CHAPTER 6

Stacks

LEARNING OBJECTIVES

When you complete Chapter 6, you will be able to...

- follow and explain stack-based algorithms using the usual computer science terminology of push, pop, and peek.
- use a Java Stack class to implement stack-based algorithms such as the evaluation of arithmetic expressions.
- implement a Stack class of your own using either an array or a linked list data structure.

CHAPTER CONTENTS

The pushdown store is a "first in–last out" list. That is, symbols may be entered or removed only at the top of the list.

JOHN E. HOPCROFT AND JEFFREY D. ULLMAN
Formal Languages and Their Relation to Automata

This chapter introduces a data structure known as a *stack* or, as it is sometimes called, a *pushdown store*. It is a simple structure, even simpler than a linked list. Yet it turns out to be one of the most useful data structures known to computer science.

6.1 INTRODUCTION TO STACKS

The drawings in the right margin depict some stacks. There is a stack of pancakes, some stacks of coins, and a stack of books. A *stack* is an ordered collection of items that can be accessed only at one end. That may not sound like what you see in these drawings, but think for a moment. Each of the stacks is ordered from top to bottom; you can identify any item in a stack by saying it is the top item, second from the top, third from the top, and so on. Unless you mess up one of the neat stacks, you can access only the top item. To remove the bottom book from the stack, you must first remove the two books on top of it. The abstract definition of a stack reflects this intuition.

Stack Definition

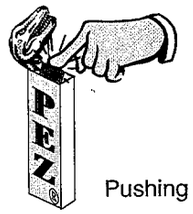
A **stack** is a data structure of ordered items such that items can be inserted and removed only at one end (called the **top**).

When we say that the items in a stack are *ordered,* all we mean is that there is one that can be accessed first (the one on top), one that can be accessed second (just below the top), a third one, and so forth. We do *not* require that the items can be compared using the < operator. The items can be of any type.

Stack items must be removed in the reverse order of that in which they are placed on the stack. For example, you can create a stack of books by first placing a dictionary, placing a thesaurus on top of the dictionary, and placing a novel on top of those so that the stack has the novel on top. When the books are removed, the novel must come off first (since it is on top), then the thesaurus, and finally the dictionary. Because of this property, a stack is called a "last-in/first-out" data structure (abbreviated LIFO).

LIFO
A stack is a last-in/first-out data structure. Items are taken out of the stack in the reverse order of their insertion.

Of course, a stack that is used in a program stores information rather than physical items such as books or pancakes. Therefore, it may help to visualize a stack as a pile of papers on which information is written. To place some information on the stack, you write the information on a new sheet of paper and place this sheet of paper on top of the stack. Getting information out of the stack is also accomplished by a simple operation since the top sheet of paper can be removed and read. There is just one restriction: Only the top sheet of paper is accessible. To read the third sheet from the top, for example, the top two sheets must be removed from the stack.



Pushing

A stack is analogous to a mechanism that is used in a popular candy holder called a *Pez® dispenser*, shown in the margin. The dispenser stores candy in a slot underneath an animal head figurine. Candy is loaded into the dispenser by pushing each piece into the hole. There is a spring under the candy with the tension adjusted so that, when the animal head is tipped backward, one piece of candy pops out. If this sort of mechanism were used as a stack data structure, the data would be written on the candy (which may violate some health laws, but it still makes a good analogy). Using this analogy, you can understand why adding an item to a stack is called a **push** operation and removing an item from a stack is called a **pop** operation.

Popping

The Stack Class—Specification

The key methods of a stack class are specified in Figure 6.1. Our specification lists a stack constructor and five methods. The most important methods are `push` (to add an item at the top of the stack) and `pop` (to remove the top item). Another method, called `peek`, allows a programmer to examine the top item without actually removing it. There are no methods that allow a program to access items other than the top. To access any item other than the top one, the program must remove items one at a time from the top until the desired item is reached.

FIGURE 6.1 Specification of the Key Methods of a Generic Stack Class

Specification

These are the key methods of a stack class, which we will implement in several different ways. Although this is a specification for a generic stack, we could also implement a stack that contains primitive values (such as `int`) directly. The Java Class Libraries also provide a generic stack class called `java.util.Stack`, which has these same key methods.

♦ **Constructor for the Generic Stack<E>**
`public Stack( )`
Initialize an empty stack.
Postcondition:

(FIGURE 6.1 continued)

♦ **isEmpty**
`public boolean isEmpty( )`
Determine whether this stack is empty.
Returns:
`true` if this stack is empty; otherwise `false`.

♦ **peek**
`public E peek( )`
Get the top item of this stack without removing the item.
Precondition:
This stack is not empty.
Returns:
the top item of the stack
Throws: `EmptyStackException`
Indicates that this stack is empty.

♦ **pop**
`public E pop( )`
Get the top item, removing it from this stack.
Precondition:
This stack is not empty.
Postcondition:
The return value is the top item of this stack, and the item has been removed.
Throws: `EmptyStackException`
Indicates that this stack is empty.

♦ **push**
`public void push(E item)`
Push a new item onto this stack. The new item may be the null reference.
Parameters:
`item` – the item to be pushed onto this stack
Postcondition:
The item has been pushed onto this stack.
Throws: `OutOfMemoryException`
Indicates insufficient memory for pushing a new item onto this stack.

♦ **size**
`public int size( )`
Accessor method to determine the number of items in this stack.
Returns:
the number of items in this stack

If a program attempts to pop an item off an empty stack, it is asking for the impossible; this error is called **stack underflow**. The pop method indicates a stack underflow by throwing an `EmptyStackException`. This exception is defined in `java.util.EmptyStackException`. To help you avoid a stack underflow, the class provides a method to determine whether a stack is empty. There is also a method to obtain the stack's current size.

We Will Implement a Generic Stack

The stacks implemented in this chapter are generic stacks

Later we will implement the stack in several different ways. Some of our implementations will have extra methods beyond the five given in the figure. Also, each of our implementations will be a *generic* stack that depends on an unspecified data type for the stack's elements. We'll call the class `Stack`, but just like any generic class, it can only be used with Java objects. For example, we can't put primitive `int` values into our `Stack` without a boxing conversion. Because of this, it might sometimes be useful to have a stack that contains primitive values directly. Although we won't show those simpler stacks in this chapter, you can find source code for them in this book's online resources at `http://www.cs.colorado.edu/~main/dsoj.html`.

The Java Class Libraries also provide a stack of objects called `java.util.Stack`, with the same five methods from Figure 6.1.

PROGRAMMING EXAMPLE: Reversing a Word

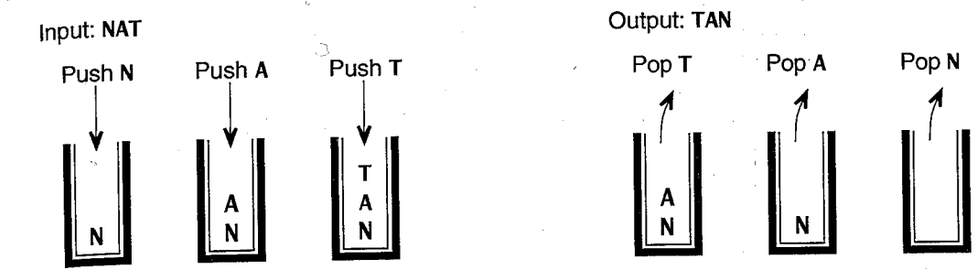
uses for stacks

Stacks are very intuitive—even cute—but are they good for anything? Surprisingly, they have many applications. Most compilers use stacks to analyze the syntax of a program. Stacks are used to keep track of local variables when a program is run. Stacks can be used to search a maze or a family tree or other types of branching structures. In this book, we will discuss examples related to each of these applications. But before we present any complicated applications of the stack ADT, let us first practice with a simple problem so that we can see how a stack is used.

Suppose you want a program to read in a word and then write it out backward. If the program reads in `NAT`, then it will output `TAN`. If it reads in `TAPS`, it will output `SPAT`. The author Roald Dahl wrote a book called `ESIOTROT`, which our program converts to `TORTOISE`. One way to accomplish this task is to read the input one letter at a time and place each letter in a stack of characters. After the word is read, the letters in the stack are written out, but because of the way a stack works, they are written out in reverse order. The outline is shown here:

```
// Reversing the spelling of a word
Declare a stack of characters;
while (there are more characters of the word to read)
    Read a character and push the character onto the stack.
while (the stack is not empty)
    Pop a character off the stack and write that character to the screen.
```


FIGURE 6.2 Using a Stack to Reverse Spelling



This computation is illustrated in Figure 6.2. At all times in the computation, the only available item is on "top." Figure 6.2 suggests another intuition for thinking about a stack. You can view a stack as a hole in the ground and view the items as being placed in the hole one on top of the other. To retrieve an item, you must first remove the items on top of it.

Self-Test Exercises for Section 6.1

1. Suppose a program uses a stack of characters to read in a word and then write the word out backward as described in this section. Now suppose the input word is `DAHL`. List all the activations of the `push` and `pop` methods. List them in the order in which they will be executed and indicate the character that is pushed or popped. What is the output?
2. Consider the stack class given in Figure 6.1 on page 306. The `peek` method lets you look at the top item in the stack without changing the stack. Describe how you can define a new method that returns the second item from the top of the stack without permanently changing the stack. (If you temporarily change the stack, then change it back before the method ends.) Your description will be in terms of `peek`, `pop`, and `push`. Give your solution in pseudocode, not in Java.

6.2 STACK APPLICATIONS

As an an exercise to learn about data structures, we will eventually implement the stack class ourselves. For now, though, we'll look at some example applications that use Java's generic stack from `java.util.Stack`. Our first example uses a stack that contains characters to analyze a string of parentheses.

PROGRAMMING EXAMPLE: Balanced Parentheses

Later in this chapter, we will describe how stacks can be used to evaluate arithmetic expressions. At the moment, we will describe a simpler method called `isBalanced` that does a closely related task. The algorithm checks an expres-

sion to see if the parentheses match correctly. It allows three kinds of parentheses: `( )`, `[ ]`, or `{ }`. Any symbol other than one of these parentheses is ignored.

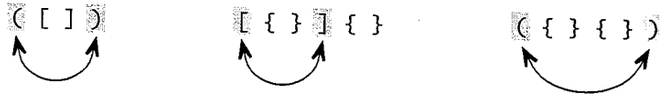
For example, consider the string `"{[X + Y*(Z + 7)]*(A + B)}"`. Each of the left parentheses has a corresponding right parenthesis. Also, as the string is read from left to right, there is never an occurrence of a right parenthesis that cannot be matched with a corresponding left parenthesis. Therefore, the activation of `isBalanced("{[X + Y*(Z + 7)]*(A + B)}")` returns `true`.

On the other hand, consider the string `"((X + Y*{Z + 7}*[A + B])"`. The parentheses around the subexpression `Z + 7` match each other, as do the parentheses around `A + B`. And one of the left parentheses in the expression matches the final right parenthesis. But the other left parenthesis has no matching right parenthesis. Hence, `isBalanced("((X + Y*{Z + 7}*[A + B])")` returns `false`.

The technique used is simple: The algorithm scans the characters of the string from left to right. Every time a left parenthesis occurs, it is pushed onto the stack. Every time a right parenthesis occurs, a matching left parenthesis is popped off the stack. If the correct kind of left parenthesis is not on top of the stack, then the string is unbalanced. For example, when a curly right parenthesis `'}'` occurs in the string, the matching curly left parenthesis `'{'` should be on top of the stack.

All symbols other than parentheses are ignored. If all goes smoothly and the stack is empty at the end of the expression, then the parentheses match. On the other hand, three things might go wrong: (1) The stack is empty when the algorithm needs to pop a symbol; or (2) The wrong kind of parenthesis appears at some point; or (3) Symbols are still in the stack after all the input has been read. In each of these cases, the parentheses are not balanced.

Let's look at some examples. Since no symbols other than parentheses can affect the results, we will use expressions of just parentheses symbols. All of the following are balanced (shading and arrows help find matching parentheses):



If you think about these examples, you can begin to understand the algorithm. In the first example, the parentheses match because they have the same number of left and right parentheses, but the algorithm does more than just count parentheses. The algorithm actually matches parentheses. Every time it encounters a `')'`, the symbol it pops off the stack is the matching `'('`. When it encounters a `']'`, the symbol it pops off the stack is the matching `'['`. When it encounters a `'}'`, the symbol it pops off the stack is the matching `'{'`.

The complete sequences of stack configurations from two executions of the algorithm are shown in Figure 6.3. The stacks shown in the figure show the configuration after processing each character of the expression.

In general, the stack works by keeping a stack of the unmatched left parentheses. Every time the algorithm encounters a right parenthesis, the corresponding left parenthesis is deleted (popped) from the stack. If the parentheses in the input match correctly, things work out perfectly, and the stack is empty at the end of the input line.

The balancing algorithm is implemented by the `isBalanced` method of Figure 6.4. Notice that the method uses a stack of `Character` objects. We can push ordinary `char` values into the stack (through autoboxing), and we can use the popped values as if they were ordinary `char` values (through auto-unboxing). One technique in the implementation may be new to you: acting on the string's next character via a *switch* statement. We'll discuss this technique after you've looked through the program.

FIGURE 6.3 Stack Configurations for the Parentheses Balancing Algorithm

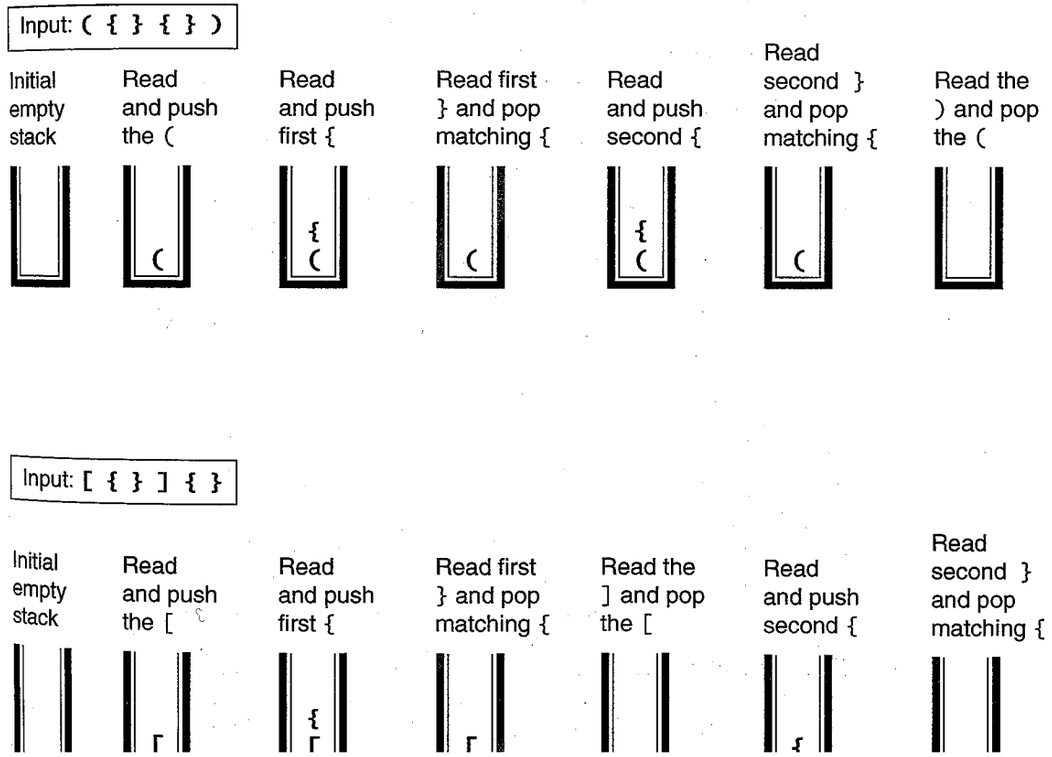


FIGURE 6.4 A Method to Check for Balanced Parentheses

A Method Implementation

```
public static boolean isBalanced(String expression)
// Postcondition: A true return value indicates that the parentheses in the
// given expression are balanced. Otherwise, the return value is false.
// Note that characters other than ( ), { }, and [ ] are ignored.
{
   // Meaningful names for characters
   final char LEFT_NORMAL  = '(';
   final char RIGHT_NORMAL = ')';
   final char LEFT_CURLY   = '{';
   final char RIGHT_CURLY  = '}';
   final char LEFT_SQUARE  = '[';
   final char RIGHT_SQUARE = ']';

   Stack<Character> store = new Stack<Character>; // From java.util.Stack
   int i;                                         // An index into the string
   boolean failed = false;                        // Change to true for a mismatch

   for (i = 0; !failed && (i < expression.length( )); i++)
   {
      switch (expression.charAt(i))
      {
         case LEFT_NORMAL:
         case LEFT_CURLY:
         case LEFT_SQUARE:
            store.push(expression.charAt(i));
            break;
         case RIGHT_NORMAL:
            if (store.isEmpty( ) || (store.pop( ) != LEFT_NORMAL))
               failed = true;
            break;
         case RIGHT_CURLY:
            if (store.isEmpty( ) || (store.pop( ) != LEFT_CURLY))
               failed = true;
            break;
         case RIGHT_SQUARE:
            if (store.isEmpty( ) || (store.pop( ) != LEFT_SQUARE))
               failed = true;
            break;
      }
   }

   return (store.isEmpty( ) && !failed);
}
```


PROGRAMMING TIP

THE SWITCH STATEMENT

The for-loop in Figure 6.4 processes character number `i` of the expression during each iteration. There are several possible actions, depending on what kind of character appears at `expression.charAt(i)`. An effective statement to select among many possible actions is the switch statement, with the general form:

```
switch (<Control value>)
{
   <Body of the switch statement>
}
```

When the switch statement is reached, the control value is evaluated. The program then looks through the body of the switch statement for a matching case label. For example, if the control value is the character 'A', then the program looks for a case label of the form `case 'A':`. If a matching case label is found, then the program goes to that label and begins executing statements. Statements are executed one after another—but if a **break** statement (of the form `break;`) occurs, then the program skips to the end of the body of the switch statement.

If the control value has no matching case label, then the program will look for a **default label** of the form `default:`. This label handles any control values that don't have their own case label.

If there is no matching case label and no default label, then the whole body of the switch statement is skipped.

For the `isBalanced` method of Figure 6.4, the switch statement has one case label for each of the six possible kinds of parentheses. The three kinds of left parentheses are all handled together by putting their case statements one after another. Each of the right parentheses is handled with its own case statement. For example, one of the right parentheses is the character `RIGHT_NORMAL`, which is an ordinary right parenthesis `')'`. The `RIGHT_NORMAL` character is handled as shown here:

```
switch (expression.charAt(i))
{
   ...
   case RIGHT_NORMAL:
      if (store.isEmpty( ) || (store.pop( ) != LEFT_NORMAL))
         failure = true;
      break;
   ...
}
```

Evaluating Arithmetic Expressions

In this next programming example, we will design and write a calculator program. This will be an example of a program that uses two stacks—one is a stack of characters, and the other is a stack of double numbers.

Evaluating Arithmetic Expressions—Specification

input to the calculator program

The program takes as input a fully parenthesized numeric expression such as the following:

```
((((12 + 9)/3) + 7.2)*((6 - 4)/8))
```

The expression consists of integers or double numbers, together with the operators +, -, *, and /. To focus on the use of the stack (rather than on input details), we require that each input number be non-negative. (Otherwise, it is hard to distinguish the subtraction operator from a minus sign that is part of a negative number.) We will assume that the expression is formed correctly so that each operation has two arguments. Finally, we will also assume that the expression is fully parenthesized with ordinary parentheses `'('` and `')'`, meaning that each operation has a pair of matched parentheses surrounding its arguments. We can later enhance our program so that these assumptions are no longer needed.

output of the calculator program

The output will simply be the value of the arithmetic expression.

Evaluating Arithmetic Expressions—Design

Most of the program's work will be carried out by a method that reads one line of input and evaluates that line as an arithmetic expression. To get a feel for the problem, let's start by doing a simple example by hand. Consider the following expression:

do an example by hand

```
(((6 + 9)/3)*(6 - 4))
```

If we were to evaluate this expression by hand, we might first evaluate the innermost expressions, `(6 + 9)` and `(6 - 4)`, to produce the smaller expression:

```
((15/3)*2)
```

Next we would evaluate the expression `(15/3)` and replace this expression with its value of `5`. That would leave us with the expression `(5*2)`. Finally, we would evaluate this last operation to get the answer of `10`.

To convert this intuitive approach into a fully specified algorithm that can be implemented, we need to do things in a more systematic way: We need a specific way to find the expression to be evaluated next and a way to remember the results of our intermediate calculations.

First let's find a systematic way of choosing the next expression to be evaluated. (After that, we can worry about how we will keep track of the intermediate results.) We know that the expression to be evaluated first must be one of the innermost expressions—which is a subexpression that has just one operation. Let's decide to evaluate the leftmost of these innermost expressions. For instance, consider our example of:


```
(((6 + 9)/3)*(6 - 4))
```

The innermost expressions are `(6 + 9)` and `(6 - 4)`, and the leftmost one of these is `(6 + 9)`. If we evaluate this *leftmost of the innermost expressions*, we obtain:

```
((15/3)*(6 - 4))
```

We could now go back and evaluate the other innermost expression `(6 – 4)`, but why bother? There is a simpler approach that spares us the trouble of remembering any other expressions. After we evaluate the leftmost of the innermost expressions, we are left with another simpler arithmetic expression, namely `((15/3)*(6 – 4))`, so we can simply repeat the process with this simpler expression: We again evaluate the leftmost of the innermost expressions of our new simpler expression. The entire process will look like the following:

1. Evaluate the leftmost of the innermost expressions in

   ```
   (((6 + 9)/3)*(6 - 4))
   ```

   to produce the simpler expression `((15/3)*(6 - 4))`.

2. Evaluate the leftmost of the innermost expressions in

   ```
   ((15/3)*(6 - 4))
   ```

   to produce the simpler expression `(5*(6 - 4))`.

3. Evaluate the leftmost of the innermost expressions in

   ```
   (5*(6 - 4))
   ```

   to produce the simpler expression `(5 * 2)`.

4. Evaluate the leftmost of the innermost expressions in

   ```
   (5 * 2)
   ```

   to obtain the final answer of `10`.

translating the hand method to an algorithm

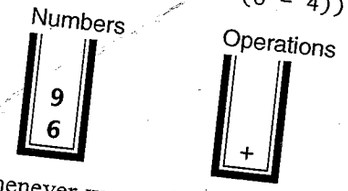
This method works fine with pencil and paper, but the algorithm must read the input one character at a time from left to right. How does the algorithm find *the leftmost of the innermost expressions*? Look at the preceding example. The end of the expression to be evaluated is always a right parenthesis `')'`, and moreover, it is always the *first* right parenthesis. After evaluating one of these innermost

is no need to back up; to find the next right parenthesis, we can just keep reading left to right from where we left off. The next right parenthesis will indicate the end of the next expression to be evaluated.

Now we know how to find the expression to be evaluated next, but how do we keep track of our intermediate values? For this, we use two stacks. One stack will contain numbers; there will be numbers from the input as well as numbers that were computed when subexpressions were evaluated. The other stack will hold symbols for the operations that still need to be evaluated. Because a stack processes data in a last-in/first-out manner, it will turn out that the correct two numbers are on the top of the number stack at the same time that the appropriate operation is at the top of the stack of operations. To better understand how the process works, let's evaluate our sample expression one more time, this time using the two stacks.

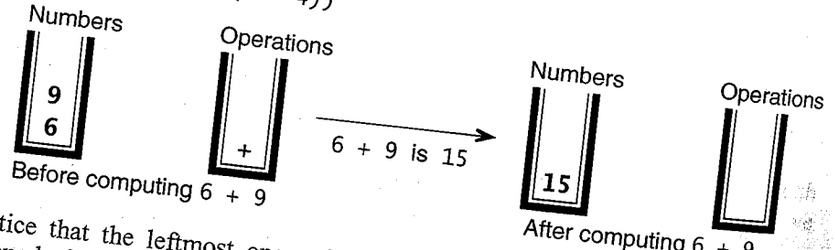
We begin by reading up to the first right parenthesis; the numbers we encounter along the way are pushed onto the number stack, and the operations we encounter along the way are pushed onto the operation stack. When we reach the first right parenthesis, our two stacks look like this:

Characters read so far (shaded):
(((6 + 9) / 3) * (6 - 4))



Whenever we reach a right parenthesis, we combine the top two numbers (on the number stack) using the topmost operation (on the character stack). In our example, we compute 6 + 9, yielding 15, and this number 15 is pushed back onto the number stack:

Characters read so far (shaded):
(((6 + 9) / 3) * (6 - 4))

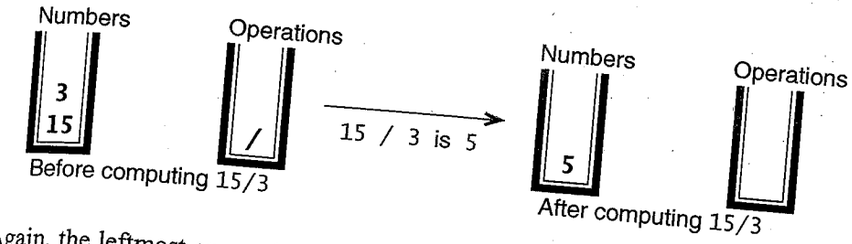


Notice that the leftmost operand (6 in this example) is the *second* number popped off the stack. For addition, this does not matter—who cares whether we have added 6 + 9 or 9 + 6? But the order of the operands does matter for subtraction and division.

Next we simply continue the process: by reading up to the next right parenthesis, pushing the numbers we encounter onto the number stack, and pushing the

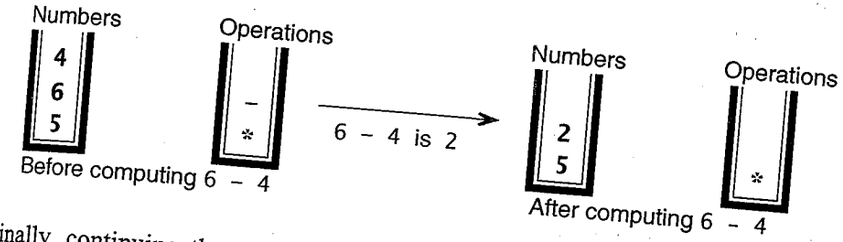
operations we encounter onto the operation stack. When we reach the next right parenthesis, we combine the top two numbers using the topmost operation. Here's what happens in our example when we reach the second right parenthesis:

Characters read so far (shaded):
(((6 + 9) / 3) * (6 - 4))



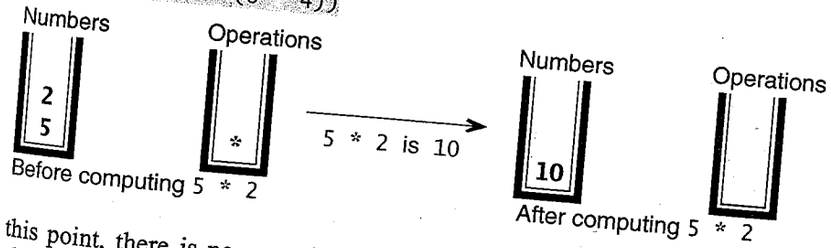
Again, the leftmost operand (15) is the second number popped off the stack, so the correct evaluation is 15/3, not 3/15. Continuing the process, we obtain:

Characters read so far (shaded):
(((6 + 9) / 3) * (6 - 4))



Finally, continuing the process one more time does not add anything to the stacks, but it does read the last right parenthesis and does combine two numbers from the numbers stack with an operation from the operation stack:

Characters read so far (shaded):
(((6 + 9) / 3) * (6 - 4))



At this point, there is no more input, and there is exactly one number in the number stack, namely 10. That number is the answer. Notice that when we used the two stacks, we performed the exact same evaluations as we did when we first evaluated this expression in a simple pencil-and-paper fashion.

cases for evaluating an arithmetic expression

To evaluate our expression, we only need to repeatedly handle the input items according to the following cases:

Numbers. When a number is encountered in the input, the number is read and pushed onto the numbers stack. We allow the numbers to be any double number, but we do not allow a + or - sign at the front of the number. So, we can have numbers such as `42.8` and `0.34`, but not `-42.8` or `-0.34`. This restriction is because we haven't yet developed the algorithm for distinguishing between a + or - sign that is part of a number and a + or - sign that is an arithmetic addition or subtraction.

Operation Characters. When one of the four operation characters is encountered in the input, the character is read and pushed onto the operations stack.

Right Parenthesis. When a right parenthesis is read from the input, an "evaluation step" takes place. The step pops the top two numbers from the number stack and pops the top operation from the operation stack. The two numbers are combined using the operation (with the second number popped as the left operand). The result of the operation is pushed back onto the numbers stack.

Left Parenthesis or Blank. The only other characters that appear in the input are left parentheses and blanks. These are read and thrown away, not affecting the computation. A more complete algorithm would need to process the left parentheses in some way to ensure that each left parenthesis is balanced by a right parenthesis, but for now we are assuming that the input is completely parenthesized in a proper manner.

The processing of input items halts when the end of the input line occurs, indicated by `'\n'` in the input. At this point, the answer is the single number that remains in the number stack.

We now have our algorithm, which we plan to implement as a method called `evaluate`. The parameter to evaluate is a string that provides the arithmetic expression. The return value is the value of the arithmetic expression as a double number. For example, `evaluate("(((60 + 40)/50) * (16-4))")` returns 24.0.

Implementation of the Evaluate Method

Our implementation of `evaluate` requires some understanding of the `Scanner` class from Appendix B. This class is often attached to keyboard input, but in our evaluate implementation, we create a Scanner called `input` that contains all the characters from the arithmetic expression. This allows us to more easily read the expression, detecting which parts are numbers and which parts are operations. In particular, we used these items:

- The method `input.hasNext( )` returns `true` if there are still more parts of the expression to be processed.

- The expression `input.hasNext(UNSIGNED_DOUBLE)` is true if the next part of the input expression is a double number (with no + or - sign in front). This uses a constant `UNSIGNED_DOUBLE` that is defined and discussed in Appendix B.
- The statement `next = input.findInLine(UNSIGNED_DOUBLE)` sets the string `next` equal to the next part of the input expression, which must be a double number. Once the double number is in the string next, we can convert it to a Double value and push it onto the stack with:

```
numbers.push(new Double(next));
```

- Similarly, `next = input.findInLine(CHARACTER)` sets `next` equal to the next single character (skipping spaces) in the input expression.

Our implementation also uses one other methods, evaluateStackTops, which appears along with the evaluate method in Figure 6.5. In order to compile these methods, you will need to include the definitions of UNSIGNED_DOUBLE and CHARACTER from Appendix B; the code also requires these import statements:

```
import java.util.Stack;     // Provides the generic Stack class
import java.util.Scanner;   // Provides the Scanner class
import java.util.Pattern;   // Provides the Pattern class
```

FIGURE 6.5 A Method to Evaluate a Fully Parenthesized Arithmetic Expression

Method Specification and Implementation

♦ **evaluate**

`public static double evaluate(String expression)`

The `evaluate` method evaluates the arithmetic expression.

Parameters:

`expression`– a fully parenthesized arithmetic expression

Precondition:

The expression must be a fully parenthesized arithmetic expression formed from double numbers (with no + or minus sign in front), any of the four arithmetic operations (+ - * /), and spaces.

Returns:

the value of the arithmetic expression

Throws: `IllegalArgumentException`

Indicates that the expression had the wrong format.

(continued)

(FIGURE 6.5 continued)

```
public static double evaluate(String expression)
{
   // Two generic stacks to hold the expression's numbers and operations:
   Stack<Double> numbers = new Stack<Double>( );
   Stack<Character> operations = new Stack<Character>( );

   // Convert the expression to a Scanner for easier processing.
   // The next String holds the next piece of the expression: a number, operation or parenthesis
   Scanner input = new Scanner(expression);
   String next;

   while (input.hasNext( ))
   {
      if (input.hasNext(UNSIGNED_DOUBLE))
      {  // The next piece of the expression is a number
         next = input.findInLine(UNSIGNED_DOUBLE);
         numbers.push(new Double(next));
      }
      else
      {  // The next piece of the input is an operation (+ - * or /) or a parenthesis
         next = input.findInLine(CHARACTER);
         switch (next.charAt(0))
         {
         case '+': // Addition
         case '-': // Subtraction
         case '*': // Multiplication
         case '/': // Division
            operations.push(next.charAt(0));
            break;
         case ')': // Right parenthesis (the evaluateStackTops function is on the next page)
            evaluateStackTops(numbers, operations);
            break;
         case '(': // Left parenthesis
            break;
         default : // Illegal character
            throw new IllegalArgumentException("Illegal character");
         }
      }
   }
   if (numbers.size( ) != 1)
      throw new IllegalArgumentException("Illegal input expression");
   return numbers.pop( );
}
```

This code requires java.util.Stack, java.util.Scanner and java.util.Pattern.

See Appendix B for UNSIGNED_DOUBLE and CHARACTER that we use here to simplify reading from a Scanner

(continued)

(FIGURE 6.5 continued)

Method Specification and Implementation

♦ **evaluateStackTops**

```
public static void evaluateStackTops
   (Stack<Double> numbers, Stack<Character> operations)
```

This method applies an operation to two numbers taken from the numbers stack.

Precondition:
There must be at least two numbers on the numbers stack, and the top character on the operations stack must be the character '+' '-' '*' or '/'.

Postcondition:
The top two numbers have been popped from the numbers stack, and the top operation has been popped from the operations stack. The two numbers have been combined using the operation (with the second number popped as the left operand).

Throws: `IllegalArgumentException`
Indicates that the stacks fail the precondition.

```
public static void evaluateStackTops
(Stack<Double> numbers, Stack<Character> operations)
{
   double operand1, operand2;

   // Check that the stacks have enough items, and get the two operands.
   if ((numbers.size( ) < 2) || (operations.isEmpty( )))
      throw new IllegalArgumentException("Illegal expression");
   operand2 = numbers.pop( );
   operand1 = numbers.pop( );

   // Carry out an action based on the operation on the top of the stack.
   switch (operations.pop( ))
   {
      case '+': numbers.push(operand1 + operand2);
                break;
      case '-': numbers.push(operand1 - operand2);
                break;
      case '*': numbers.push(operand1 * operand2);
                break;
      case '/': // Note: A division by zero is possible. The result would be one of the
                // constants Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY.
                numbers.push(operand1 / operand2);
                break;
      default : throw new IllegalArgumentException("Illegal operation");
   }
}
```

Evaluating Arithmetic Expressions—Testing and Analysis

testing

As usual, you should test your program on boundary values that are most likely to cause problems. For this program, one kind of boundary value consists of the simplest kind of expressions: those that combine only two numbers. To test that operations are performed correctly, you should test simple expressions for each of the operations +, −, *, and /. These simple expressions should have only one operation. Be sure to test the division and subtraction operations carefully to ensure that the operations are performed in the correct order. After all, `3/15` is not the same as `15/3`, and `3 – 15` is not the same as `15 – 3`. These are perhaps the only boundary values. But it is important also to test some cases with nested parentheses, and you can test an illegal division such as `15/0`. What does the program do? It throws an `IllegalArgumentException` in the `evaluateStackTops` method.

time analysis

Let's estimate the number of operations that our program will use on an expression of length n. We will count each of the following as one program operation: reading or peeking at a symbol, performing one of the arithmetic operations (+, −, *, or /), pushing an item onto one of the stacks, and popping an item off of one of the stacks. We consider each kind of operation separately.

Time Spent Reading Characters. There are only n symbols in the input, so the program can read at most n symbols. No character is "peeked" at more than once either, so this aspect of the program has no more than $2n$ operations.

Time Spent Evaluating Arithmetic Operations. Each arithmetic operation performed by the program is the evaluation of an operation symbol in the input. Because there are no more than n arithmetic operations in the input, there are at most n arithmetic operations performed. In actual fact, there are far fewer than n operations since many of the input symbols are digits or parentheses. But there are certainly no more than n arithmetic operation symbols, so it is safe to say that there are no more than n arithmetic operations performed.

Number of Push Operations. Since there are no more than n arithmetic operation symbols, we know that there are at most n operation symbols pushed onto the operation stack. The number stack may contain input numbers and numbers obtained from evaluating arithmetic expressions. Again, an upper bound will suffice: There are at most n input numbers and at most n arithmetic operations evaluated. Thus, at most, $2n$ numbers are pushed onto the number stack. This gives an upper bound of $3n$ total push operations onto the stacks.

Number of Pop Operations. Once we know the total number of items that are pushed onto the two stacks, we have a bound on how many things can be popped off of the two stacks. After all, you cannot pop off an item unless it was first pushed onto the stack. Thus, there is an upper bound of $3n$ pop operations from any of the stacks.

Total Number of Operations. Now let's total things up. The total number of operations is no more than $2n$ reads/peeks, plus n arithmetic operations performed, plus $3n$ items pushed onto a stack, plus $3n$ items popped off of a stack—for a grand total of $9n$. The actual number of operations will be less than this because we have used generous upper bounds in several estimates, but $9n$ is enough to conclude that the algorithm for this program is $O(n)$; this is a linear algorithm in the number of stack operations.

Evaluating Arithmetic Expressions—Enhancements

The program in Figure 6.5 on page 319 is a fine example of how to use stacks. As a computer scientist, you will find yourself using stacks in this manner in many different situations. However, the program is not a fine example of a finished program. Before we can consider it to be a finished product, we need to add a number of enhancements to make the program more robust and friendly.

Some enhancements are easy. It is useful (and easy) to write a main program that repeatedly gets and evaluates arithmetic expressions. Another nice enhancement would be to permit expressions that are not fully parenthesized and to use the Java precedence rules to decide the order of operations when parentheses are missing. We will discuss topics related to this enhancement in Section 6.4, where (surprise!) we'll see that a stack is useful for this purpose, too.

Self-Test Exercises for Section 6.2

3. How would you modify the calculator program in Figure 6.5 on page 319 to allow the symbol ^ to be used for exponentiation? Describe the changes; do not write out the code.
4. How would you modify the calculator program in Figure 6.5 on page 319 to allow for comments in the calculator input? Comments appear at the end of the expression, starting with a double slash `//` and continuing to the end of the line. Describe the changes; do not write out the code.
5. Write some illegal expressions that are caught by the calculator program in Figure 6.5 on page 319, resulting in an `IllegalArgumentException`.
6. Write some illegal expressions that are not caught by the calculator program in Figure 6.5 on page 319.
7. Carry out the operations of `evaluate` by hand on the input expression `( ( (60 + 40)/50) * (16 - 4) )`. Draw the two stacks after each push or pop.
8. What kind of expressions cause the `evaluation` stacks to grow large?
9. What is the time analysis, in big-O notation, of the `evaluation` method? Explain your reasoning.

6.3 IMPLEMENTATIONS OF THE STACK ADT

We will give two implementations of our generic stack class: an implementation using an array and an implementation using a linked list. Each implementation will be for a generic class but as we discussed earlier, we could just as easily implement the stack to hold one of Java's primitive types.

Array Implementation of a Stack

Figure 6.6 gives the specification and implementation of a generic `ArrayStack` class that includes all the earlier methods we mentioned (from Figure 6.1 on page 306). The class stores its items in an array, so the class also has extra methods for explicitly dealing with capacity. The class implementation uses two instance variables described here:

Invariant of the ArrayStack Class

1. The number of items in the stack is stored in the instance variable `manyItems`.
2. The items in the stack are stored in a partially filled array called `data`, with the bottom of the stack at `data[0]`, the next item at `data[1]`, and so on to the top of the stack at `data[manyItems-1]`.

the stack items are stored in a partially filled array

In other words, our stack implementation is simply a partially filled array implemented in the usual way: an array and a variable to indicate how much of the array is being used. The stack bottom is at `data[0]`, and the top position is the last array position used. Each method (except the constructor) can assume that the stack is represented in this way when the operation is activated. Each method has the responsibility of ensuring that the stack is still represented in this manner when the method finishes.

implementing the stack operations

The methods that operate on our stack are now straightforward. To initialize the stack, set the instance variable `manyItems` to zero, indicating an empty array and hence an empty stack. To add an item to the stack (in the `push` method), we store the new item in `data[manyItems]`, and then we increment `manyItems` by one. To look at the top item in the stack (method `peek`), we simply look at the item in array position `data[manyItems-1]`. To remove an item from the stack (in the `pop` method), we decrement `manyItems` and then return the value of `data[manyItems]` (which is the value that was on top prior to the pop operation). The methods to test for emptiness and to return the size work by examining the value of `manyItems`.

FIGURE 6.6 Specification and Implementation of the Array Version of the Generic Stack Class

Generic Class ArrayStack

❖ **public class ArrayStack<E> from the package edu.colorado.collections**

An `ArrayStack<E>` is a stack of references to `E` objects.

Limitations:

(1) The capacity of one of these stacks can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructors, `clone`, `ensureCapacity`, `push`, and `trimToSize` will result in an `OutOfMemoryError` when free memory is exhausted.

(2) A stack's capacity cannot exceed the largest integer 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.

Specification

♦ **Constructor for the ArrayStack<E>**

`public ArrayStack( )`

Initialize an empty stack with an initial capacity of 10. Note that the `push` method works efficiently (without needing more memory) until this capacity is reached.

Postcondition:

This stack is empty and has an initial capacity of 10.

Throws: `OutOfMemoryError`

Indicates insufficient memory for: `new Object[10]`.

♦ **Second Constructor for the ArrayStack<E>**

`public ArrayStack(int initialCapacity)`

Initialize an empty stack with a specified initial capacity. Note that the `push` method works efficiently (without needing more memory) until this capacity is reached.

Parameters:

`initialCapacity` – the initial capacity of this stack

Precondition:

`initialCapacity` is non-negative.

Postcondition:

This stack is empty and has the given initial capacity.

Throws: `IllegalArgumentException`

Indicates that `initialCapacity` is negative.

Throws: `OutOfMemoryError`

Indicates insufficient memory for: `new Object[initialCapacity]`.

(continued)

(FIGURE 6.6 continued)

♦ **clone**

```
public ArrayStack<E> clone( )
```

Generate a copy of this stack.

Returns:
The return value is a copy of this stack. Subsequent changes to the copy will not affect the original, nor vice versa.

Throws: `OutOfMemoryError`
Indicates insufficient memory for creating the clone.

♦ **ensureCapacity**

```
public void ensureCapacity(int minimumCapacity)
```

Change the current capacity of this stack.

Parameters:
`minimumCapacity` – the new capacity for this stack

Postcondition:
This stack's capacity has been changed to at least `minimumCapacity`. If the capacity was already at or greater than `minimumCapacity`, then the capacity is left unchanged.

Throws: `OutOfMemoryError`
Indicates insufficient memory for: `new Object[minimumCapacity]`.

♦ **getCapacity**

```
public int getCapacity( )
```

Accessor method to determine the current capacity of this stack. The `push` method works efficiently (without needing more memory) until this capacity is reached.

Returns:
the current capacity of this stack

♦ **isEmpty—peek—pop—push—size**

```
public boolean isEmpty( )
public E peek( )
public E pop( )
public void push(E item)
public int size( )
```

These are the standard stack specifications from Figure 6.1 on page 307.

♦ **trimToSize**

```
public void trimToSize( )
```

Reduce the current capacity of this stack to its actual size (i.e., the number of items it contains).

Postcondition:
This stack's capacity has been changed to its current size.

Throws: `OutOfMemoryError`
Indicates insufficient memory for altering the capacity.

(continued)

(FIGURE 6.6 continued)

Implementation

```
// File: ArrayStack.java from the package edu.colorado.collections
// Complete documentation is available on pages 325-326 or from the Stack link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;
import java.util.EmptyStackException;

public class ArrayStack<E> implements Cloneable
{
   // Invariant of the ArrayStack class:
   //   1. The number of items in the stack is in the instance variable manyItems.
   //   2. For an empty stack, we do not care what is stored in any of data; for a
   //      nonempty stack, the items in the stack are stored in a partially filled array called
   //      data, with the bottom of the stack at data[0], the next item at data[1], and so on
   //      to the top of the stack at data[manyItems-1].
   private E[ ] data;
   private int manyItems;

   public ArrayStack( )
   {
      final int INITIAL_CAPACITY = 10;
      manyItems = 0;
      data = (E[]) new Object[INITIAL_CAPACITY];
   }

   public ArrayStack(int initialCapacity)
   {
      if (initialCapacity < 0)
         throw new IllegalArgumentException
         ("initialCapacity too small " + initialCapacity);
      manyItems = 0;
      data = (E[]) new Object[initialCapacity];
   }
```

(continued)

(FIGURE 6.6 continued)

```
public ArrayStack<E> clone( )
{  // Clone an ArrayStack.
   ArrayStack<E> answer;

   try
   {
      answer = (ArrayStack<E>) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {
      // This exception should not occur. But if it does, it would probably indicate a
      // programming error that made super.clone unavailable.
      // The most common error would be forgetting the "Implements Cloneable"
      // clause at the start of this class.
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }

   answer.data = data.clone( );

   return answer;
}

public void ensureCapacity(int minimumCapacity)
{
   E biggerArray[ ];

   if (data.length < minimumCapacity)
   {
      biggerArray =(E[]) new Object[minimumCapacity];
      System.arraycopy(data, 0, biggerArray, 0, manyItems);
      data = biggerArray;
   }
}

public int getCapacity( )
{
   return data.length;
}

public boolean isEmpty( )
{
   return (manyItems == 0);
}
```

(continued)

(FIGURE 6.6 continued)

```
public E peek( )
{
   if (manyItems == 0)
      // EmptyStackException is from java.util and its constructor has no argument.
      throw new EmptyStackException( );
   return data[manyItems-1];
}

public E pop( )
{
   E answer;
   if (manyItems == 0)
      // EmptyStackException is from java.util and its constructor has no argument.
      throw new EmptyStackException( );
   answer = data[--manyItems];
   data[manyItems] = null; // This line is needed only when the data type is Object.
   return answer;
}

public void push(E item)
{
   if (manyItems == data.length)
   {
      // Double the capacity and add 1; this works even if manyItems is 0. However, in
      // case that manyItems*2 + 1 is beyond Integer.MAX_VALUE, there will be an
      // arithmetic overflow and the stack will fail.
      ensureCapacity(manyItems * 2 + 1);
   }
   data[manyItems] = item;
   manyItems++;
}

public int size( )
{
   return manyItems;
}

public void trimToSize( )
{
   E trimmedArray[ ];

   if (data.length != manyItems)
   {
      trimmedArray = (E[]) new Object[manyItems];
      System.arraycopy(data, 0, trimmedArray, 0, manyItems);
      data = trimmedArray;
   }
}
```

Linked List Implementation of a Stack

A linked list is a natural way to implement a stack as a dynamic structure whose size can grow and shrink one item at a time. The head of the linked list serves as the top of the stack. Figure 6.7 contains the specification and implementation for a stack class that is implemented with a linked list. The class is called `ObjectLinkedStack` to distinguish it from our earlier `ObjectStack`. Here is a precise statement of the invariant of this version of the new stack ADT:

Invariant of the Generic LinkedStack Class

1. The items in the stack are stored in a linked list, with the top of the stack stored at the head node, down to the bottom of the stack at the final node.
2. The instance variable `top` is the head reference of the linked list of items.

As usual, all methods (except the constructors) assume that the stack is represented in this way when the method is activated, and all methods ensure that the stack continues to be represented in this way when the method finishes.

Because we are using a linked list, there are no capacity worries. Thus, there are no methods that deal with capacity. A program could build a stack with more than `Integer.MAX_VALUE` items, limited only by its amount of memory. However, beyond `Integer.MAX_VALUE`, the return value of the `size` method will be wrong because of arithmetic overflow.

As a further consequence of using a linked list, it makes sense to utilize the generic `Node<E>` class from Appendix E. Thus, in Figure 6.7 you will find this import statement:

```
import edu.colorado.nodes.Node;
```

By using the `Node<E>` class, many of the stack methods can be implemented with just a line or two of code.

Discussion of the Linked List Implementation of the Stack

The constructor, `size`, and `isEmpty` each require just one line of code. The `size` method actually uses the node's `listLength` method to do its work since we are not maintaining an instance variable that keeps track of the number of nodes. Since the head node of the list is the top of the stack, the implementation of `peek` is easy: `peek` just returns the data from the head node. The operations `push` and `pop` work by adding and removing nodes, always working at the head of the linked list. Adding and removing nodes at the head of the linked list is straightforward using the techniques of Section 4.2. The `clone` method makes use of the node's `listCopy` method to copy the original stack to the clone.

FIGURE 6.7 Specification and Implementation for the Linked List Version of the Generic Stack Class

Generic Class LinkedStack

❖ **public class LinkedStack<E> from the package edu.colorado.collections**

A `LinkedStack` is a stack of references to `E` objects.

Limitations:

Beyond `Int.MAX_VALUE` items, `size` is wrong.

Specification

♦ **Constructor for the LinkedStack<E>**

```
public LinkedStack( )
```

Initialize an empty stack.

Postcondition:

This stack is empty.

♦ **clone**

```
public LinkedStack<E> clone( )
```

Generate a copy of this stack.

Returns:

The return value is a copy of this stack. Subsequent changes to the copy will not affect the original, nor vice versa.

Throws: `OutOfMemoryError`

Indicates insufficient memory for creating the clone.

♦ **isEmpty—peek—pop—push—size**

```
public boolean isEmpty( )
public E peek( )
public E pop( )
public void push(E item)
public int size( )
```

These are the standard stack specifications from Figure 6.1 on page 307.

Implementation

```
// File: LinkedStatck.java from the package edu.colorado.collections
// Complete documentation is available above or from the LinkedStack link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;
import java.util.EmptyStackException;
import edu.colorado.nodes.Node;
```

(continued)

(FIGURE 6.7 continued)

```
public class LinkedStack<E> implements Cloneable
{
   // Invariant of the LinkedStack class:
   //   1. The items in the stack are stored in a linked list, with the top of the stack stored
   //      at the head node, down to the bottom of the stack at the final node.
   //   2. The instance variable top is the head reference of the linked list of items.
   private Node<E> top;

   public LinkedStack( )
   {
      top = null;
   }

   public LinkedStack<E> clone( )
   {  // Clone a LinkedStack.
      LinkedStack<E> answer;

      try
      {
         answer = (LinkedStack<E>) super.clone( );
      }
      catch (CloneNotSupportedException e)
      {
         // This exception should not occur. But if it does, it would probably indicate a
         // programming error that made super.clone unavailable. The most common error
         // is forgetting the "Implements Cloneable" clause at the start of this class.
         throw new RuntimeException
         ("This class does not implement Cloneable.");
      }

      answer.top = Node.listCopy(top); // Generic listCopy method
      return answer;
   }

   public boolean isEmpty( )
   {
      return (top == null);
   }

   public E peek( )
   {
      if (top == null)
         // EmptyStackException is from java.util and its constructor has no argument.
         throw new EmptyStackException( );
      return top.getData( );
   }
```

(continued)

(FIGURE 6.7 continued)

```
   public E pop( )
   {
      E answer;

      if (top == null)
         // EmptyStackException is from java.util and its constructor has no argument.
         throw new EmptyStackException( );

      answer = top.getData( );
      top = top.getLink( );
      return answer;
   }

   public void push(E item)
   {
      top = new Node<E>(item, top);
   }

   public int size( )
   {
      return Node.listLength(top); // Generic listLength method
   }
}
```

Self-Test Exercises for Section 6.3

10. For the array version of the stack, which element of the array contains the top of the stack? In the linked list version, where is the stack's top?
11. For the array version of the stack, write a new member function that returns the maximum number of items that can be added to the stack without stack overflow.
12. Give the full implementation of an accessor method that returns the second item from the top of the stack without actually changing the stack. Write separate solutions for the two different stack versions.
13. For the linked list version of the stack, do we maintain references to both the head and the tail?
14. Is the constructor really needed for the linked list version of the stack? What would happen if we omitted the constructor?
15. Do a time analysis of the `size` method for the linked list version of the stack. If the method is not constant time, then can you think of a different approach that is constant time?
16. What kind of exception is thrown if you try to pop an empty stack?