21.5 Stacks

A stack is a constrained list—*new nodes can be added to and removed from a stack only at the top*. For this reason, a stack is referred to as a **last-in, first-out** (**LIFO**) data structure. The link member in the bottom node is set to `null` to indicate the bottom of the stack. A stack is not required to be implemented as a linked list—it can also be implemented using an array or an `ArrayList`, for example.

Stack Operations

The primary methods for manipulating a stack are `push` and `pop`, which add a new node to the top of the stack and remove a node from the top of the stack, respectively. Method `pop` also returns the data from the popped node.

Stack Applications

Stacks have many interesting applications. For example, when a program calls a method, the called method must know how to return to its caller, so the return address of the calling method is pushed onto the program-execution stack (discussed in Section 6.6). If a series of method calls occurs, the successive return addresses are pushed onto the stack in last-

in, first-out order so that each method can return to its caller. Stacks support recursive method calls in the *same* manner as they do conventional nonrecursive method calls.

The program-execution stack also contains the memory for local variables on each invocation of a method during a program's execution. When the method returns to its caller, the memory for that method's local variables is popped off the stack, and those variables are no longer known to the program. If the local variable is a reference and the object to which it referred has no other variables referring to it, the object can be garbage collected.

Compilers use stacks to evaluate arithmetic expressions and generate machine-language code to process them. The exercises in this chapter explore several applications of stacks, including using them to develop a complete working compiler.

`Stack<E>` Class That Contains a `List<E>`

We take advantage of the close relationship between lists and stacks to implement a stack class by reusing the `List<E>` class (Fig. 21.3). We use *composition* by including a reference to a `List` object as a `private` instance variable. This chapter's list, stack and queue data structures are implemented to store references to objects of any type to encourage further reusability. Class `Stack<E>` (Fig. 21.9) is declared in package `com.deitel.datastructures` (line 3) for

reuse. `Stack<E>`'s source-code file does not import `List<E>`, because the classes are in the same package. Use the following command to compile class `Stack`:

```
javac -d .. -cp .. Stack.java
```

```
// Fig. 21.9: Stack.java
// Stack uses a composed List object.
package com.deitel.datastructures;

import java.util.NoSuchElementException;

public class Stack<E> {
   private List<E> stackList;

   // constructor
   public Stack() {stackList = new List<E>("stac

   // add object to stack
   public void push(E object) {stackList.insertA

   // remove object from stack
   public E pop() throws NoSuchElementException
      return stackList.removeFromFront();
   }

   // determine if stack is empty
   public boolean isEmpty() {return  stackList.i

   // output stack contents
   public void print() {stackList.print();}
}
```

Fig. 21.9

`Stack` uses a composed `List` object.

`Stack<E>` Methods

Class `Stack<E>` has four methods—`push`, `pop`, `isEmpty` and `print`—which are essentially `List<E>`'s `insertAtFront`, `removeFromFront`, `isEmpty` and `print` methods. `List<E>`'s other methods (such as `insertAtBack` and `removeFromBack`) should not be accessible to `Stack<E>` client code—composing a `private List<E>` (line 8) to implement our `Stack<E>`'s capabilities, rather than inheritance, enables us to hide the remaining `List<E>` methods.

We implement each `Stack<E>` method as a call to a `List<E>` method. This is called **delegation**—each `Stack<E>` method delegates its work to the appropriate `List<E>` method. In particular, `Stack<E>` delegates calls to `List<E>` methods `insertAtFront` (line 14), `removeFromFront` (line 18), `isEmpty` (line 22) and `print` (line 25).

Testing Class `Stack<E>`

Class `StackTest` (Fig. 21.10) creates a `Stack<Integer>`

object called `stack` (line 8). The program pushes `int`s onto the stack (lines 11, 13, 15 and 17). *Autoboxing* converts each argument to an `Integer`. Lines 21–33 pop the objects from the stack. If `pop` is invoked on an empty stack, the method throws a `NoSuchElementException`. In this case, the program displays the exception's stack trace, which shows the methods on the program-execution stack at the time the exception occurred. The program outputs the contents of the stack after each `pop` operation. Use the following commands to compile and run class `StackTest`:

```
javac -cp .;.. StackTest.java
 java -cp .;.. StackTest
```

```
 1   // Fig. 21.10: StackTest.java
 2   // Stack manipulation program.
 3   import com.deitel.datastructures.Stack;
 4   import java.util.NoSuchElementException;
 5
 6   public class StackTest {
 7      public static void main(String[] args) {
 8         Stack<Integer> stack = new Stack<>();
 9
10         // use push method
11         stack.push(-1);
12         stack.print();
13         stack.push(0);
14         stack.print();
15         stack.push(1);
16         stack.print();
17         stack.push(5);
18         stack.print();
19
```

```
20          // remove items from stack
21          boolean continueLoop = true;
22
23          while (continueLoop) {
24             try {
25                int removedItem = stack.pop(); // re
26                System.out.printf("%n%d popped%n", r
27                stack.print();
28             }
29             catch (NoSuchElementException noSuchEle
30                continueLoop = false;
31                noSuchElementException.printStackTra
32             }
33          }
34       }
35    }
```

```
The stack is: -1
The stack is: 0 -1
The stack is: 1  0 -1
The stack is: 5  1  0 -1

5 popped
The stack is: 1 0 -1

1 popped
The stack is: 0 -1

0 popped
The stack is: -1

-1 popped
Empty stack
java.util.NoSuchElementException: stack is empty
        at com.deitel.datastructures.List.removeFromF
        at com.deitel.datastructures.Stack.pop(Stack.
           at StackTest.main(StackTest.java:25)
```

Fig. 21.10

Stack manipulation program.