

21.4 Linked Lists

A linked list is a linear collection (i.e., a sequence) of self-referential-class objects, called **nodes**, connected by reference *links*—hence, the term “linked” list. Typically, a program accesses a linked list via a reference to its first node. The program accesses each subsequent node via the link reference stored in the previous node. By convention, the link reference in the last node of the list is set to `null` to indicate “end of list.” Data is stored in and removed from linked lists dynamically—the program creates and deletes nodes as necessary. Stacks and queues are also linear data structures and, as we’ll see, are constrained versions of linked lists. Trees are *nonlinear* data structures.

Lists of data can be stored in conventional Java arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented in the data structure is *unpredictable*. Linked lists are dynamic, so the length of a list can increase or decrease as necessary, whereas the size of a conventional Java array cannot be altered—it’s fixed when the program creates the array. [Of course, `ArrayLists` *can* grow and shrink.] Conventional arrays can become full. Linked lists become full only when the system has *insufficient memory* to satisfy dynamic storage allocation requests. Package `java.util` contains class `LinkedList` (discussed in [Chapter 16](#)) for

implementing and manipulating linked lists that grow and shrink during program execution.



Performance Tip 21.1

After locating the insertion point, insertion into a linked list is fast—only two references have to be modified. All existing node objects remain at their current locations in memory.

Linked lists can be maintained in sorted order simply by inserting each new element at the proper point in the list. (It does, of course, take time to *locate* the proper insertion point.) Existing list elements do not need to be moved.



Performance Tip 21.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

21.4.1 Singly Linked Lists

Linked list nodes normally are *not stored contiguously* in memory. Rather, they're logically contiguous. [Figure 21.2](#) illustrates a linked list with several nodes. This diagram presents a **singly linked list**—each node contains one

reference to the next node in the list. Often, linked lists are implemented as *doubly linked lists*—each node contains a reference to the next node in the list *and* a reference to the preceding one.



Performance Tip 21.3

The elements of an array are contiguous in memory. This allows immediate access to any array element, because its address can be calculated directly as its offset from the beginning of the array. Linked lists do not afford such immediate access—an element can be accessed only by traversing the list from the front (or the back in a doubly linked list).

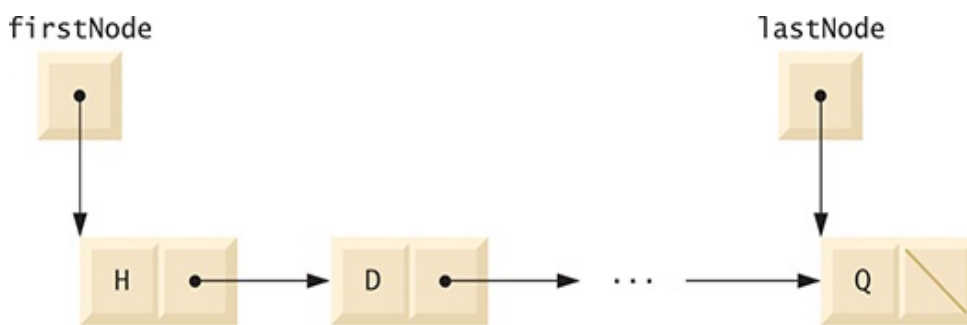


Fig. 21.2

Linked-list graphical representation.

Description

21.4.2 Implementing a Generic List Class

The program of [Figs. 21.3–21.4](#) uses an object of our generic `List` class to manipulate a list of miscellaneous objects. The program consists of three classes—`ListNode` ([Fig. 21.3](#), lines 8–28), `List` ([Fig. 21.3](#), lines 31–132) and `ListTest` ([Fig. 21.4](#)). Encapsulated in each `List` object is a linked list of `ListNode` objects.

[*Note:* The `List` and `ListNode` classes are placed in the package `com.deitel.datastructures`, so that they can be reused throughout this chapter. In [Section 21.4.10](#), we discuss the `package` statement (line 3 of [Fig. 21.3](#)) and show how to compile and run programs that use classes in your own packages.]

```
1 // Fig. 21.3: List.java
2 // ListNode and List class declarations.
3 package com.deitel.datastructures;
4
5 import java.util.NoSuchElementException;
6
7 // class to represent one node in a list
8 class ListNode<E> {
9     // package access members; List can access them
10     E data; // data for this node
11     ListNode<E> nextNode; // reference to the next node
12
13     // constructor creates a ListNode that refers to itself
14     ListNode(E object) {this(object, null);}
15
16     // constructor creates ListNode that refers to null
```

```

17      // object and to the next ListNode
18      ListNode(E object, ListNode<E> node) {
19          data = object;
20          nextNode = node;
21      }
22
23      // return reference to data in node
24      E getData() {return data;}
25
26      // return reference to next node in list
27      ListNode<E> getNext() {return nextNode;}
28  }
29
30  // class List definition
31  public class List<E> {
32      private ListNode<E> firstNode;
33      private ListNode<E> lastNode;
34      private String name; // string like "list" us
35
36      // constructor creates empty List with "list"
37      public List() {this("list");}
38
39      // constructor creates an empty List with a n
40      public List(String listName) {
41          name = listName;
42          firstNode = lastNode = null;
43      }
44
45      // insert item at front of List
46      public void insertAtFront(E insertItem) {
47          if (isEmpty()) { // firstNode and lastNode
48              firstNode = lastNode = new ListNode<E>(
49              }
50          else { // firstNode refers to new node
51              firstNode = new ListNode<E>(insertItem,
52              }
53          }
54
55      // insert item at end of List
56      public void insertAtBack(E insertItem) {

```

```

57         if (isEmpty()) { // firstNode and lastNode
58             firstNode = lastNode = new ListNode<E>(
59                 59             }
60         else { // lastNode's nextNode refers to ne
61             lastNode = lastNode.nextNode = new List
62                 62             }
63                 63         }
64
65         // remove first node from List
66     public E removeFromFront() throws NoSuchEleme
67         if (isEmpty()) { // throw exception if Lis
68             throw new NoSuchElementException(name +
69                 69             }
70
71         E removedItem = firstNode.data; // retriee
72
73         // update references firstNode and lastNo
74         if (firstNode == lastNode) {
75             firstNode = lastNode = null;
76         }
77         else {
78             firstNode = firstNode.nextNode;
79         }
80
81         return removedItem; // return removed nod
82     }
83
84     // remove last node from List
85     public E removeFromBack() throws NoSuchEleme
86         if (isEmpty()) { // throw exception if Li
87             throw new NoSuchElementException(name
88                 88             }
89
90         E removedItem = lastNode.data; // retriev
91
92         // update references firstNode and lastNo
93         if (firstNode == lastNode) {
94             firstNode = lastNode = null;
95         }
96         else { // locate new last node

```

```

    97         ListNode<E> current = firstNode;
                98
    99         // loop while current node does not re
   100         while (current.nextNode != lastNode) {
   101             current = current.nextNode;
                102         }
                103
   104         lastNode = current; // current is new
   105         current.nextNode = null;
                106     }
                107
   108     return removedItem; // return removed nod
                109     }
                110
   111     // determine whether list is empty; returns
   112     public boolean isEmpty() {return firstNode =
                113
   114         // output list contents
   115         public void print() {
   116             if (isEmpty()) {
   117                 System.out.printf("Empty %s\n", name);
   118                 return;
   119             }
                120
   121         System.out.printf("The %s is: ", name);
   122         ListNode<E> current = firstNode;
                123
   124         // while not at end of list, output curre
   125         while (current != null) {
   126             System.out.printf("%s ", current.data)
   127             current = current.nextNode;
   128         }
                129
   130         System.out.println();
   131     }
   132 }

```

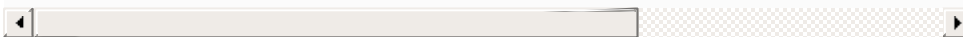


Fig. 21.3

`ListNode` and `List` class declarations.

21.4.3 Generic Classes `ListNode` and `List`

Generic class `ListNode` (Fig. 21.3, lines 8–28) declares package-access fields `data` and `nextNode`. The `data` field is a reference of type `E`, so its type will be determined when the client code creates the corresponding `List` object.

Variable `nextNode` stores a reference to the next `ListNode` object in the linked list (or `null` if the node is the last one in the list).

Lines 32–33 of class `List` (lines 31–132) declare references to the first and last `ListNode`s in a `List` (`firstNode` and `lastNode`, respectively). The constructors (lines 37 and 40–43) initialize both references to `null`. The most important methods of class `List` are `insertAtFront` (lines 46–53), `insertAtBack` (lines 56–63), `removeFromFront` (lines 66–82) and `removeFromBack` (lines 85–109). Method `isEmpty` (line 112) is a *predicate method* that determines whether the list is empty (i.e., the reference to the first node of the list is `null`). Predicate methods typically test a condition and do not modify the object on which they're called. If the list is empty, method `isEmpty` returns `true`; otherwise, it returns `false`. Method `print` (lines 115–131) displays the

list's contents. We discuss class `List`'s methods in more detail after we discuss class `ListTest`.

21.4.4 Class `ListTest`

Method `main` of class `ListTest` (Fig. 21.4) creates a `List<Integer>` object (line 8), then inserts objects at the beginning of the list using method `insertAtFront`, inserts objects at the end of the list using method `insertAtBack`, deletes objects from the front of the list using method `removeFromFront` and deletes objects from the end of the list using method `removeFromBack`. After each insert and remove operation, `ListTest` calls `List` method `print` to display the current list contents. If an attempt is made to remove an item from an empty list, a `NoSuchElementException` is thrown, so the method calls to `removeFromFront` and `removeFromBack` are placed in a `try` block that's followed by an appropriate exception handler. Notice in lines 11, 13, 15 and 17 that the application passes literal primitive `int` values to methods `insertAtFront` and `insertAtBack`. Each of these methods was declared with a parameter of the generic type `E` (Fig. 21.3, lines 46 and 56). Since this example manipulates a `List<Integer>`, the type `E` represents the type-wrapper class `Integer`. In this case, the JVM *autoboxes* each literal value in an `Integer` object, and that object is actually inserted into the list.

```

1  // Fig. 21.4: ListTest.java
2  // ListTest class to demonstrate List capabilities
3  import com.deitel.datastructures.List;
4  import java.util.NoSuchElementException;
5
6  public class ListTest {
7      public static void main(String[] args) {
8          List<Integer> list = new List<>();
9
10         // insert integers in list
11         list.insertAtFront(-1);
12         list.print();
13         list.insertAtFront(0);
14         list.print();
15         list.insertAtBack(1);
16         list.print();
17         list.insertAtBack(5);
18         list.print();
19
20         // remove objects from list; print after each
21         try {
22             int removedItem = list.removeFromFront();
23             System.out.printf("%n%d removed%n", removedItem);
24             list.print();
25
26             removedItem = list.removeFromFront();
27             System.out.printf("%n%d removed%n", removedItem);
28             list.print();
29
30             removedItem = list.removeFromBack();
31             System.out.printf("%n%d removed%n", removedItem);
32             list.print();
33
34             removedItem = list.removeFromBack();
35             System.out.printf("%n%d removed%n", removedItem);
36             list.print();
37         }
38     catch (NoSuchElementException noSuchElementException) {
39         noSuchElementException.printStackTrace();
40     }

```

```
41     }  
42 }
```



```
The list is: -1  
The list is: 0 -1  
The list is: 0 -1 1  
The list is: 0 -1 1 5  
  
0 removed  
The list is: -1 1 5  
  
-1 removed  
The list is: 1 5  
5 removed  
The list is: 1  
  
1 removed  
Empty list
```

Fig. 21.4

ListTest class to demonstrate List capabilities.

21.4.5 List Method insertAtFront

Now we discuss each method of class `List` (Fig. 21.3) in detail and provide diagrams showing the reference

manipulations performed by methods `insertAtFront`, `insertAt-Back`, `removeFromFront` and `removeFromBack`. Method `insertAtFront` (lines 46–53 of [Fig. 21.3](#)) places a new node at the front of the list. The steps are:

1. Call `isEmpty` to determine whether the list is empty (line 47).
2. If the list is empty, assign to `firstNode` and `lastNode` the new `ListNode` that was initialized with `insertItem` (line 48). (Recall that assignment operators evaluate right to left.) The `ListNode` constructor at line 14 calls the `ListNode` constructor at lines 18–21 to set instance variable `data` to refer to the `insertItem` and to set reference `nextNode` to `null`, because this is the first and last node in the list.
3. If the list is not empty, the new node is “linked” into the list by setting `firstNode` to a new `ListNode` object and initializing that object with `insertItem` and `firstNode` (line 51). When the `ListNode` constructor (lines 18–21) executes, it sets instance variable `data` to refer to the `insertItem` passed as an argument and performs the insertion by setting the `nextNode` reference of the new node to the `ListNode` passed as an argument, which previously was the first node.

In [Fig. 21.5](#), part (a) shows a list and a new node during the `insertAtFront` operation and before the program links the new node into the list. The dotted arrows in part (b) illustrate *Step 3* of the `insertAtFront` operation that enables the node containing 12 to become the new first node in the list.

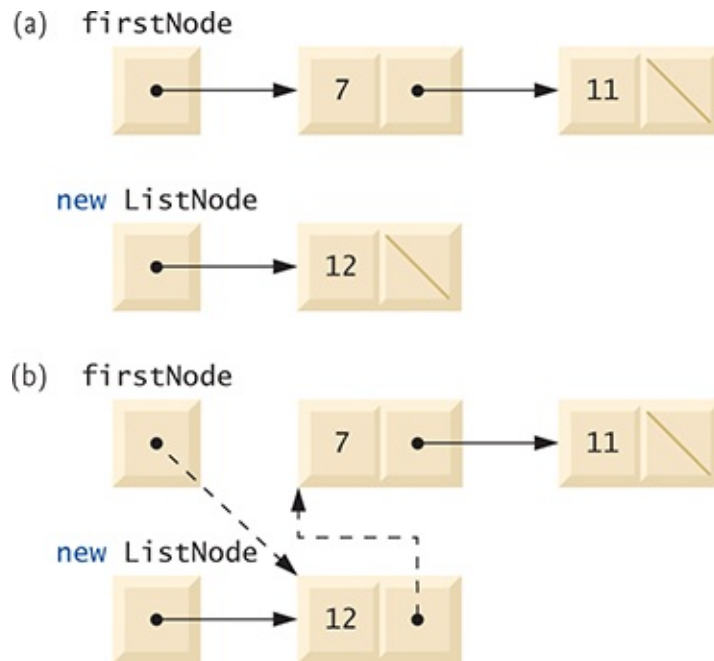


Fig. 21.5

Graphical representation of operation `insertAtFront`.

Description

21.4.6 List Method `insertAtBack`

Method `insertAtBack` (lines 56–63 of [Fig. 21.3](#)) places a new node at the back of the list. The steps are:

1. Call `isEmpty` to determine whether the list is empty (line 57).
2. If the list is empty, assign to `firstNode` and `lastNode` the new `ListNode` that was initialized with `insertItem` (line 58). The

`ListNode` constructor at line 14 calls the constructor at lines 18–21 to set instance variable `data` to refer to the `insertItem` passed as an argument and to set reference `nextNode` to `null`.

3. If the list is not empty, line 61 links the new node into the list by assigning to `lastNode` and `lastNode.nextNode` the reference to the new `ListNode` that was initialized with `insertItem`. `ListNode`'s constructor (line 14) sets instance variable `data` to refer to the `insertItem` passed as an argument and sets reference `nextNode` to `null`, because this is the last node in the list.

In Fig. 21.6, part (a) shows a list and a new node during the `insertAtBack` operation and before linking the new node into the list. The dotted arrows in part (b) illustrate *Step 3* of method `insertAtBack`, which adds the new node to the end of a list that's not empty.

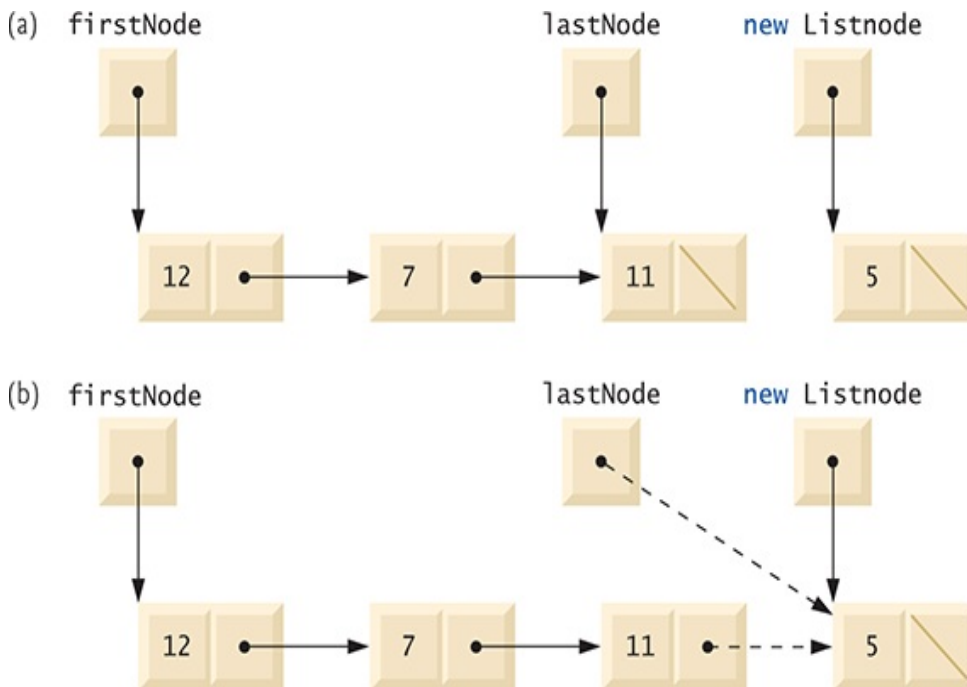


Fig. 21.6

Graphical representation of operation `insertAtBack`.

Description

21.4.7 List Method `removeFromFront`

Method `removeFromFront` (lines 66–82 of [Fig. 21.3](#)) removes the first node of the list and returns a reference to the removed data. If the list is empty when the program calls this method, the method throws an `NoSuchElementException` (lines 67–69). Otherwise, the method returns a reference to the removed data. The steps are:

1. Assign `firstNode.data` (the data being removed) to `removedItem` (line 71).
2. If `firstNode` and `lastNode` refer to the same object (line 74), the list has only one element at this time. So, the method sets `firstNode` and `lastNode` to `null` (line 75) to remove the node from the list (leaving the list empty).
3. If the list has more than one node, then the method leaves reference `lastNode` as is and assigns the value of `firstNode.nextNode` to `firstNode` (line 78). Thus, `firstNode` references the node that was previously the second node in the list.
4. Return the `removedItem` reference (line 81).

In [Fig. 21.7](#), part (a) illustrates the list before the removal operation. The dashed lines and arrows in part (b) show the reference manipulations.

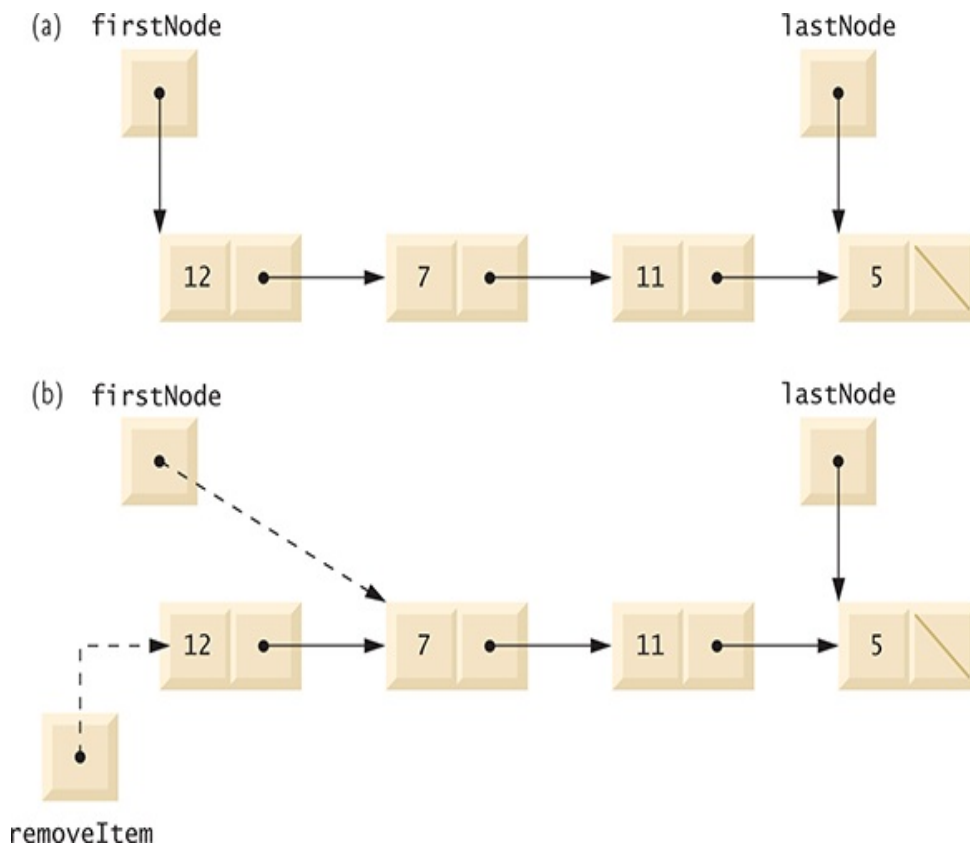


Fig. 21.7

Graphical representation of operation `removeFromFront`.

Description

21.4.8 List Method `removeFromBack`

Method `removeFromBack` (lines 85–109 of [Fig. 21.3](#)) removes the last node of a list and returns a reference to the

removed data. The method throws a `NoSuchElementException` (lines 86–88) if the list is empty when the program calls this method. The steps are:

1. Assign `lastNode.data` (the data being removed) to `removedItem` (line 90).
2. If the `firstNode` and `lastNode` refer to the same object (line 93), the list has only one element at this time. So, line 94 sets `firstNode` and `lastNode` to `null` to remove that node from the list (leaving the list empty).
3. If the list has more than one node, create the `ListNode` reference `current` and assign it `firstNode` (line 97).
4. Now “walk the list” with `current` until it references the node before the last node. The `while` loop (lines 100–102) assigns `current.nextNode` to `current` as long as `current.nextNode` (the next node in the list) is not `lastNode`.
5. After locating the second-to-last node, assign `current` to `lastNode` (line 104) to update which node is last in the list.
6. Set the `current.nextNode` to `null` (line 105) to remove the last node from the list and terminate the list at the current node.
7. Return the `removedItem` reference (line 108).

In Fig. 21.8, part (a) illustrates the list before the removal operation. The dashed lines and arrows in part (b) show the reference manipulations. [We limited the `List` insertion and removal operations to the front and the back of the list. In Exercise 21.26, you’ll enhance the `List` class to enable insertions and deletions *anywhere* in the `List`.]

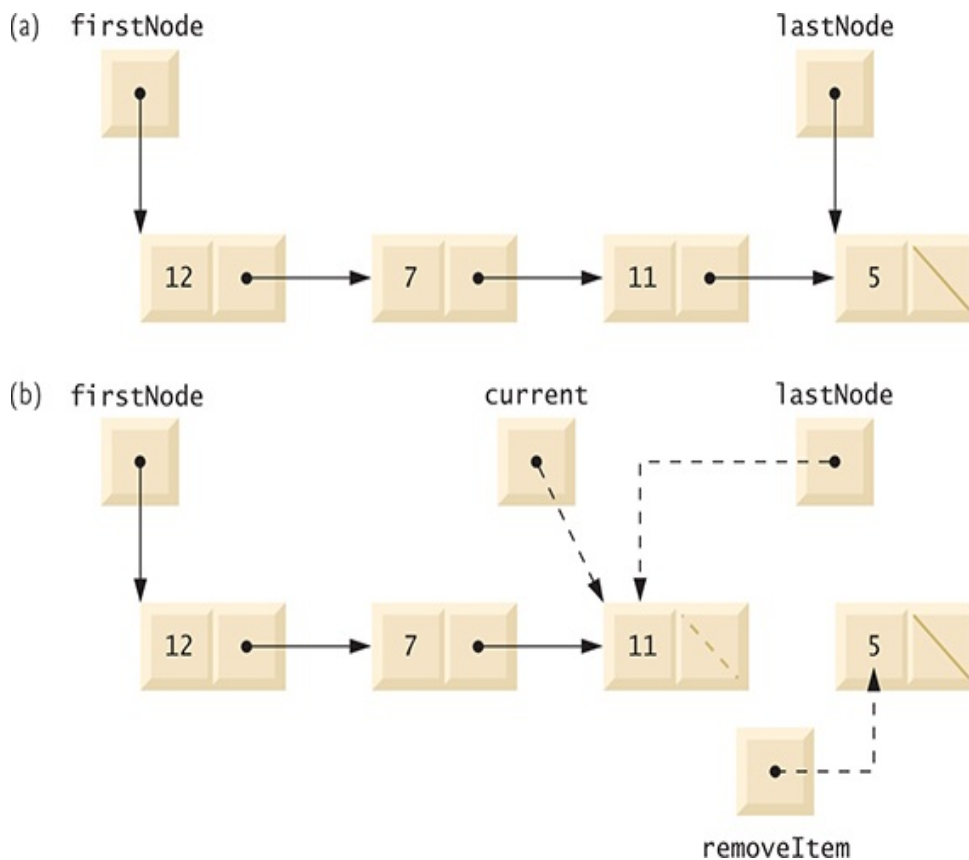


Fig. 21.8

Graphical representation of operation `removeFromBack`.

Description

21.4.9 List Method `print`

Method `print` (lines 115–131 of Fig. 21.3) first determines whether the list is empty (lines 116–119) and, if so, displays a message and returns control to the calling method. Otherwise, `print` outputs the list's data. Line 122 creates `ListNode`

`current` and initializes it with `firstNode`. While `current` is not `null`, there are more items in the list. Therefore, line 126 outputs a string representation of `current.data`. Line 127 moves to the next node in the list by assigning the value of reference `current.nextNode` to `current`. This printing algorithm is identical for linked lists, stacks and queues.

21.4.10 Creating Your Own Packages

As you know, the Java API types (classes, interfaces and `enums`) are organized in *packages* that group related types. Packages facilitate software reuse by enabling programs to `import` existing classes, rather than *copying* them into the folders of each program that uses them. Programmers use packages to organize program components, especially in large programs. For example, you might have one package containing the types that make up your program's graphical user interface, another for the types that manage your application's data and another for the types that communicate with servers over a network. In addition, packages help you specify unique names for every type you declare, which (as we'll discuss) helps prevent class-name conflicts. This section introduces how to create and use your own packages. Much of what we discuss here is handled for you by IDEs such as NetBeans, Eclipse and IntelliJ IDEA. We focus on creating and using packages with the JDK's command-line tools.

Steps for Declaring a Reusable Class

Before a class can be imported into multiple programs, it must be placed in a package to make it reusable. The steps for creating a reusable class are:

1. Declare one or more **public** types (classes, interfaces and **enums**). Only **public** types can be reused outside the package in which they're declared.
2. Choose a unique package name and add a **package declaration** to the source-code file for each reusable type that should be part of the package.
3. Compile the types so that they're placed in the appropriate package directory.
4. Import the reusable types into a program and use them.

We'll now discuss each of these steps in more detail.

Step 1: Creating **public** Types for Reuse

For *Step 1*, you declare the types that will be placed in the package, including both the reusable types and any supporting types. In [Fig. 21.3](#), class `List` is **public**, so it's reusable outside its package. Class `ListNode`, however, is not **public**, so it can be used *only* by class `List` and any other types declared in the *same* package. If a source-code file contains more than one type, all types in the file are placed in

the same package when the file is compiled.

Step 2: Adding the package Statements

For *Step 2*, you provide a **package** declaration containing the package's name. All source-code files containing types that should be part of the same package must contain the *same* package declaration. [Figure 21.3](#) contains:

```
package com.deitel.datastructures;
```



indicating that all the types declared in this file—`ListNode` and `List` in [Fig. 21.3](#)—are part of the `com.deitel.datastructures` package.

Each Java source-code file may contain only *one* package declaration, and it must *precede* all other declarations and statements. If no **package** statement is provided in a Java source-code file, the types declared in that file are placed in the so-called *default package* and are accessible only to other classes in the default package that are located in the *same* directory. All prior programs in this book have used this default package.

Package Naming

Conventions

A package name's parts are separated by dots (.), and there typically are two or more parts. To ensure *unique* package names, you typically begin the name with your institution's or company's Internet domain name in reverse order—e.g., our domain name is `deitel.com`, so we begin our package names with `com.deitel`. For the domain name `yourcollege.edu`, you'd begin the package name with `edu.yourcollege`.

After the reversed domain name, you can specify additional parts in a package name. If you're part of a university with many schools or company with many divisions, you might use the school or division name as the next part of the package name. Similarly, if the types are for a specific project, you might include the project name as part of the package name. We chose `datastructures` as the next part in our package name to indicate that classes `ListNode` and `List` are from this data-structures chapter.

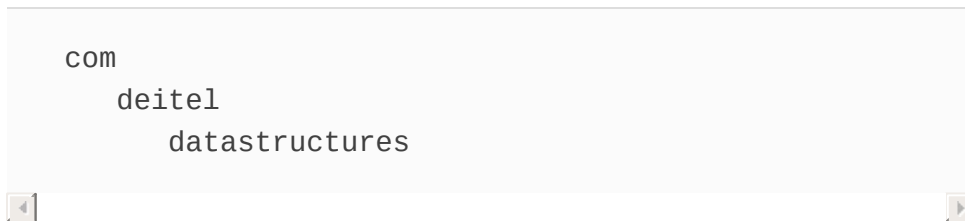
Fully Qualified Names

The package name is part of the **fully qualified type name**, so the name of class `List` is actually `com.deitel.datastructures.List`. You can use this fully qualified name in your programs, or you can `import` the class and use its **simple name** (the class name by itself—`List`) in the program. If another package also contains a

List class, the fully qualified class names can be used to distinguish between the classes in the program and prevent a **name conflict** (also called a **name collision**).

Step 3: Compiling Packaged Types

Step 3 is to compile the class so that it's stored in the appropriate package. When a Java file containing a **package** declaration is compiled, the resulting class file is placed in a directory specified by the declaration. Classes in the package `com.deitel.datastructures` are placed in the directory



```
com
  deitel
    datastructures
```

The names in the **package** declaration specify the exact location of the package's classes.

The `javac` command-line option `-d` causes the compiler to create the directories based on the **package** declaration. The option also specifies where the top-level directory in the package name should be placed on your system—you may specify a relative or complete path to this location. For example, the command

```
javac -d . List.java
```

specifies that the first directory in our package name (`com`) should be placed in the current directory. The period (`.`) after `-d` in the preceding command represents the *current directory* on the Windows, UNIX, Linux and macOS operating systems (and several others as well). Similarly, the command

```
javac -d .. List.java
```

specifies that the first directory in our package name (`com`) should be placed in the *parent* directory—we did this for all the reusable classes in this chapter. Once you compile with the `-d` option, the package’s `datastructures` directory contains the files `ListNode.class` and `List.class`.

Step 4: Importing Types from Your Package

Once types are compiled into a package, they can be imported (*Step 4*). Class `ListTest` ([Fig. 21.4](#)) is in the *default package* because its `.java` file does not contain a `package` declaration. Class `ListTest` is in a *different* package from `List`, so you must either `import` class `List` so that class `ListTest` can use it (line 3 of [Fig. 21.4](#)) or you must *fully qualify* the name `List` everywhere it’s used throughout class

ListTest. For example, line 8 of [Fig. 21.4](#) could have been written as:

```
com.deitel.datastructures.List<Integer> list =  
    new com.deitel.datastructures.List<>();
```

Single-Type-Import vs. Type-Import-On-Demand Declarations

Lines 3–4 of [Fig. 21.4](#) are **single-type-import declarations**—they each specify *one* class to import. When a source-code file uses *multiple* classes from a package, you can import those classes with a **type-import-on-demand declaration** of the form

```
import packagename.*;
```

which uses an asterisk (*) at its end to inform the compiler that *all* **public** classes from the *packagename* package can be used without fully qualifying their names in the file containing the **import**. Only those classes that are *used* are loaded at execution time. The preceding **import** allows you to use the simple name of any type from the *packagename* package. Throughout this book, we provide single-type-import declarations as a form of documentation to show you

specifically which types are used in each program.



Common Programming Error 21.1

Using the `import` declaration `import java.`; causes a compilation error. You must specify the full package name from which you want to import classes.*



Error-Prevention Tip 21.1

Using single-type-import declarations helps avoid naming conflicts by importing only the types you actually use in your code.

Specifying the Classpath When Compiling a Program

When compiling `ListTest`, `javac` must locate the `.class` files for class `List` to ensure that class `ListTest` uses it correctly. The compiler uses a special object called a **class loader** to locate the classes it needs. The class loader

begins by searching the standard Java classes that are bundled with the JDK. Then it searches for **optional packages**. Java provides an **extension mechanism** that enables new (optional) packages to be added to Java for development and execution purposes. If the class is not found in the standard Java classes or in the extension classes, the class loader searches the **classpath**—a list of directories or **archive files** containing reusable types. Each directory or archive file is separated from the next by a **directory separator**—a semicolon (;) on Windows or a colon (:) on UNIX/Linux/ macOS. Archive files are individual files that contain directories of other files, typically in a compressed format. For example, the standard classes used by your programs are contained in the archive file `rt.jar`, which is installed with the JDK. Archive files normally end with the `.jar` or `.zip` filename extensions.

By default, the classpath consists only of the current directory. However, the classpath can be modified by

1. providing the `-classpath` *listOfDirectories* option to the `javac` compiler or
2. setting the `CLASSPATH` **environment variable** (a special variable that you define and the operating system maintains so that programs can search for classes in the specified locations).

If you compile `ListTest.java` without specifying the `-classpath` option, as in

```
javac ListTest.java
```

the class loader assumes that the additional package(s) used by the `ListTest` program are in the *current directory*. As we mentioned, we placed our package in the *parent* directory so that it could be used by other programs in this chapter. To compile `ListTest.java`, use the command

```
javac -classpath .;.. ListTest.java
```

on Windows or the command

```
javac -classpath .:.. ListTest.java
```

on UNIX/Linux/macOS. The `.` in the classpath enables the class loader to locate `ListTest` in the current directory. The `..` enables the class loader to locate the contents of package `com.deitel.datastructures` in the parent directory. The `-classpath` option may also be abbreviated as `-cp`.



Common Programming Error 21.2

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (`.`) in the classpath to specify

the current directory.



Software Engineering Observation 21.2

In general, it's a better practice to use the `-classpath` option of the compiler, rather than the `CLASSPATH` environment variable, to specify the classpath for a program. This enables each program to have its own classpath.



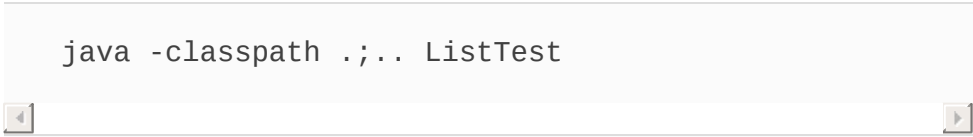
Error-Prevention Tip 21.2

Specifying the classpath with the `CLASSPATH` environment variable can cause subtle and difficult-to-locate errors in programs that use different versions of the same package.

Specifying the Classpath When Executing a Program

When you execute a program, the JVM must be able to locate the `.class` files for the program's classes. Like the compiler, the `java` command uses a *class loader* that searches the

standard classes and extension classes first, then searches the classpath (the current directory by default). The classpath can be specified explicitly by using the same techniques discussed for the compiler. As with the compiler, it's better to specify an individual program's classpath via command-line JVM options. You can specify the classpath in the `java` command via the `-classpath` or `-cp` command-line options, followed by a list of directories or archive files. Again, if classes must be loaded from the current directory, be sure to include a dot (.) in the classpath to specify the current directory. To execute the `ListTest` program, use the following command:



```
java -classpath .;.. ListTest
```

(Again, use `:` rather than `;` on UNIX/Linux/macOS.) *You'll need to use similar `javac` and `java` commands for each of this chapter's remaining examples.* For more about the classpath, visit docs.oracle.com/javase/8/docs/technotes/tools/index.