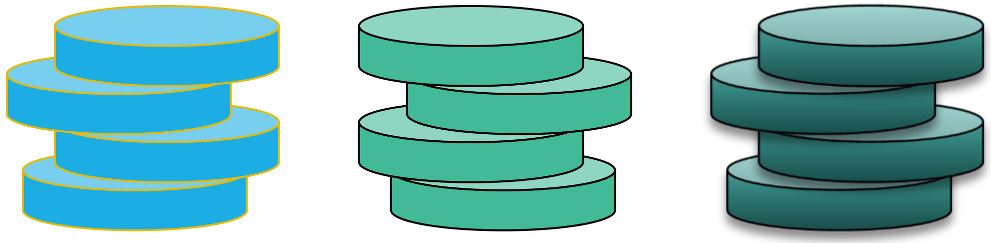
INTERNATIONAL TRAINING INSTITUTE

Week 8: Stacks

4009 DATA STRUCTURES & ALGORITHM (DSA)

Learning Outcomes

- ❖Create, manipulate and understand the data structure of Stacks
- ❖Define and understand the stacks data structure
- ❖Manipulate stacks coding in Java and use of Java's inbuilt methods
- ❖Understand and implement a generic stack code using Arrays in Java
- ❖Understand and implement basic non generic stacks using Arrays in Java
- ❖Understand the importance of generic code in comparison to non-generic code



Content Overview

Data Structure Trade-offs

Data structures	Advantages	Disadvantages
Array	Quick insertion, very fast access to index known	Slow search, slow deletion, fixed size
Ordered Array	Quicker Search than unsorted array	Slow insertion and deletion, fixed size
Stack	Provide Last-in First-out access	Slow access to other items
Queue	Provide First-in First out access	Slow access to other items
Linked List	Quick Insertion, Quick deletion	Slow search
Binary Tree	Quick search, insertion, deletion (if tree balanced)	Deletion algorithm is complex
Hash Table	Very fast access if key known, fast insertion	Slow deletion, access slow if key not known, inefficient memory usage
Heap	Fast insertion, deletion, access to large item	Slow access to other items
Graphs	Model real world situations	Some algorithms are slow and complex

Stack introduction

- A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.
- A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack). The name “stack” is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser.

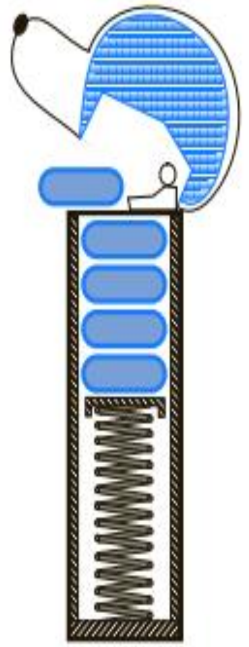


Figure 6.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

Stack introduction

- In this case, the fundamental operations involve the “pushing” and “popping” of plates on the stack. When we need a new plate from the dispenser, we “pop” the top plate off the stack, and when we add a plate, we “push” it down on the stack to become the new top plate.
- Perhaps an even more amusing example is a PEZ ® candy dispenser, which stores mint candies in a spring-loaded container that “pops” out the topmost candy in the stack when the top of the dispenser is lifted (see Figure 6.1)

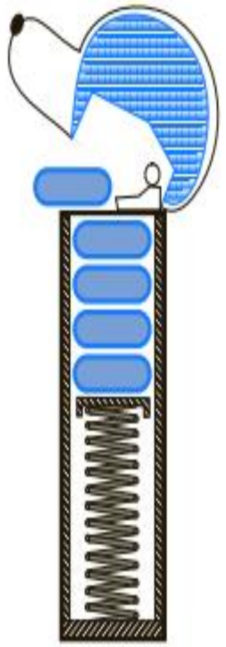


Figure 6.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

Application of Stacks in the real world

- Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site’s address is “pushed” onto the stack of addresses. The browser then allows the user to “pop” back to previously visited sites using the “back” button.

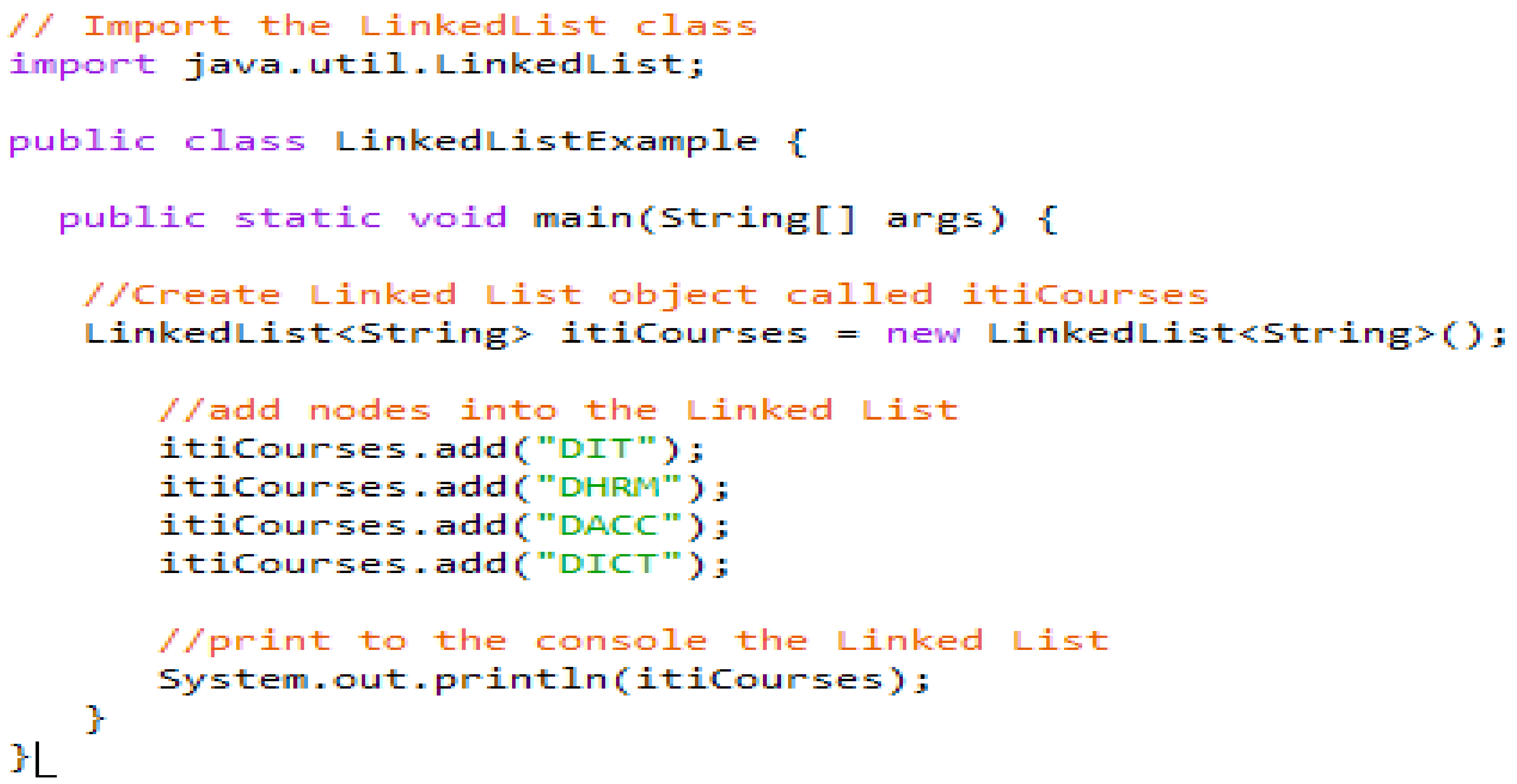
- Text editors or Word processors like Microsoft Word usually provide an “undo” mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.

Basic operations of stacks

1. **Push:** Adds a new element on the stack.
2. **Pop:** Removes and returns the top element from the stack.
3. **Peek or top:** Returns the top element on the stack.
4. **isEmpty:** Checks if the stack is empty.
5. **Size:** Finds the number of elements in the stack.

The stack abstract data type (ADT)

- Stacks are the simplest of all data structures, yet they are also among the most important.
- Stacks are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms.
- A stack is an abstract data type (ADT) that supports the following two update methods:
  - ❑**push(e):** Adds element or inserts an element (e) to the top of the stack.
  - ❑**pop( ):** Removes and returns the top element from the stack (or null if the stack is empty).
  - ❑Additionally, a stack supports the following accessor methods:
  - ❑**top( )** or **peek():** Returns the top element of the stack, without removing it (or null if the stack is empty).
  - ❑**size( ):** Returns the number of elements in the stack.
  - ❑**isEmpty( ):** Returns a boolean indicating whether the stack is empty.
- Note: when using Stacks, we assume that elements added to the stack can have arbitrary type and that a newly created stack is empty

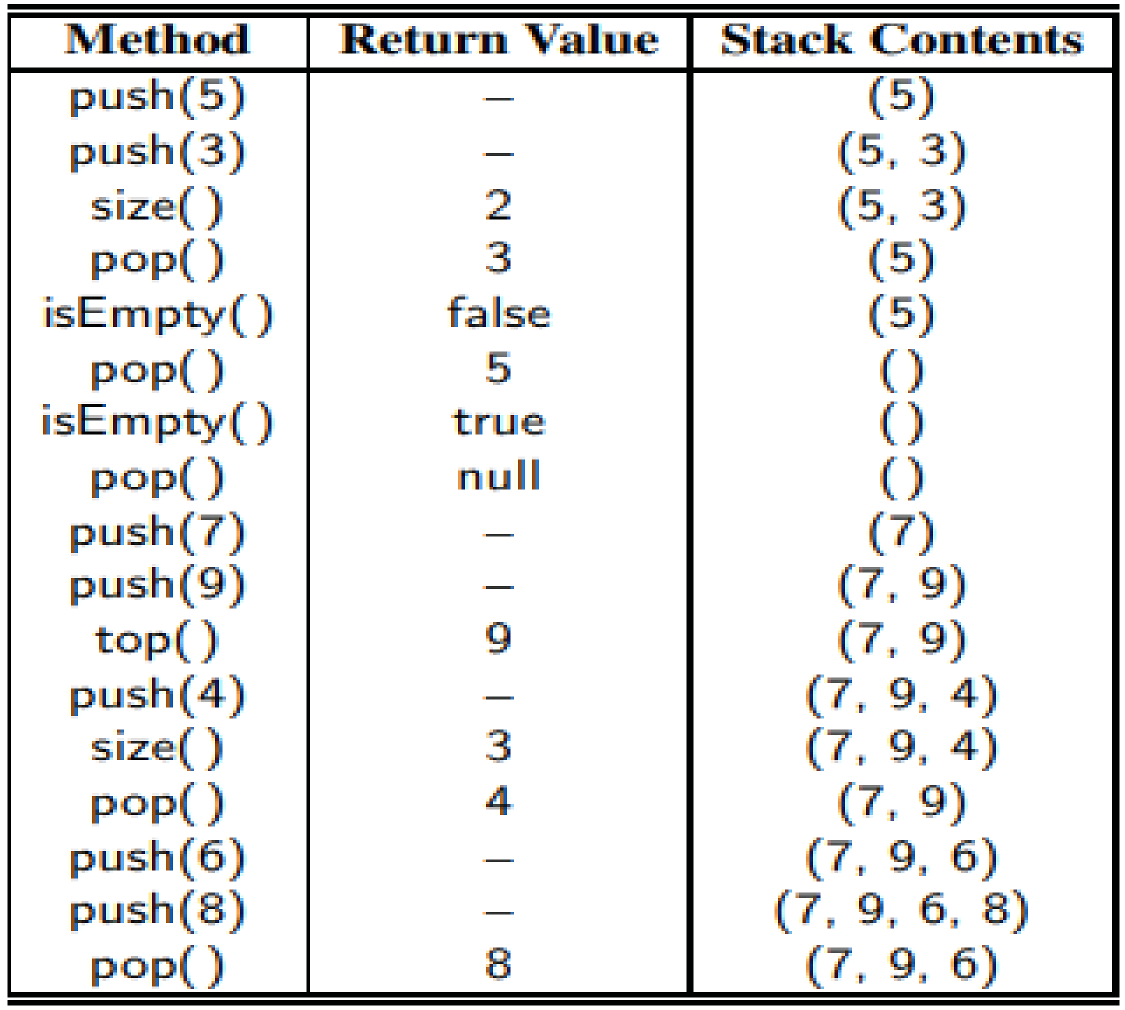


```
// Import the LinkedList class
import java.util.LinkedList;

public class LinkedListExample {

  public static void main(String[] args) {

    //Create Linked List object called itiCourses
    LinkedList<String> itiCourses = new LinkedList<String>();

       //add nodes into the Linked List
       itiCourses.add("DIT");
       itiCourses.add("DHRM");
       itiCourses.add("DACC");
       itiCourses.add("DICT");

       //print to the console the Linked List
       System.out.println(itiCourses);
    }
}
```

Figure 2: Implementation of Linked Link in JGrasp

Example of Stack operations and effect on a stack

- The following table shows a series of stack operations and their effects on an initially empty stack S of integers.



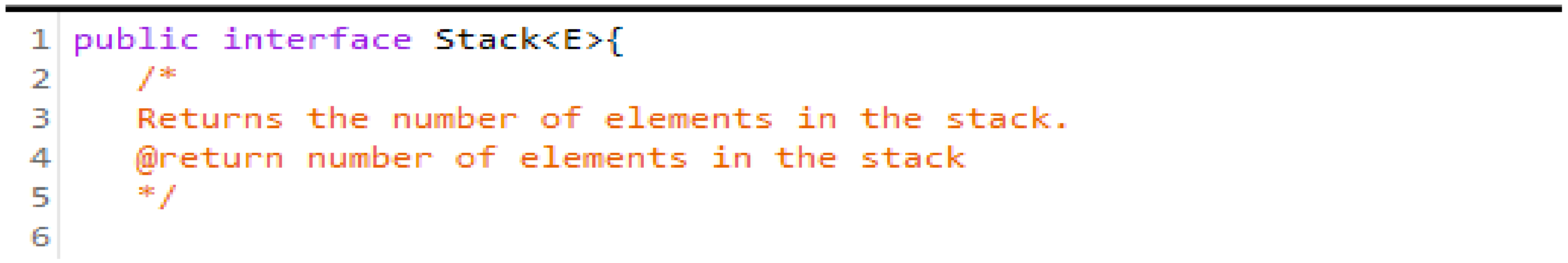
Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

A stack interface in Java

An Application Programming Interface (API) in the form of a Java Interface defines the abstraction of a stack. The API also describes the names of the methods that the ADT supports and how they are to declared and used.

To define as interface the use of Java generics must be used to allow elements to be stored in the stack and belong to any datatype or object type.

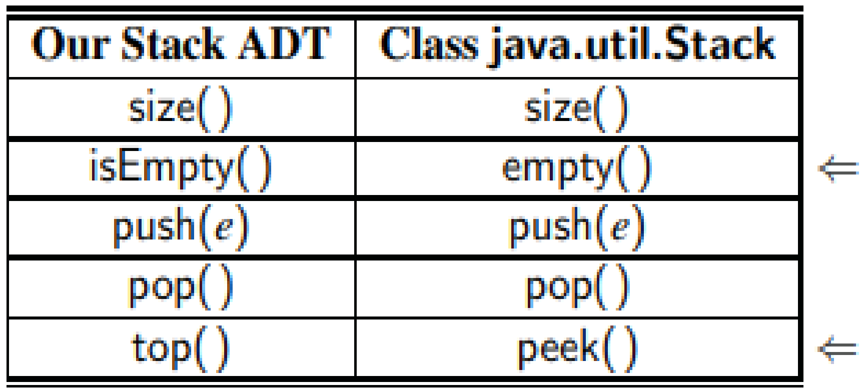
For example, a variable representing a stack of integers could be declared with type Stack. The formal type parameter is used as the parameter type for the push method, and the return type for both pop and top.



```java
public interface Stack<E>{
   /*
   Returns the number of elements in the stack.
   @return number of elements in the stack
   */

```

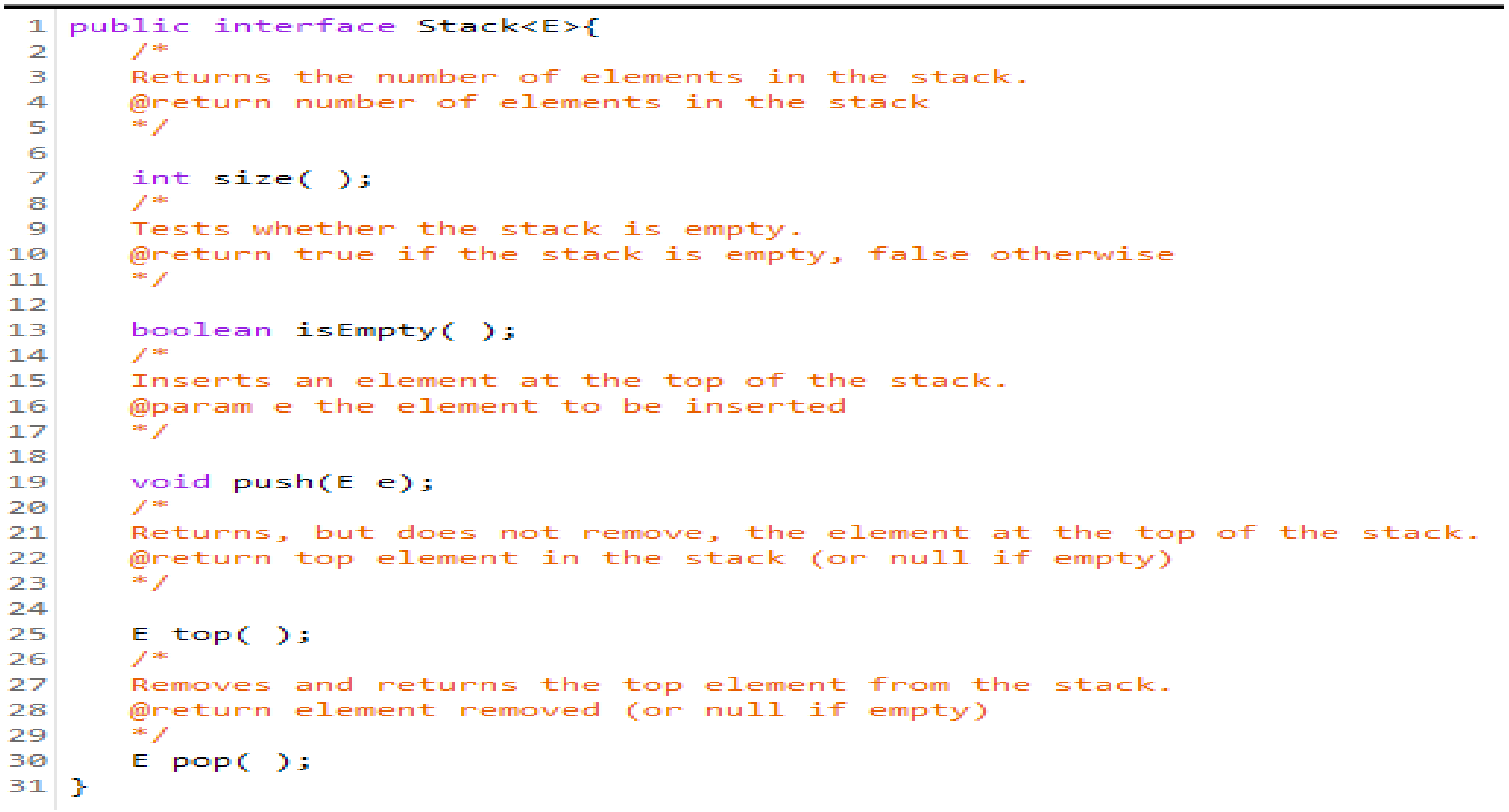
java.util.Stack class

- Like Arrays and LinkedList, Java provides a class named java.util.Stack that implements the LIFO semantics of a stack.
- NOTE: before creating your program’s class, you must import java.util.Stack; class in order to use and manipulate Stacks in Java.



Our Stack ADT	Class java.util.Stack	
size()	size()	
isEmpty()	empty()	⇐
push(e)	push(e)	
pop()	pop()	
top()	peek()	⇐

Table 6.1: Methods of our stack ADT and corresponding methods of the class java.util.Stack, with differences highlighted in the right margin.

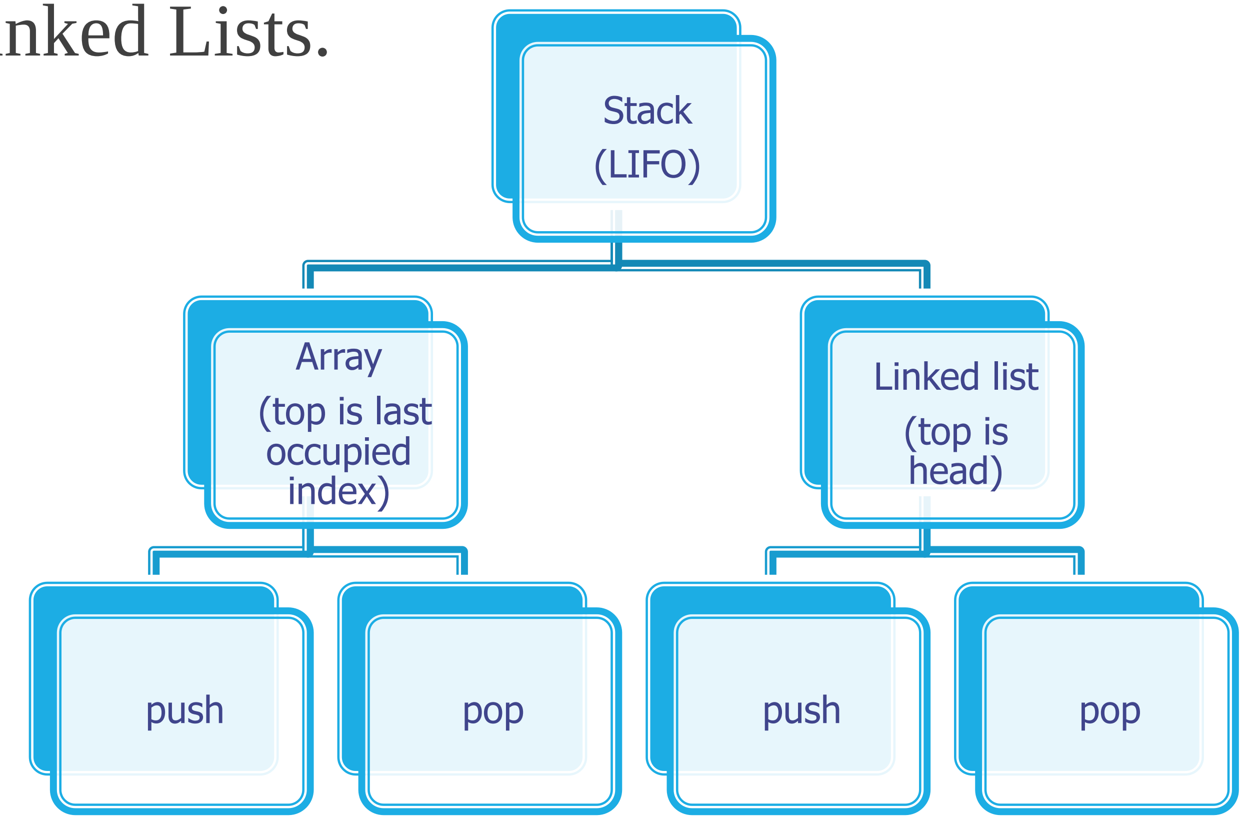


```java
public interface Stack<E>{
    /*
    Returns the number of elements in the stack.
    @return number of elements in the stack
    */

    int size( );
    /*
    Tests whether the stack is empty.
    @return true if the stack is empty, false otherwise
    */

    boolean isEmpty( );
    /*
    Inserts an element at the top of the stack.
    @param e the element to be inserted
    */

    void push(E e);
    /*
    Returns, but does not remove, the element at the top of the stack.
    @return top element in the stack (or null if empty)
    */

    E top( );
    /*
    Removes and returns the top element from the stack.
    @return element removed (or null if empty)
    */
    E pop( );
}
```

Figure 1: Example of an interface stack with comments. Note the methods and structure to implement a stack

Stacks (LIFO) implemented through Array and Linked List

The use of stacks [Last-in-first-out (LIFO)] are implemented through Arrays and Linked Lists.



A simple Array-Based Stack Implementation

- **A simple way of implementing the Stack ADT is with Arrays.**
- **Add elements from left to right**
- As our first implementation of the stack ADT, we store elements in an array, named data, with capacity N for some fixed N. We oriented the stack so that the bottom element of the stack is always stored in cell data[0] or index[0], and the top element of the stack in cell data[t] for index t that is equal to one less than the current size of the stack. (See Figure 6.2.)

Recalling that arrays start at index 0 in Java, when the stack holds elements from data[0] to data[t] inclusive, it has size t +1.

When the stack is empty it will have t equal to −1 (and thus has size t + 1, which is 0).

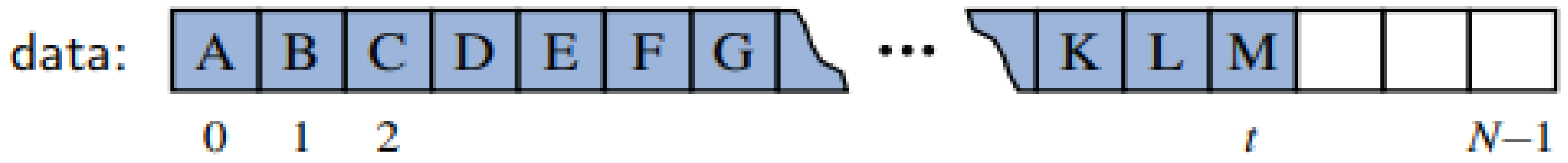
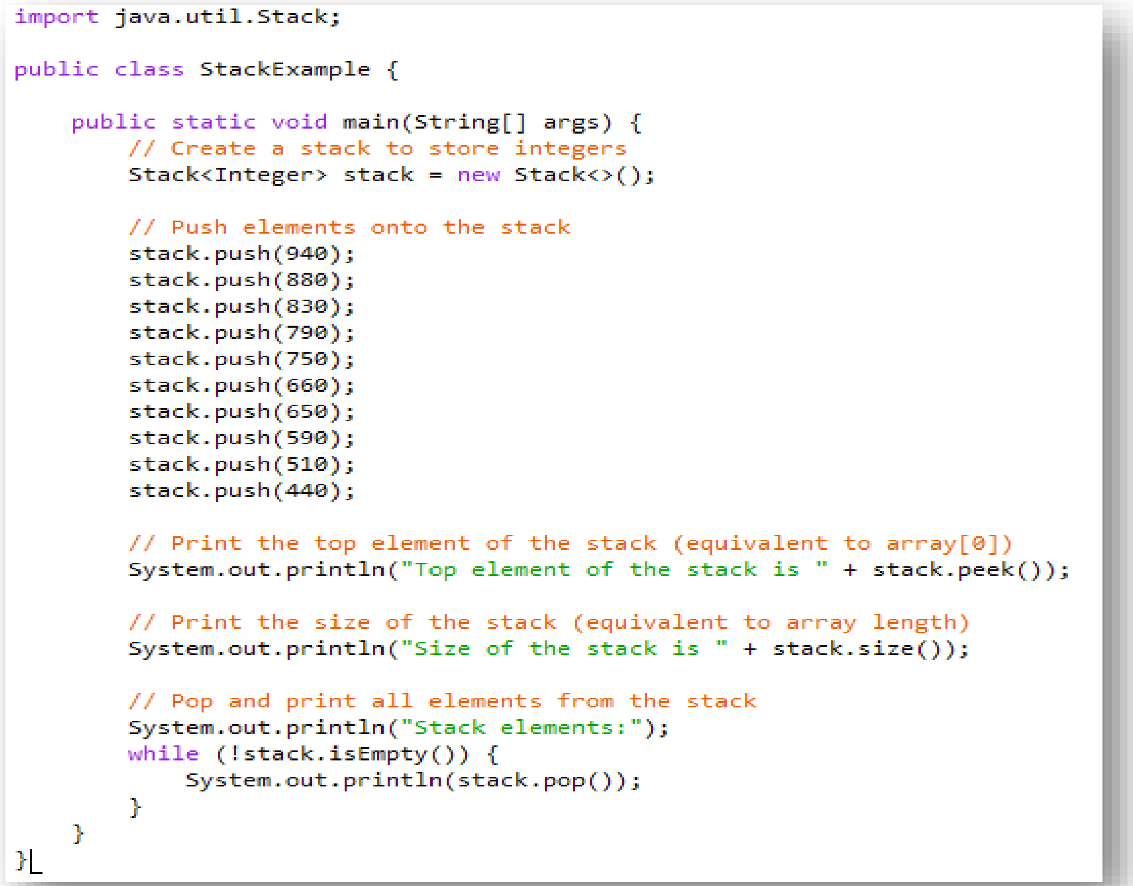
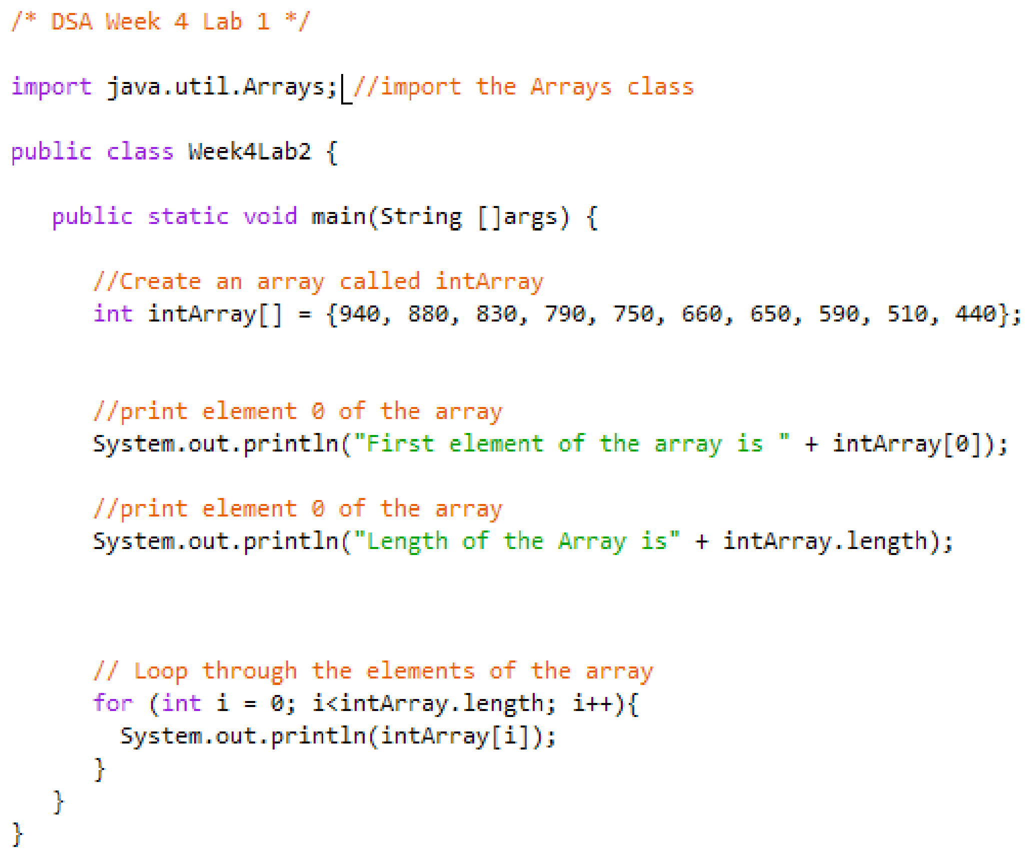


Figure 6.2: Representing a stack with an array; the top element is in cell data[t].

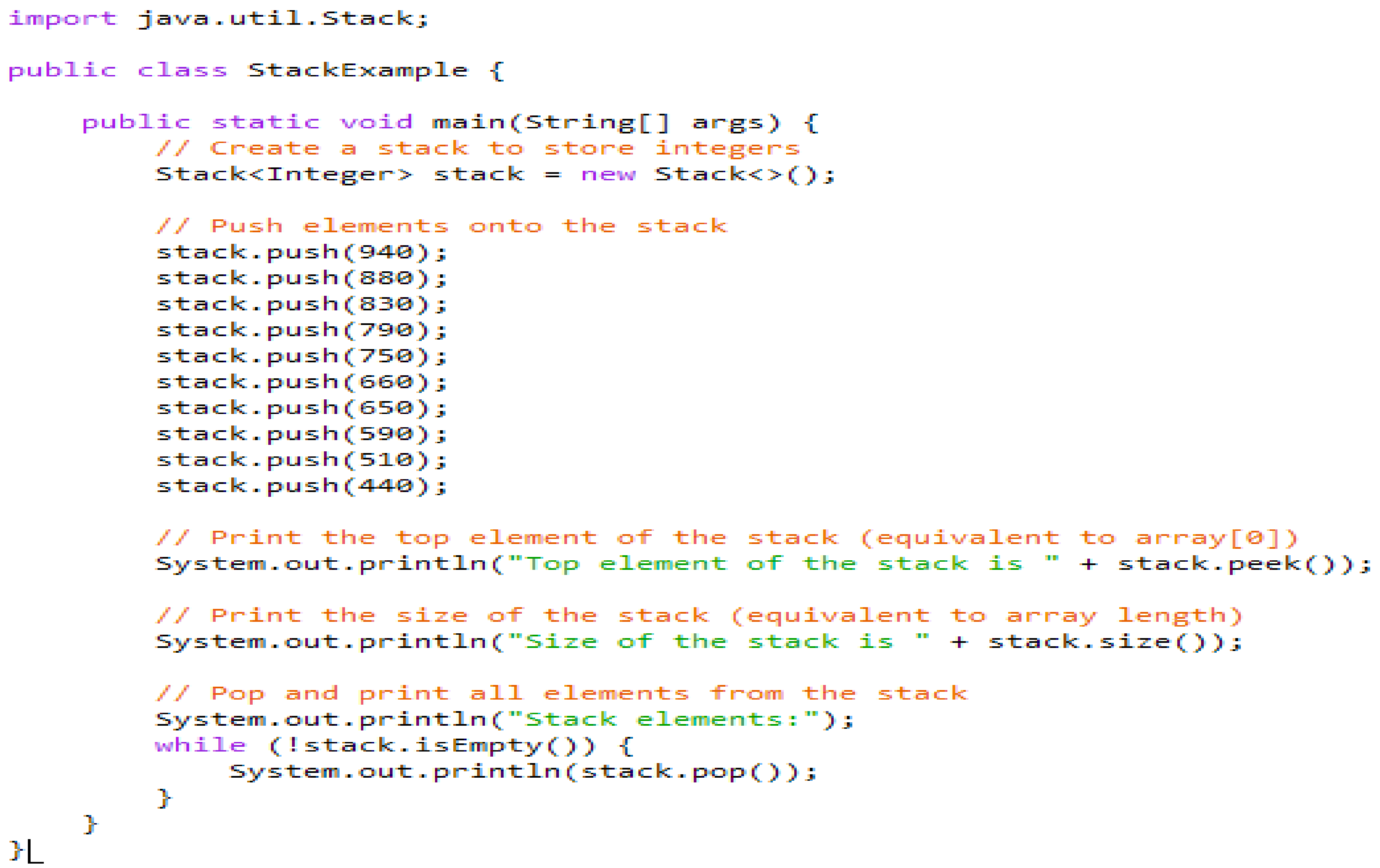
Stack & Arrays comparison



```java
import java.util.Stack;

public class StackExample {

    public static void main(String[] args) {
        // Create a stack to store integers
        Stack<Integer> stack = new Stack<>();

        // Push elements onto the stack
        stack.push(940);
        stack.push(880);
        stack.push(830);
        stack.push(790);
        stack.push(750);
        stack.push(660);
        stack.push(650);
        stack.push(590);
        stack.push(510);
        stack.push(440);

        // Print the top element of the stack (equivalent to array[0])
        System.out.println("Top element of the stack is " + stack.peek());

        // Print the size of the stack (equivalent to array length)
        System.out.println("Size of the stack is " + stack.size());

        // Pop and print all elements from the stack
        System.out.println("Stack elements:");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```



```java
/* DSA Week 4 Lab 1 */

import java.util.Arrays;//import the Arrays class

public class Week4Lab2 {

   public static void main(String []args) {

      //Create an array called intArray
      int intArray[] = {940, 880, 830, 790, 750, 660, 650, 590, 510, 440};


      //print element 0 of the array
      System.out.println("First element of the array is " + intArray[0]);

      //print element 0 of the array
      System.out.println("Length of the Array is" + intArray.length);



      // Loop through the elements of the array
      for (int i = 0; i<intArray.length; i++){
        System.out.println(intArray[i]);
      }
   }
}
```



```java
import java.util.Stack;

public class StackExample {

    public static void main(String[] args) {
        // Create a stack to store integers
        Stack<Integer> stack = new Stack<>();

        // Push elements onto the stack
        stack.push(940);
        stack.push(880);
        stack.push(830);
        stack.push(790);
        stack.push(750);
        stack.push(660);
        stack.push(650);
        stack.push(590);
        stack.push(510);
        stack.push(440);

        // Print the top element of the stack (equivalent to array[0])
        System.out.println("Top element of the stack is " + stack.peek());

        // Print the size of the stack (equivalent to array length)
        System.out.println("Size of the stack is " + stack.size());

        // Pop and print all elements from the stack
        System.out.println("Stack elements:");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

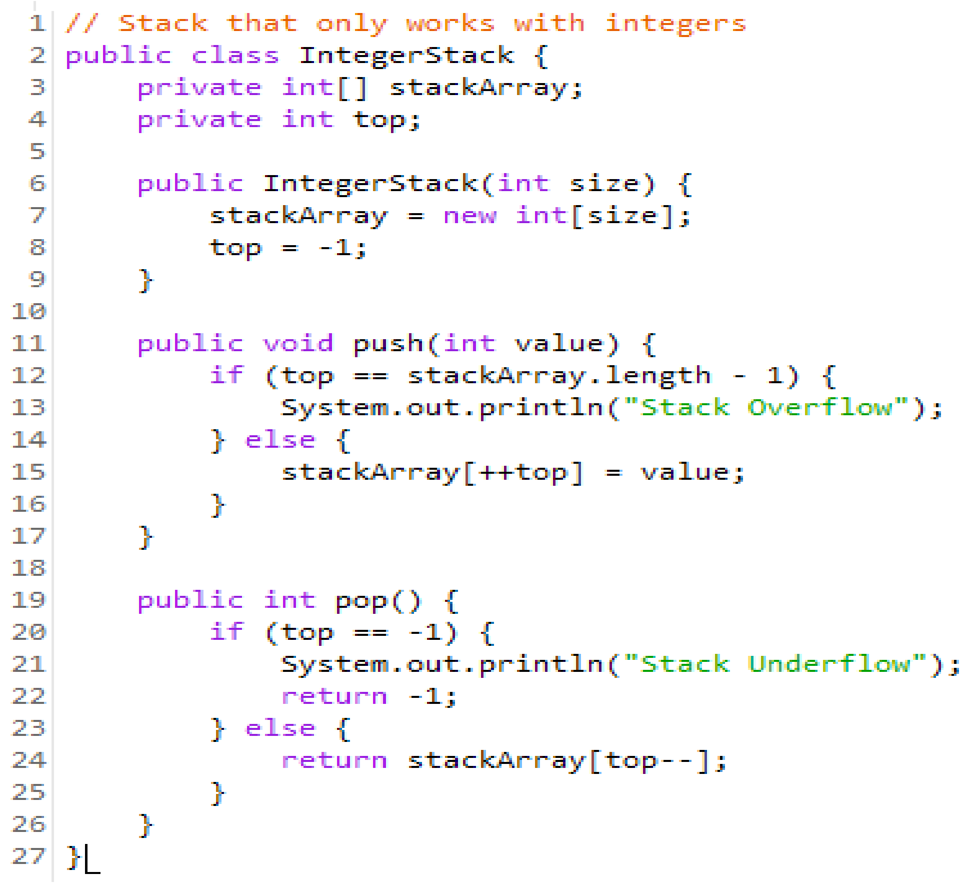
A drawback of this Array-Based stack implementation

- The array implementation of a stack is simple and efficient.
- However, the implementation has a disadvantage the array’s fixed-capacity which limited the ultimate size of the stack.
- A solution will be to allow the user of a stack to specify the capacity as a parameter to the constructor (and offer a default constructor that uses capacity of 1,000).
- However, if this estimation is wrong the application will waste memory space. Further, if an attempt is made to push an item onto a stack that has already reached its maximum capacity, the code will throw an IllegalStateException, refusing to store the new element. Thus, even with its simplicity and efficiency, the array-based stack implementation is not necessarily ideal.

Learning Java Generics through stacks

- You have learnt different data types and how to create data structures such as Arrays and LinkedLists.
- Using a non-generic code provides programmer with problem of rewriting code for different data types of use with each data structure.

Problem: the IntegerStack program or class can hold only integer values. What if we want to create a stack of String values or Double values? Is it efficient to rewrite the different class or new program for each data type?



```java
// Stack that only works with integers
public class IntegerStack {
    private int[] stackArray;
    private int top;

    public IntegerStack(int size) {
        stackArray = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == stackArray.length - 1) {
            System.out.println("Stack Overflow");
        } else {
            stackArray[++top] = value;
        }
    }

    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return -1;
        } else {
            return stackArray[top--];
        }
    }
}
```

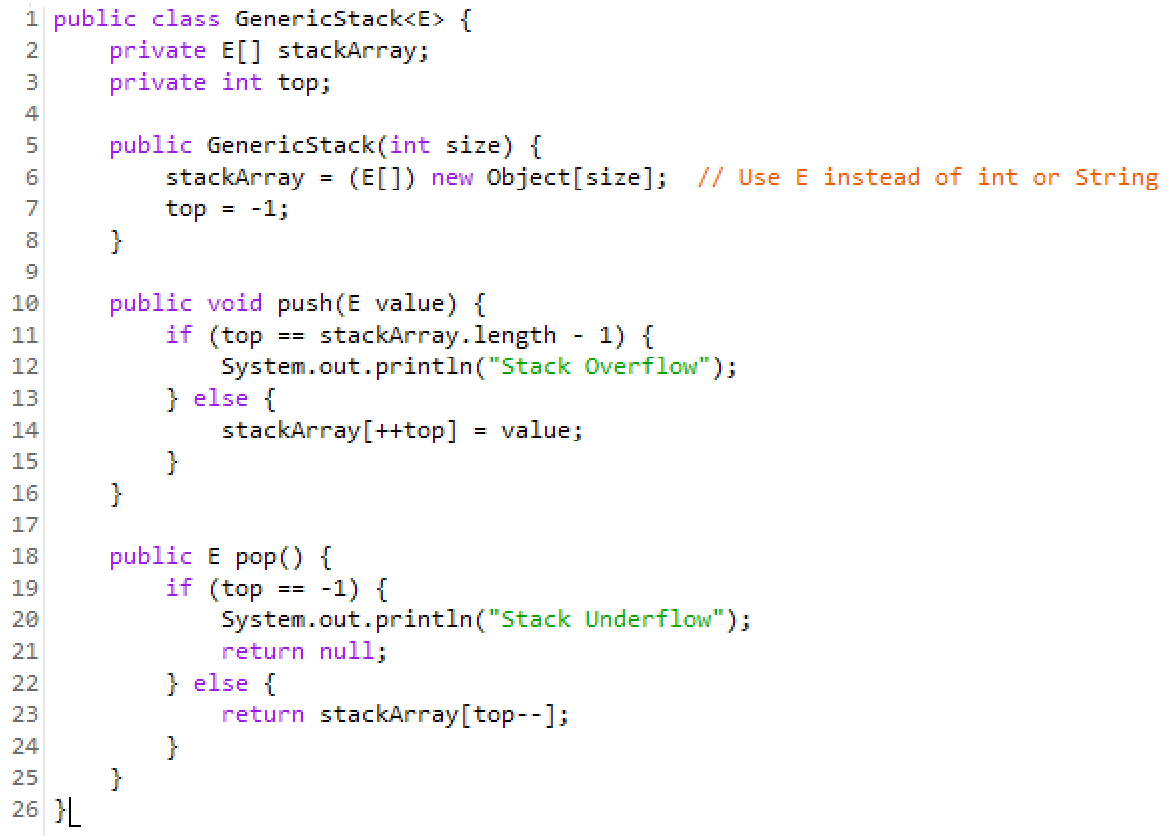
Figure 2: Stack code without Generics (non-generics)

Solution is to implement Generics

- In programming, it is important to write reusable code, hence, generics is a solution.
- Programmers must avoid writing different stack classes for every datatype (StringStack, DoubleStack, IntegerStack, etc) this would be repetitive and inefficient.
- Therefore, generics provides a way to create reusable and flexible code that can work with any datatype.

Generic syntax explanation

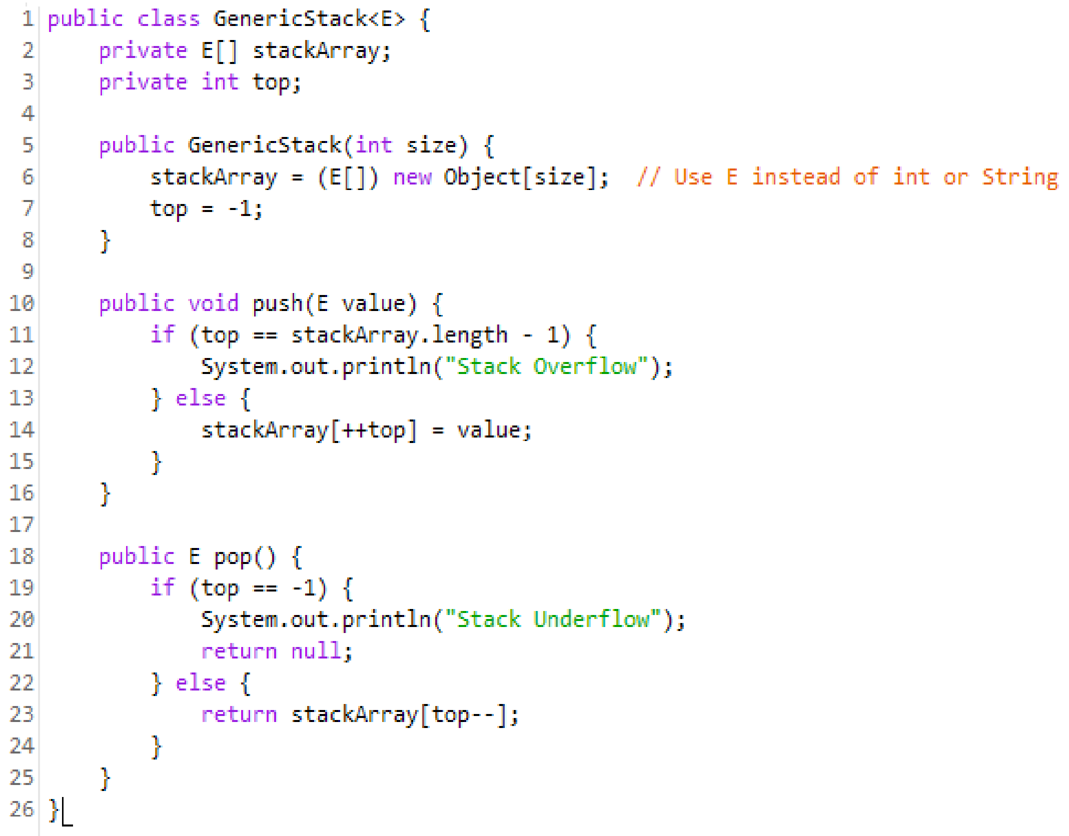
- The angle brackets (<E>) are like a placeholder for any data type (you can use other letters like T, but E is commonly used for "Element").
- E is not a fixed type like int or String; it will be replaced with a specific type when we create an object of the stack (like Integer or String).



```java
public class GenericStack<E> {
    private E[] stackArray;
    private int top;

    public GenericStack(int size) {
        stackArray = (E[]) new Object[size];  // Use E instead of int or String
        top = -1;
    }

    public void push(E value) {
        if (top == stackArray.length - 1) {
            System.out.println("Stack Overflow");
        } else {
            stackArray[++top] = value;
        }
    }

    public E pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return null;
        } else {
            return stackArray[top--];
        }
    }
}
```

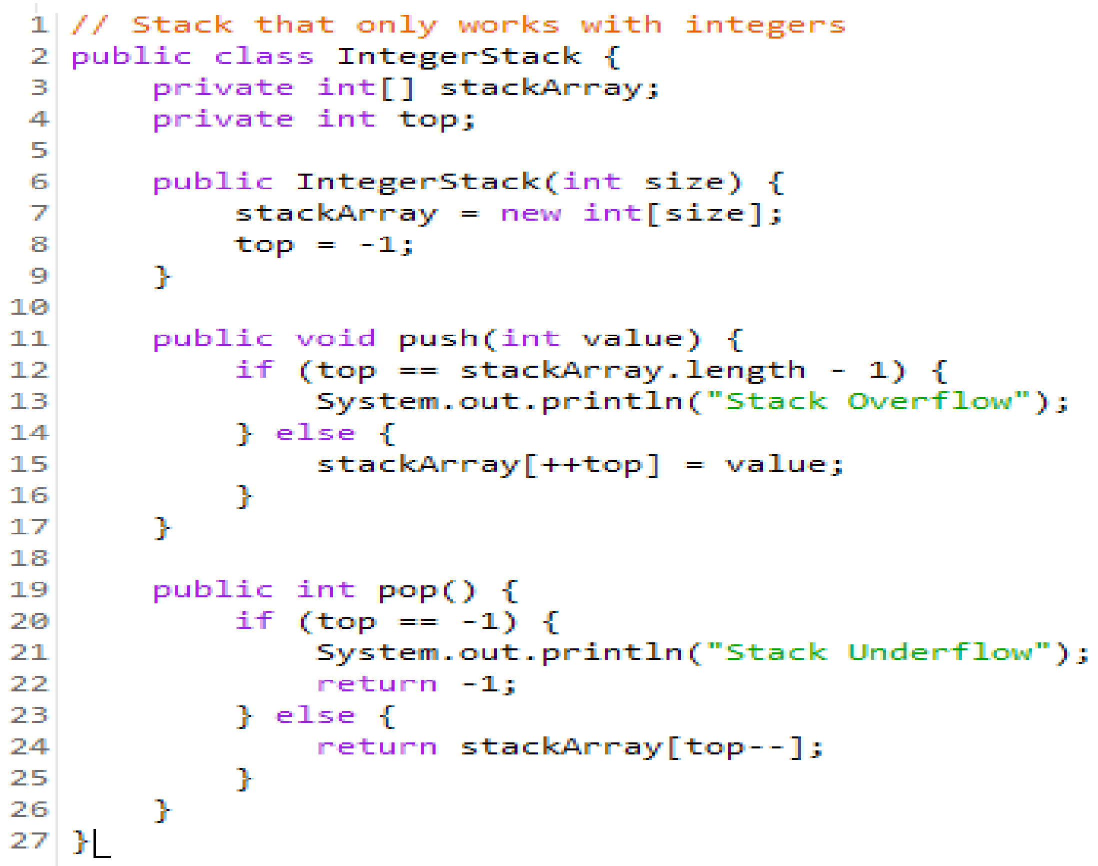
Figure 3: Improved code using Generics

Generic vs Non-Generic stack code (syntax)



```java
public class GenericStack<E> {
    private E[] stackArray;
    private int top;

    public GenericStack(int size) {
        stackArray = (E[]) new Object[size];  // Use E instead of int or String
        top = -1;
    }

    public void push(E value) {
        if (top == stackArray.length - 1) {
            System.out.println("Stack Overflow");
        } else {
            stackArray[++top] = value;
        }
    }

    public E pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return null;
        } else {
            return stackArray[top--];
        }
    }
}
```

Figure 3: Improved code using Generics

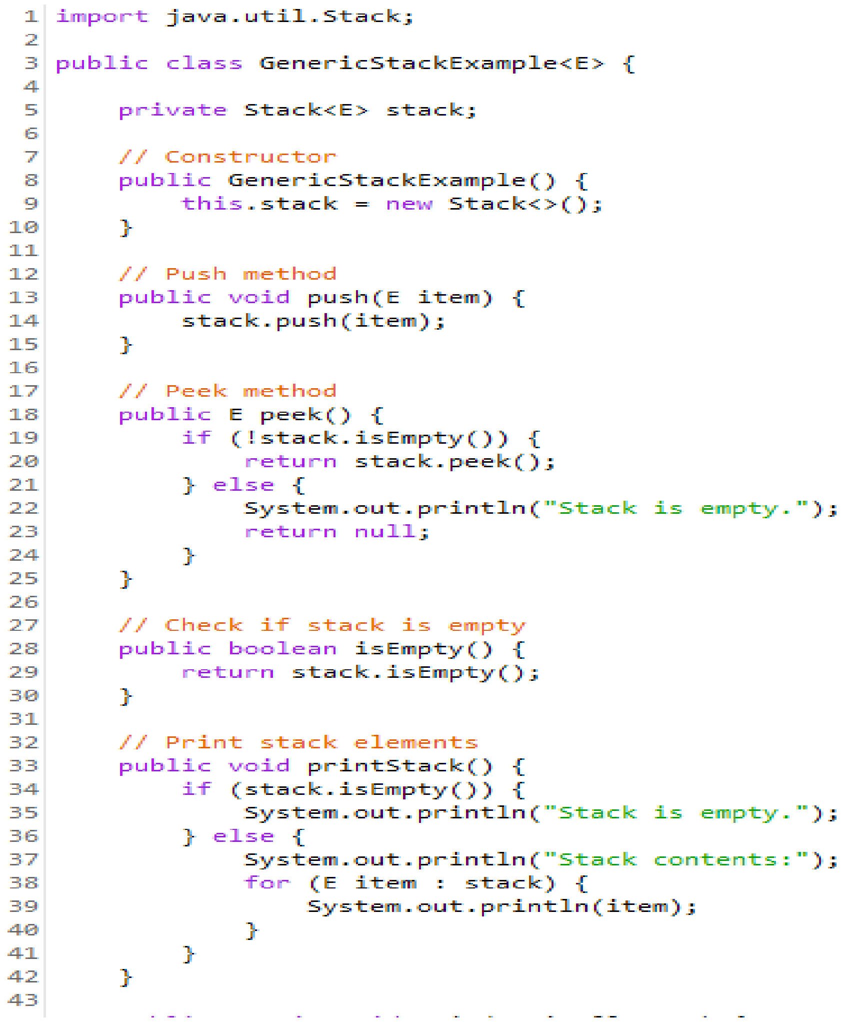


```java
// Stack that only works with integers
public class IntegerStack {
    private int[] stackArray;
    private int top;

    public IntegerStack(int size) {
        stackArray = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top == stackArray.length - 1) {
            System.out.println("Stack Overflow");
        } else {
            stackArray[++top] = value;
        }
    }

    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return -1;
        } else {
            return stackArray[top--];
        }
    }
}
```

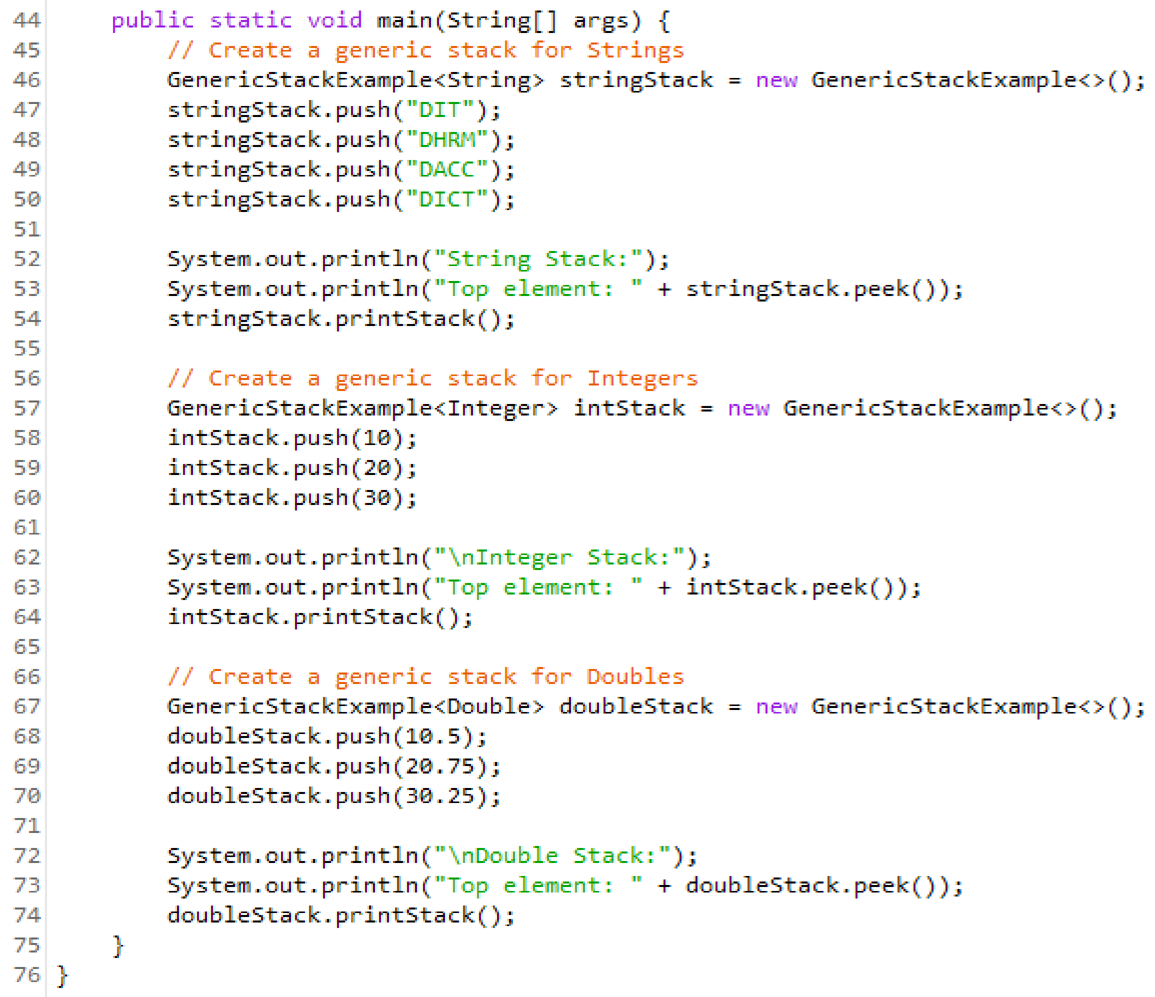
Figure 2: Stack code without Generics (non-generics)

Implementing a stack with a generic array (1/2)

The same program without generic was shown earlier in the simple Array-Based Stack Implementation section of the lecture.

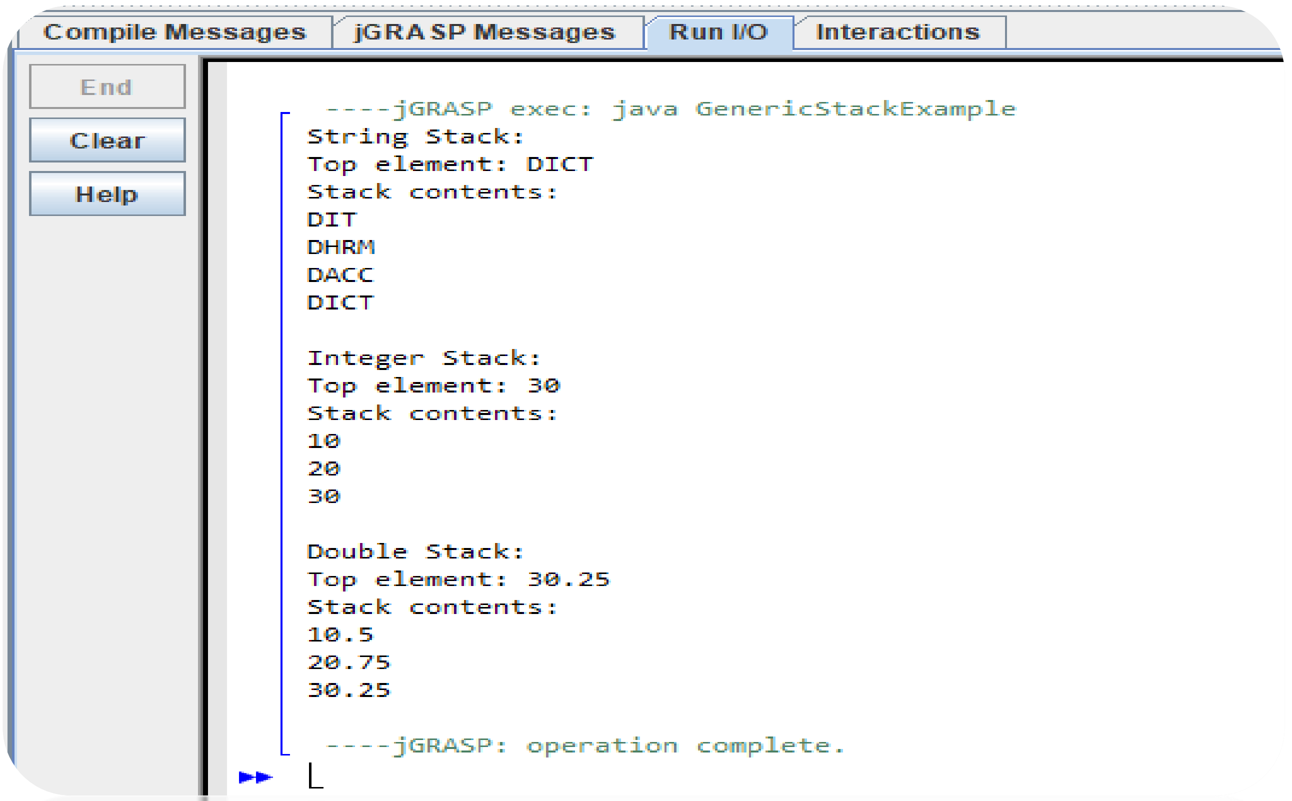


```java
import java.util.Stack;

public class GenericStackExample<E> {

    private Stack<E> stack;

    // Constructor
    public GenericStackExample() {
        this.stack = new Stack<>();
    }

    // Push method
    public void push(E item) {
        stack.push(item);
    }

    // Peek method
    public E peek() {
        if (!stack.isEmpty()) {
            return stack.peek();
        } else {
            System.out.println("Stack is empty.");
            return null;
        }
    }

    // Check if stack is empty
    public boolean isEmpty() {
        return stack.isEmpty();
    }

    // Print stack elements
    public void printStack() {
        if (stack.isEmpty()) {
            System.out.println("Stack is empty.");
        } else {
            System.out.println("Stack contents:");
            for (E item : stack) {
                System.out.println(item);
            }
        }
    }

```



```java
    public static void main(String[] args) {
        // Create a generic stack for Strings
        GenericStackExample<String> stringStack = new GenericStackExample<>();
        stringStack.push("DIT");
        stringStack.push("DHRM");
        stringStack.push("DACC");
        stringStack.push("DICT");

        System.out.println("String Stack:");
        System.out.println("Top element: " + stringStack.peek());
        stringStack.printStack();

        // Create a generic stack for Integers
        GenericStackExample<Integer> intStack = new GenericStackExample<>();
        intStack.push(10);
        intStack.push(20);
        intStack.push(30);

        System.out.println("\nInteger Stack:");
        System.out.println("Top element: " + intStack.peek());
        intStack.printStack();

        // Create a generic stack for Doubles
        GenericStackExample<Double> doubleStack = new GenericStackExample<>();
        doubleStack.push(10.5);
        doubleStack.push(20.75);
        doubleStack.push(30.25);

        System.out.println("\nDouble Stack:");
        System.out.println("Top element: " + doubleStack.peek());
        doubleStack.printStack();
    }
}
```

Implementing a stack with a generic array (2/2)



```
 ----jGRASP exec: java GenericStackExample
String Stack:
Top element: DICT
Stack contents:
DIT
DHRM
DACC
DICT

Integer Stack:
Top element: 30
Stack contents:
10
20
30

Double Stack:
Top element: 30.25
Stack contents:
10.5
20.75
30.25

 ----jGRASP: operation complete.
```

Figure 4: Program output

Topic next week

Week 9 Queues.

DSA Week 8 activities

Refer to print out materials.

- This week, you are required to complete the questions and two labs.
  - In your DSA textbook 1 page 141, answer questions 2 and 5.
  - Refer to the print out, answer all week 8 questions.
  - Refer to the print out, complete the two labs activities using the lab computers.

Note: You can complete the activities in any order, however, make afford to complete and understand everything which prepares you for well for test 2 & Final Exam.