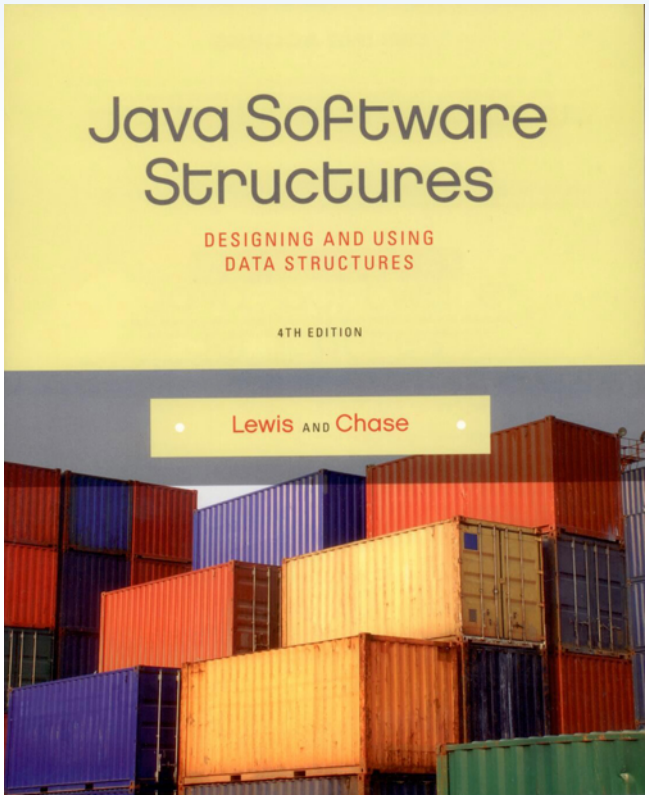


Chapter 3

Introduction to Collections - Stacks

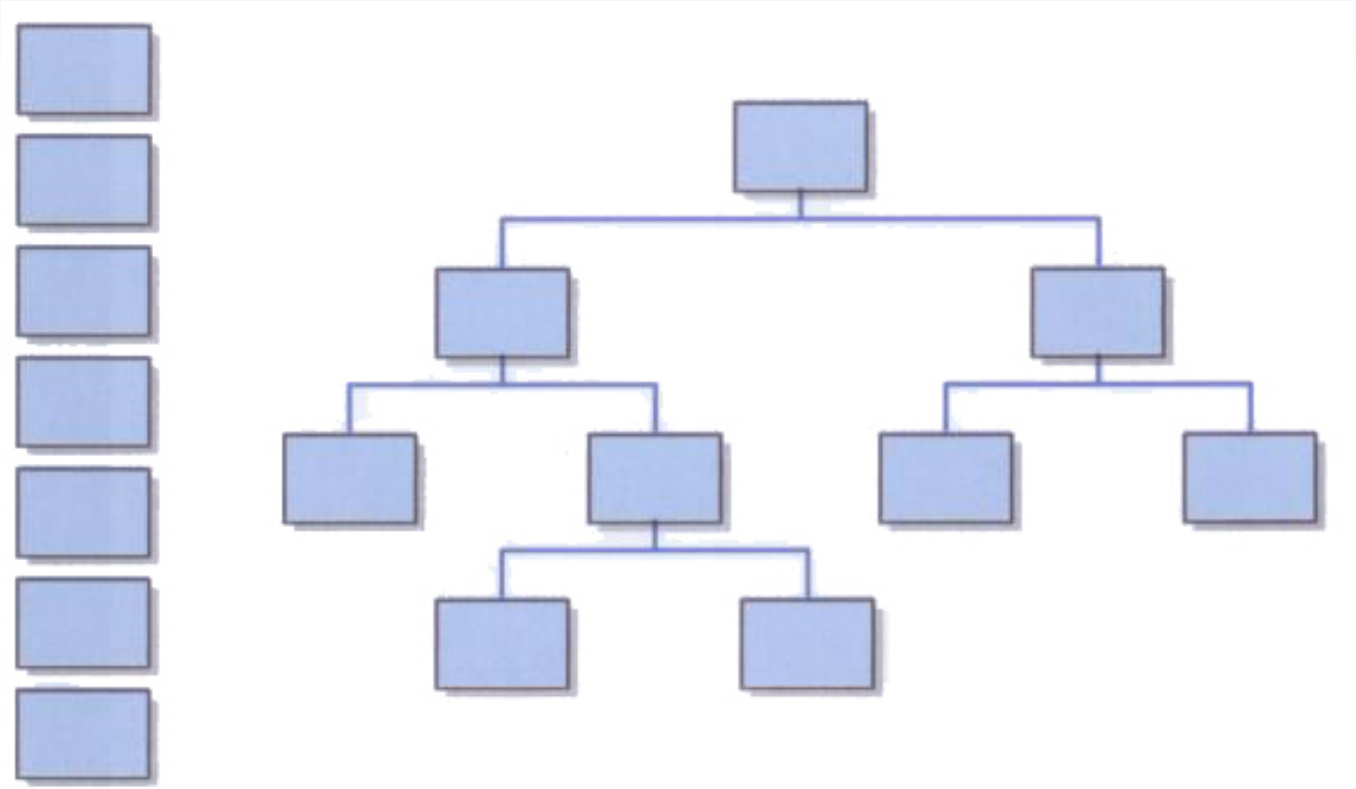
Chapter Scope

- Collection terminology
- The Java Collections API
- Abstract nature of collections
- Stacks
  - Conceptually
  - Used to solve problems
  - Implementation using arrays

Collections

- A *collection* is an object that holds and organizes other objects
- It provides operations for accessing and managing its *elements*
- Many standard collections have been defined over time
- Some collections are *linear* in nature, others are *non-linear*

Linear and Non-Linear Collections



Abstraction

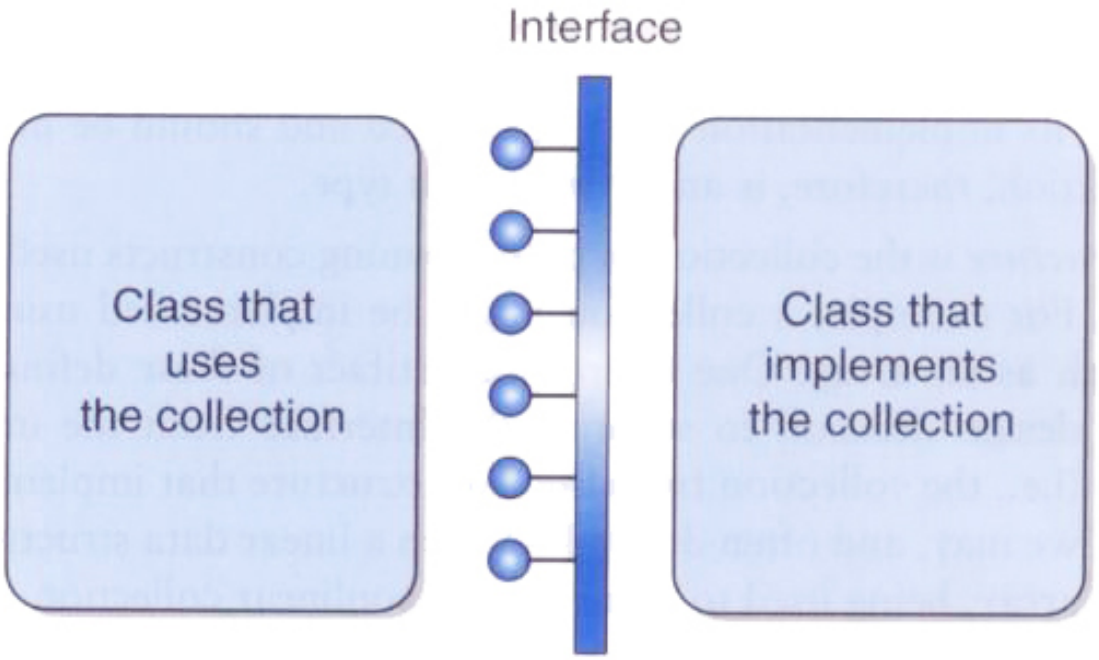
- An *abstraction* hides details to make a concept easier to manage
- All objects are abstractions in that they provide well-defined operations (the *interface*)
- They hide (*encapsulate*) the object's data and the implementation of the operations
- An object is a great mechanism for implementing a collection

Abstract Data Type

- A *data type* is a group of values and the operations defined on those values
- An *abstract data type* (ADT) is a data type that isn't pre-defined in the programming language
- A *data structure* is the set of programming constructs and techniques used to implement a collection
- The distinction between the terms ADT, data structure, and collection is sometimes blurred in casual use

Collection Abstraction

- A class that uses a collection interacts with it through a particular interface



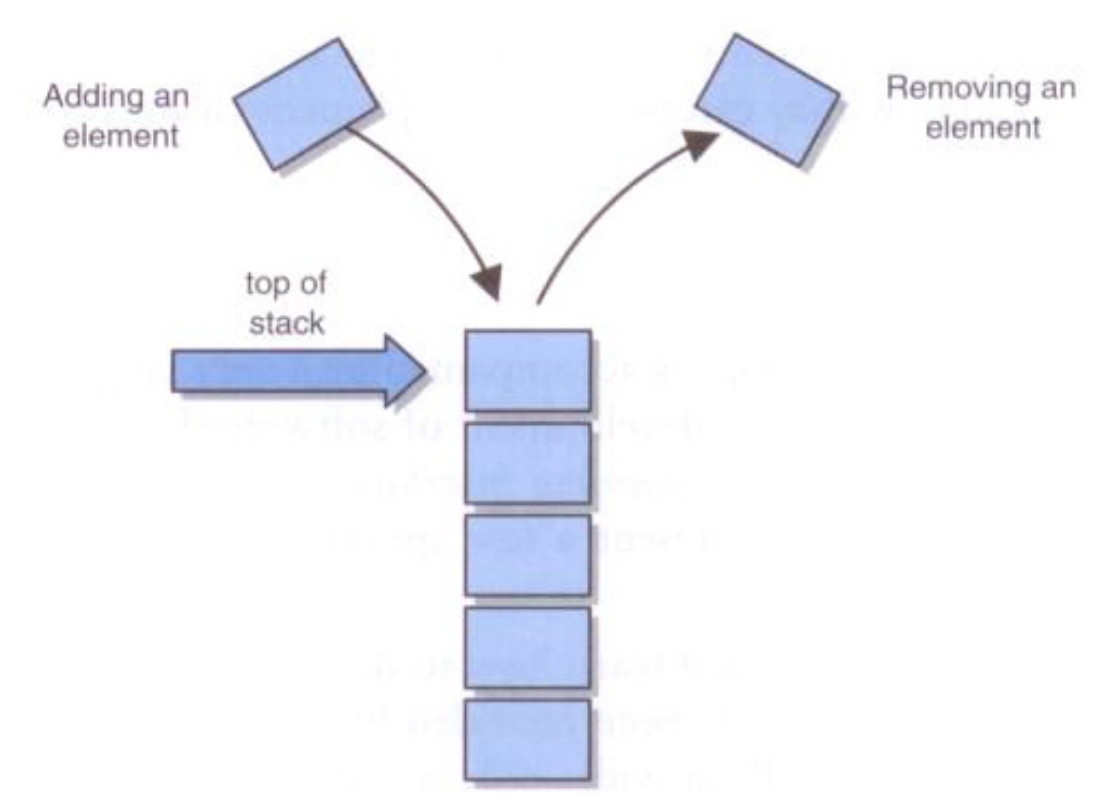
The Java Collections API

- As you know, the various classes provided with Java are referred to as the Java API (application programmer interface)
- The subset of those classes that support collections is called the *Java Collections API*
- As we explore a particular collection, we will also examine any support provided in the Collections API for that collection

Stacks

- A *stack* is a classic collection used to help solve many types of problems
- A stack is a linear collection whose elements are added in a *last in, first out* (LIFO) manner
- That is, the last element to be put on a stack is the first one to be removed
- Think of a stack of books, where you add and remove from the top, but can't reach into the middle

Stack - Conceptual View



Stack Operations

- Some collections use particular terms for their operations
- These are the classic stack operations:

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 if the stack is empty.
`size`	Determines the number of elements on the stack.

Object-Oriented Concepts

- Several OO concepts, new and old, will be brought to bear as we explore collections
- We'll rely on type compatibility rules, interfaces, inheritance, and polymorphism
- Review these concepts as necessary
- We'll also rely on a programming mechanism introduced in Java 5, *generics*, which are particularly appropriate for implementing collections

Generics

- Suppose we define a stack that holds `Object` references, which would allow it to hold any object
- But then we lose control over which types of elements are added to the stack, and we'd have to cast elements to their proper type when removed
- A better solution is to define a class that is based on a *generic type*
- The type is referred to generically in the class, and the specific type is specified only when an object of that class is created

Generics

- The generic type placeholder is specified in angle brackets in the class header:

```
class Box<T>
{
    // declarations and code that refer to T
}
```

- Any identifier can be used, but T (for Type) or E (for element) have become standard practice

Generics

- Then, when a `Box` is needed, it is instantiated with a specific class instead of T:

```
Box<Widget> box1 = new Box<Widget>();
```

- Now, `box1` can only hold `Widget` objects
- The compiler will issue errors if we try to add a non-`Widget` to the box
- And when an object is removed from the box, it is assumed to be a `Widget`

Generics

- Using the same class, another object can be instantiated:

```
Box<Gadget> box2 = new Box<Gadget>();
```

- The `box2` object can only hold `Gadget` objects
- Generics provide better type management control at compile-time and simplify the use of collection classes

Postfix Expressions

- Let's see how a stack can be used to help us solve the problem of evaluating postfix expressions
- A *postfix expression* is written with the operator following the two operands instead of between them:

15 8 -

is equivalent to the traditional (infix) expression

15 - 8

Postfix Expressions

- Postfix expressions eliminate the need for parentheses to specify the order of operations
- The infix expression

$$(3 * 4 - (2 + 5)) * 4 / 2$$

is equivalent to the postfix expression

$$3\ 4\ *\ 2\ 5\ +\ -\ 4\ *\ 2\ /$$

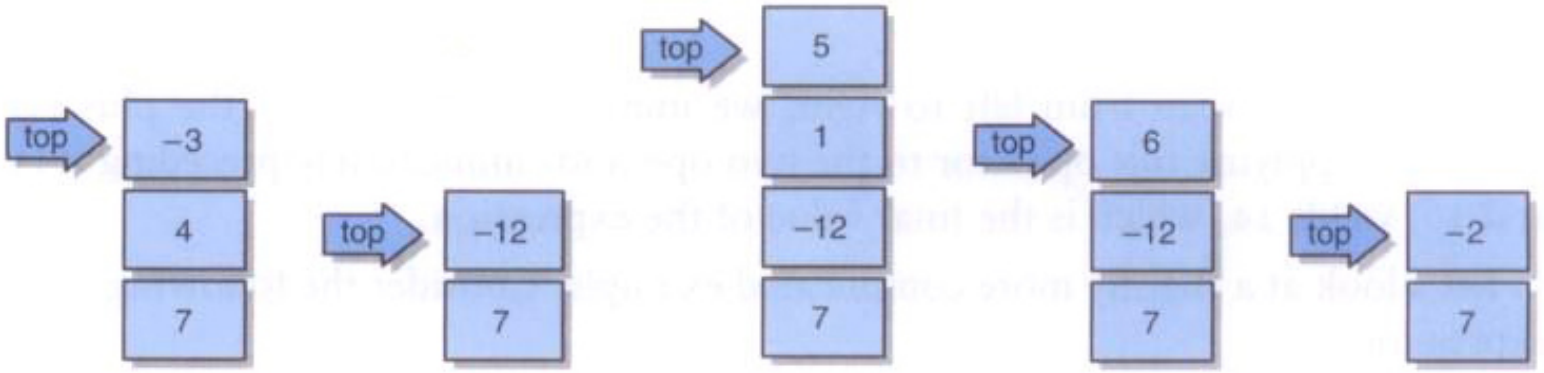
Evaluating Postfix Expressions

- We'll use a stack as follows to evaluate a postfix expression:
  - Scan the expression left to right
  - When an operand is encountered, push it on the stack
  - When an operator is encountered, pop the top two elements on the stack, perform the operation, then push the result on the stack
- When the expression is exhausted, the value on the stack is the final result

Evaluating Postfix Expressions

- Here's how the stack changes as the following expression is evaluated:

7 4 -3 * 1 5 + / *



```
import java.util.Scanner;

/**
 * Demonstrates the use of a stack to evaluate postfix expressions.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixTester
{
    /**
     * Reads and evaluates multiple postfix expressions.
     */
    public static void main(String[] args)
    {
        String expression, again;
        int result;

        Scanner in = new Scanner(System.in);

        do
        {
            PostfixEvaluator evaluator = new PostfixEvaluator();
            System.out.println("Enter a valid post-fix expression one token " +
                               "at a time with a space between each token (e.g. 5 4 + 3 2 1 - + *)");
            System.out.println("Each token must be an integer or an operator (+,-,*,/)");
            expression = in.nextLine();

            result = evaluator.evaluate(expression);
            System.out.println();
            System.out.println("That expression equals " + result);

            System.out.print("Evaluate another expression [Y/N]? ");
            again = in.nextLine();
            System.out.println();
        }
        while (again.equalsIgnoreCase("y"));
    }
}
```

```
import java.util.Stack;
import java.util.Scanner;

/**
 * Represents an integer evaluator of postfix expressions. Assumes
 * the operands are constants.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixEvaluator
{
    private final static char ADD = '+';
    private final static char SUBTRACT = '-';
    private final static char MULTIPLY = '*';
    private final static char DIVIDE = '/';

    private Stack<Integer> stack;

    /**
     * Sets up this evalutor by creating a new stack.
     */
    public PostfixEvaluator()
    {
        stack = new Stack<Integer>();
    }
```

```
/**
 * Evaluates the specified postfix expression. If an operand is
 * encountered, it is pushed onto the stack. If an operator is
 * encountered, two operands are popped, the operation is
 * evaluated, and the result is pushed onto the stack.
 * @param expr string representation of a postfix expression
 * @return value of the given expression
 */
public int evaluate(String expr)
{
    int op1, op2, result = 0;
    String token;
    Scanner parser = new Scanner(expr);

    while (parser.hasNext())
    {
        token = parser.next();

        if (isOperator(token))
        {
            op2 = (stack.pop()).intValue();
            op1 = (stack.pop()).intValue();
            result = evaluateSingleOperator(token.charAt(0), op1, op2);
            stack.push(new Integer(result));
        }
        else
            stack.push(new Integer(Integer.parseInt(token)));
    }

    return result;
}
```

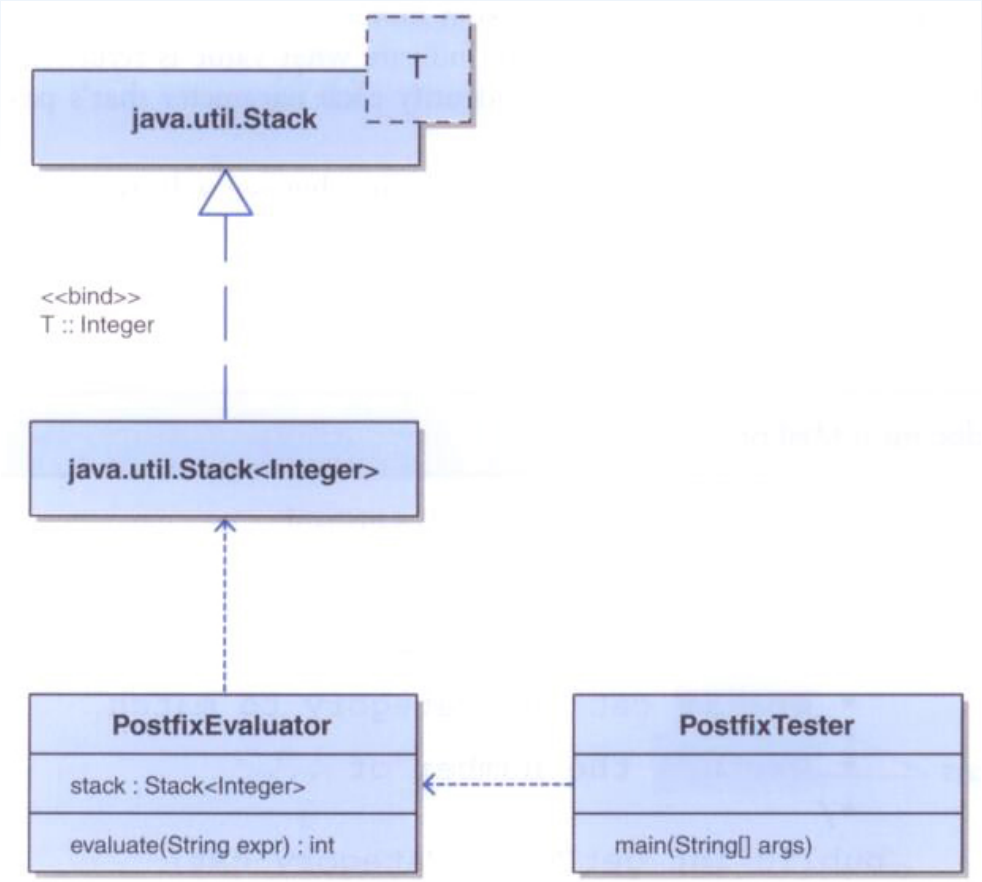
```
/**
 * Determines if the specified token is an operator.
 * @param token the token to be evaluated
 * @return true if token is operator
 */
private boolean isOperator(String token)
{
    return ( token.equals("+") || token.equals("-") ||
             token.equals("*") || token.equals("/") );
}
```

```
    /**
     * Peforms integer evaluation on a single expression consisting of
     * the specified operator and operands.
     * @param operation operation to be performed
     * @param op1 the first operand
     * @param op2 the second operand
     * @return value of the expression
     */
    private int evaluateSingleOperator(char operation, int op1, int op2)
    {
        int result = 0;

        switch (operation)
        {
            case ADD:
                result = op1 + op2;
                break;
            case SUBTRACT:
                result = op1 - op2;
                break;
            case MULTIPLY:
                result = op1 * op2;
                break;
            case DIVIDE:
                result = op1 / op2;
        }

        return result;
    }
}
```

Stacks in the Java API

- The postfix example uses the java.util.Stack class, which has been part of the Java collections API since Java 1.0.
- It implements the classic operations, without any separate interfaces defined
- As we'll see, there is not a lot of consistency regarding how collections are implemented in the Java API, and we'll explore their relative benefits as appropriate

Postfix Evaluator UML



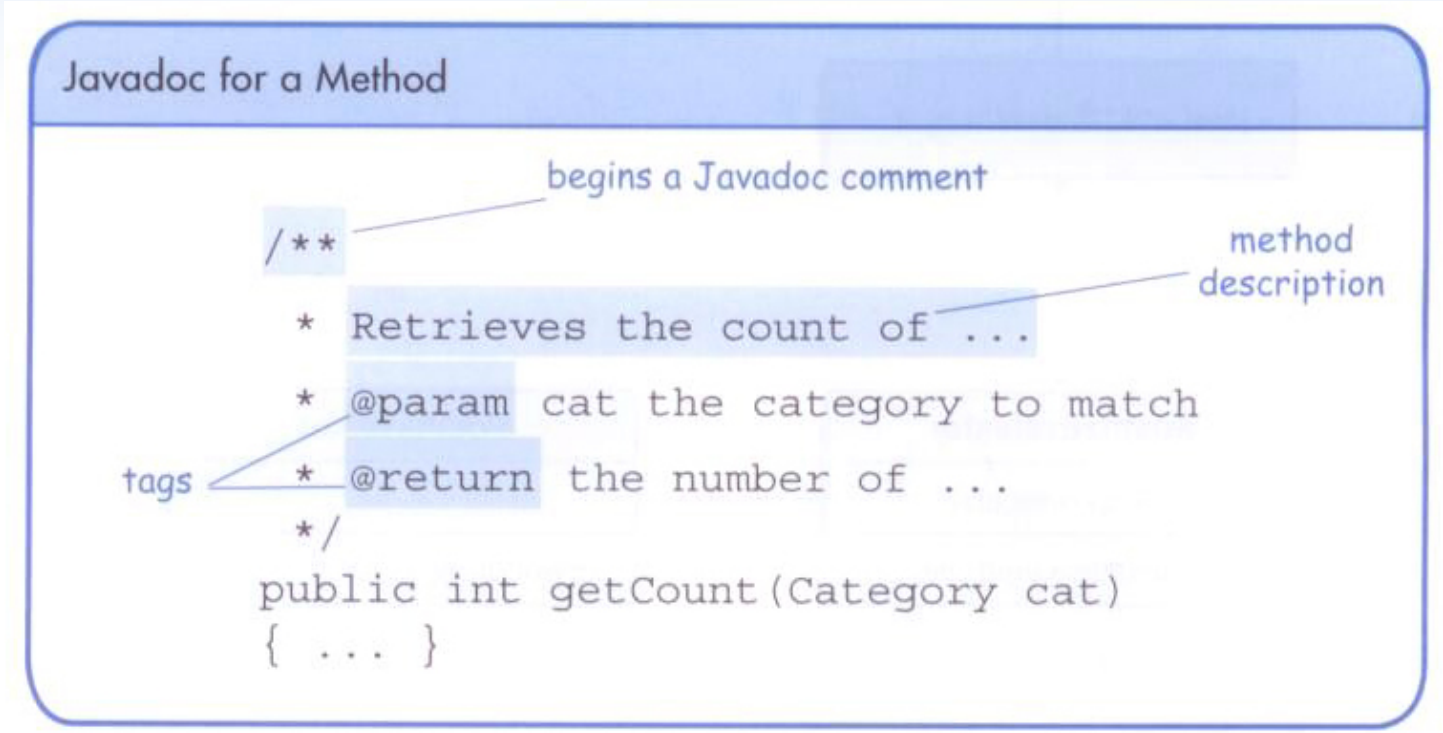
Javadoc

- The documentation style used in the postfix example is called *Javadoc*
- Javadoc comments are written in a format that allow a tool to parse the comments and extract key information
- A Javadoc tool is used to create online documentation in HTML about a set of classes
- The Java API documentation is generated in this way

Javadoc

- Javadoc comments begin with /** and end with */
- Specific Javadoc tags, which begin with a @, are used to identify particular information
- Examples of Javadoc tags:
  - @author
  - @version
  - @param
  - @return

Javadoc


Javadoc for a Method
begins a Javadoc comment
method description
tags
/**
* Retrieves the count of ...
* @param cat the category to match
* @return the number of ...
*/
public int getCount(Category cat)
{ ... }

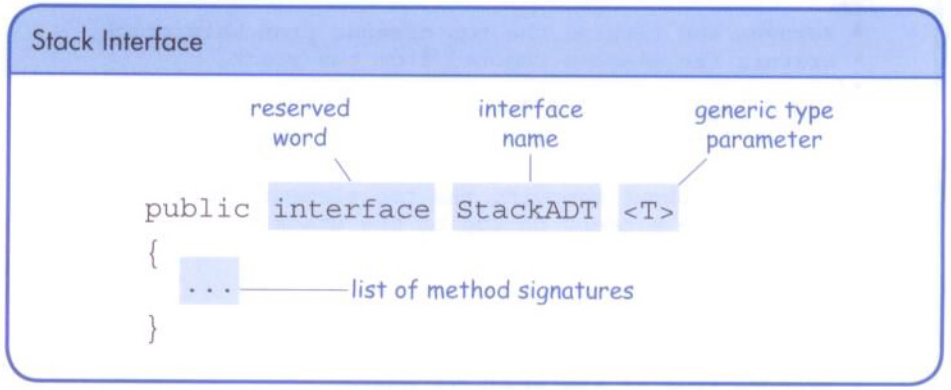
Exceptions

- When should exceptions be thrown from a collection class?
- Only when a problem is specific to the concept of the collection (not its implementation or its use)
- There's no need for the user of a collection to worry about it getting "full," so we'll take care of any such limitations internally
- But a stack should throw an exception if the user attempts to pop an empty stack

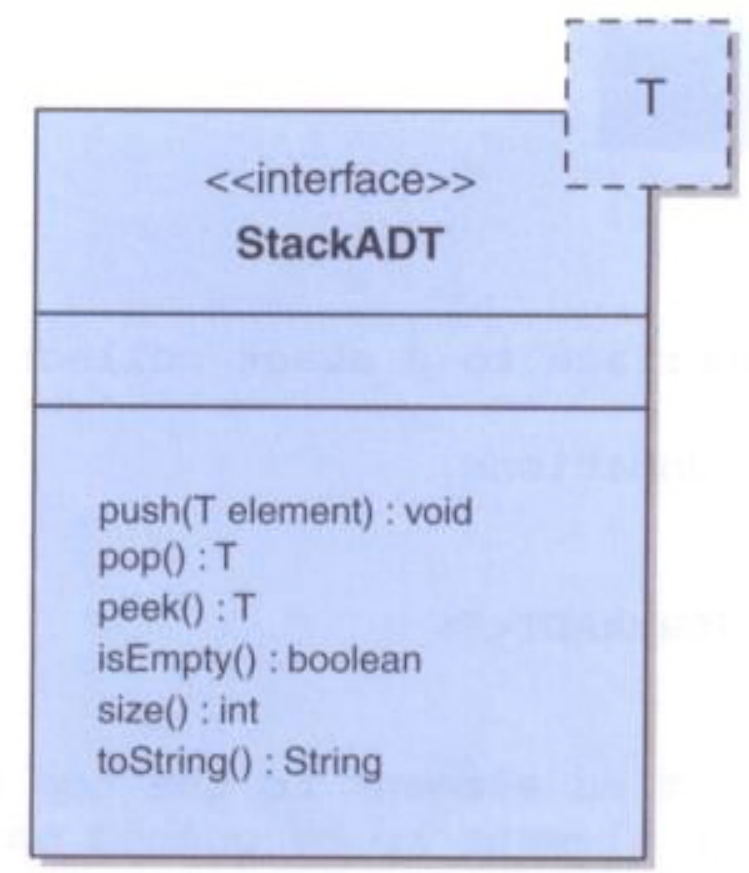
A Stack Interface

- Although the API version of a stack did not rely on a formal interface, we will define one for our own version
- To distinguish them, our collection interface names will have ADT (abstract data type) attached
- Furthermore, our collection classes and interfaces will be defined as part of a package called jsjf

```
package jsjf;

/**
 * Defines the interface to a stack collection.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface StackADT<T>
{
    /**
     * Adds the specified element to the top of this stack.
     * @param element element to be pushed onto the stack
     */
    public void push(T element);

    /**
     * Removes and returns the top element from this stack.
     * @return the element removed from the stack
     */
    public T pop();

    /**
     * Returns without removing the top element of this stack
     * @return the element on top of the stack
     */
    public T peek();

    /**
     * Returns true if this stack contains no elements.
     * @return true if the stack is empty
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this stack.
     * @return the number of elements in the stack
     */
    public int size();

    /**
     * Returns a string representation of this stack.
     * @return a string representation of the stack
     */
    public String toString();
}
```



A Stack Interface

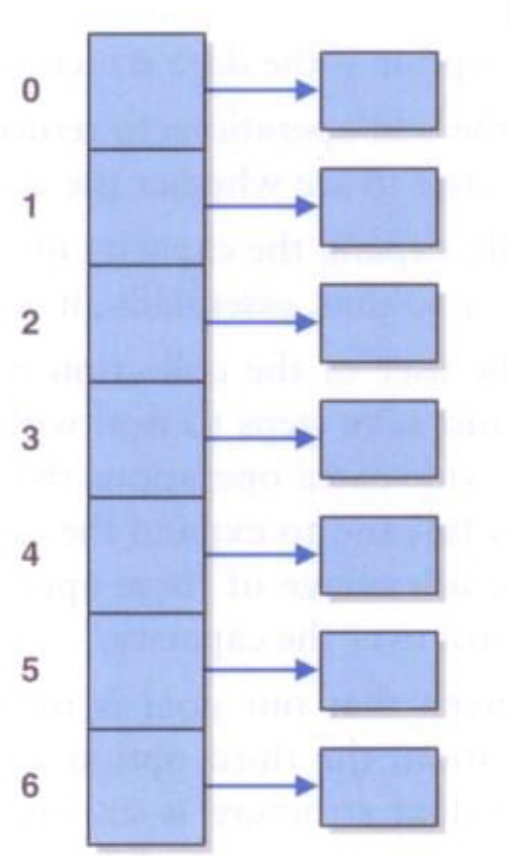


Implementing a Stack with an Array

- Let's now explore our own implementation of a stack, using an array as the underlying structure in which we'll store the stack elements
- We'll have to take into account the possibility that the array could become full
- And remember that an array of objects really stores references to those objects

Implementing Stacks with an Array

- An array of object references:



Managing Capacity

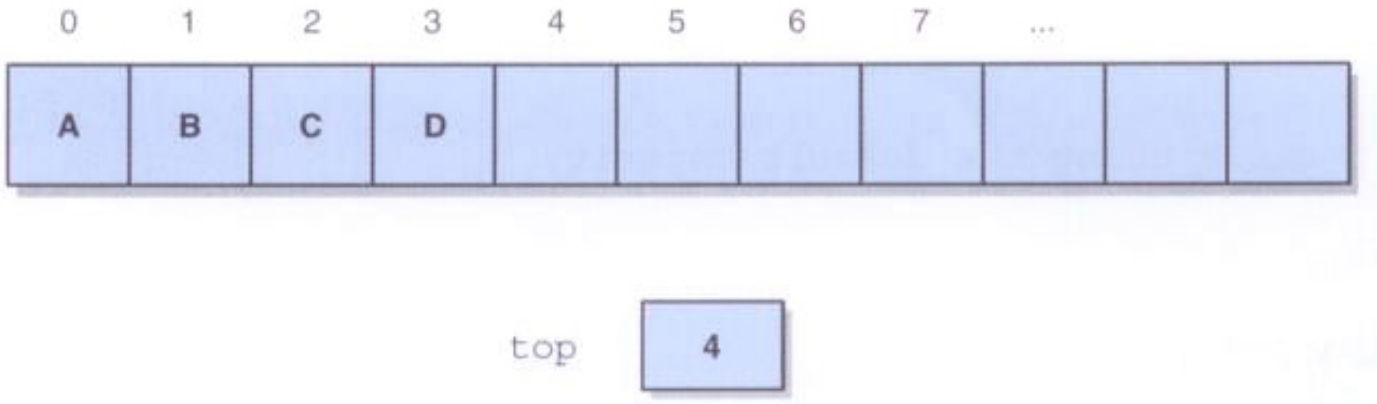
- The number of cells in an array is called its *capacity*
- It's not possible to change the capacity of an array in Java once it's been created
- Therefore, to expand the capacity of the stack, we'll create a new (larger) array and copy over the elements
- This will happen infrequently, and the user of the stack need not worry about it happening

Implementing Stacks with an Array

- By convention, collection class names indicate the underlying structure
  - So ours will be called `ArrayStack`
  - `java.util.Stack` is an exception to this convention
- Our solution will keep the bottom of the stack fixed at index 0 of the array
- A separate integer called `top` will indicate where the top of the stack is, as well as how many elements are in the stack currently

Implementing a Stack with an Array

- A stack with elements A, B, C, and D pushed on in that order:

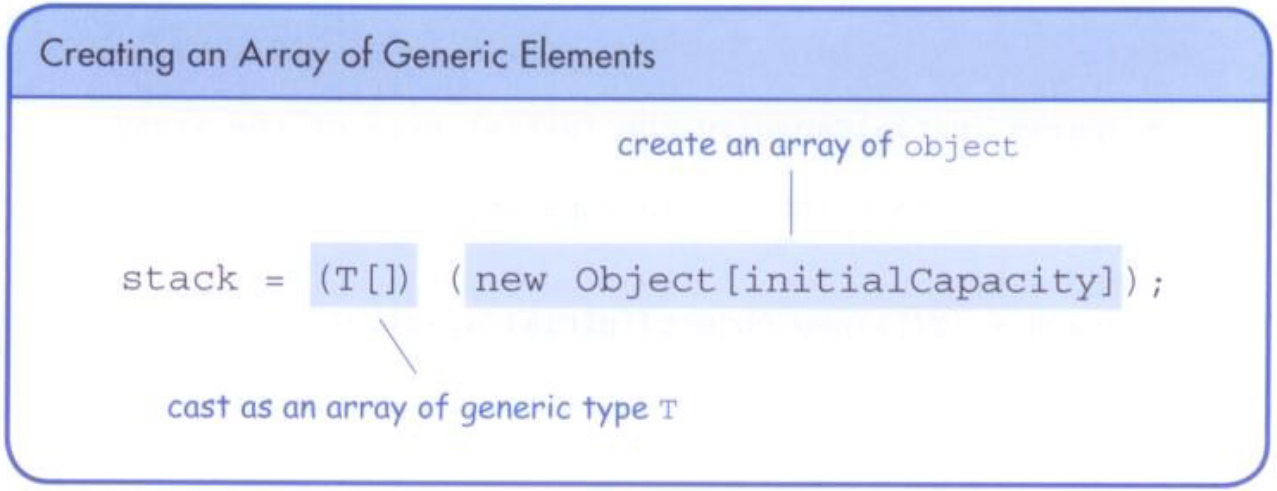


```
package jsjf;

import jsjf.exceptions.*;
import java.util.Arrays;

/**
 * An array implementation of a stack in which the
 * bottom of the stack is fixed at index 0.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ArrayStack<T> implements StackADT<T>
{
    private final static int DEFAULT_CAPACITY = 100;

    private int top;
    private T[] stack;
```

```
/**
 * Creates an empty stack using the default capacity.
 */
public ArrayStack()
{
    this(DEFAULT_CAPACITY);
}

/**
 * Creates an empty stack using the specified capacity.
 * @param initialCapacity the initial size of the array
 */
public ArrayStack(int initialCapacity)
{
    top = 0;
    stack = (T[])(new Object[initialCapacity]);
}
```

Creating an Array of a Generic Type

- You cannot instantiate an array of a generic type directly
- Instead, create an array of `Object` references and cast them to the generic type

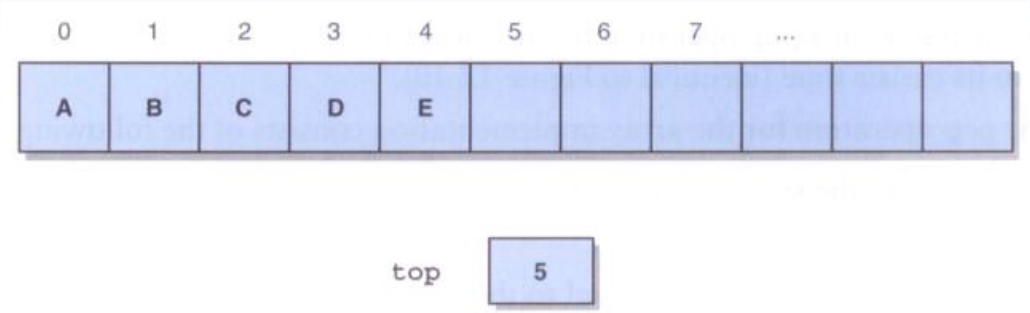


```
/**
 * Adds the specified element to the top of this stack, expanding
 * the capacity of the array if necessary.
 * @param element generic element to be pushed onto stack
 */
public void push(T element)
{
    if (size() == stack.length)
        expandCapacity();

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

/**
 * Creates a new array to store the contents of this stack with
 * twice the capacity of the old one.
 */
private void expandCapacity()
{
    stack = Arrays.copyOf(stack, stack.length * 2);
}
```

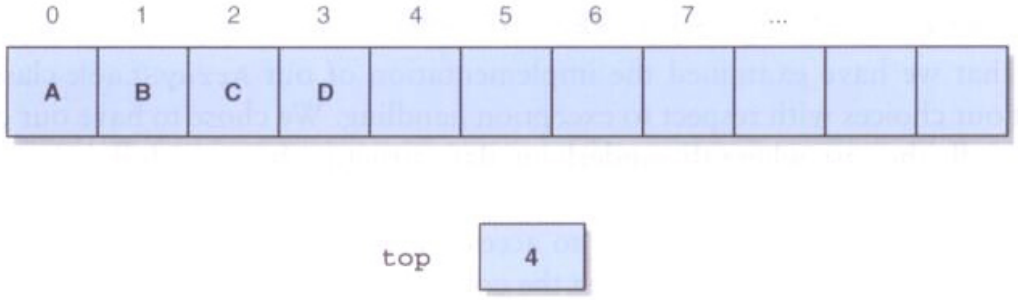
```
/**
 * Removes the element at the top of this stack and returns a
 * reference to it.
 * @return element removed from top of stack
 * @throws EmptyCollectionException if stack is empty
 */
public T pop() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("stack");

    top--;
    T result = stack[top];
    stack[top] = null;

    return result;
}

/**
 * Returns a reference to the element at the top of this stack.
 * The element is not removed from the stack.
 * @return element on top of stack
 * @throws EmptyCollectionException if stack is empty
 */
public T peek() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("stack");

    return stack[top-1];
}
```

Pushing and Popping a Stack

- After pushing element E:



- After popping:



Defining an Exception

- An `EmptyCollectionException` is thrown when a pop or a peek is attempted and the stack is empty
- The `EmptyCollectionException` is defined to extend `RuntimeException`
- By passing a string into its constructor, this exception can be used for other collections as well

```
package jsjf.exceptions;

/**
 * Represents the situation in which a collection is empty.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class EmptyCollectionException extends RuntimeException
{
    /**
     * Sets up this exception with an appropriate message.
     * @param collection the name of the collection
     */
    public EmptyCollectionException(String collection)
    {
        super("The " + collection + " is empty.");
    }
}
```