17.4 Clases autorreferenciadas

Una **clase autorreferenciada** contiene una variable de instancia que hace referencia a otro objeto del mismo tipo de clase. Por ejemplo, la declaración:

```
class Nodo
{
    private int datos;
    private Nodo siguienteNodo; // referencia al siguiente nodo enlazado

    public Nodo( int datos ) { /* cuerpo del constructor */ }
    public void establecerDatos( int datos ) { /* cuerpo del método */ }
    public int obtenerDatos() { /* cuerpo del método */ }
    public void establecerSiguiente( Nodo siguiente ) { /* cuerpo del método */ }
    public Nodo obtenerSiguiente() { /* cuerpo del método */ }
} // fin de la clase Nodo
```

declara la clase `Nodo`, la cual tiene dos variables de instancia `private`: la variable entera `datos` y la referencia `Nodo` llamada `siguienteNodo`. El campo `siguienteNodo` hace referencia a un objeto de la clase `Nodo`, un objeto de la misma clase que se está declarando aquí; es por ello que se utiliza el término "clase autorreferenciada". El campo `siguienteNodo` es un **enlace**; "vincula" a un objeto de tipo `Nodo` con otro objeto del mismo tipo. El tipo `Nodo` también tiene cinco métodos: un constructor que recibe un entero para inicializar a `datos`, un método `establecerDatos` para establecer el valor de `datos`, un método `obtenerDatos` para devolver el valor de `datos`, un método `establecerSiguiente` para establecer el valor de `siguienteNodo` y un método `obtenerSiguiente` para devolver una referencia al siguiente nodo.

Los programas pueden enlazar objetos autorreferenciados entre sí para formar estructuras de datos útiles como listas, colas, pilas y árboles. En la figura 17.1 se muestran dos objetos autorreferenciados, enlazados entre sí para formar una lista. Una barra diagonal inversa (que representa una referencia `null`) se coloca en el miembro de enlace del segundo objeto autorreferenciado para indicar que el enlace no hace referencia a otro objeto. La barra diagonal inversa es ilustrativa; no corresponde al carácter de barra diagonal inversa en Java. Utilizamos `null` para indicar el final de una estructura de datos.

Figura 17.1 | Objetos de una clase autorreferenciada enlazados entre sí.

17.5 Asignación dinámica de memoria

Para crear y mantener estructuras dinámicas de datos se requiere la **asignación dinámica de memoria**; la habilidad para que un programa obtenga más espacio de memoria en tiempo de ejecución, para almacenar nuevos nodos y para liberar el espacio que ya no se necesita. Recuerde que los programas de Java no liberan explícitamente la memoria asignada en forma dinámica. En vez de ello, Java realiza la recolección automática de basura en los objetos que ya no son referenciados en un programa.

El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de espacio en disco disponible en un sistema con memoria virtual. A menudo, los límites son mucho más pequeños ya que la memoria disponible de la computadora debe compartirse entre muchas aplicaciones.

La expresión de declaración y creación de instancia de clase:

```
Nodo nodoParaAgregar = new Nodo( 10 ); // 10 son los datos de nodoParaAgregar
```

asigna la memoria para almacenar un objeto `Nodo` y devuelve una referencia al objeto, que se asigna a `nodoParaAgregar`. Si no hay disponible suficiente memoria, la expresión lanza una excepción `OutOfMemoryError`.

Las siguientes secciones hablan sobre listas, pilas, colas y árboles que utilizan asignación dinámica de memoria y clases autorreferenciadas para crear estructuras de datos dinámicas.

17.6 Listas enlazadas

Una lista enlazada es una colección lineal (es decir, una secuencia) de objetos de una clase autorreferenciada, conocidos como **nodos**, que están conectados por enlaces de referencia; es por ello que se utiliza el término lista "enlazada". Por lo general, un programa accede a una lista enlazada mediante una referencia al primer nodo en la lista. El programa accede a cada nodo subsiguiente a través de la referencia de enlace almacenada en el nodo anterior. Por convención, la referencia de enlace en el último nodo de una lista se establece en `null`. Los datos se almacenan en forma dinámica en una lista enlazada; el programa crea cada nodo según sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo referencias a objetos de otras clases. Las pilas y las colas son también estructuras de datos lineales y, como veremos, son versiones restringidas de las listas enlazadas. Los árboles son estructuras de datos no lineales.

Pueden almacenarse listas de datos en los arreglos, pero las listas enlazadas ofrecen varias ventajas. Una lista enlazada es apropiada cuando el número de elementos de datos que se van a representar en la estructura de datos es impredecible. Las listas enlazadas son dinámicas, por lo que la longitud de una lista puede incrementarse o reducirse, según sea necesario. Sin embargo, el tamaño de un arreglo "convencional" en Java no puede alterarse; el tamaño del arreglo se fija en el momento en que el programa lo crea. Los arreglos "convencionales" pueden llenarse. Las listas enlazadas se llenan sólo cuando el sistema no tiene suficiente memoria para satisfacer las peticiones de asignación dinámica de almacenamiento. El paquete `java.util` contiene la clase `LinkedList` para implementar y manipular listas enlazadas que crezcan y se reduzcan durante la ejecución del programa. Hablaremos sobre la clase `LinkedList` en el capítulo 19, Colecciones.

Tip de rendimiento 17.1

Un arreglo puede declararse de manera que contenga más elementos que el número de elementos esperados, pero esto desperdicia memoria. Las listas enlazadas proporcionan una mejor utilización de memoria en estas situaciones. Las listas enlazadas permiten al programa adaptarse a las necesidades de almacenamiento en tiempo de ejecución.

Tip de rendimiento 17.2

La inserción en una lista enlazada es rápida; sólo hay que modificar dos referencias (después de localizar el punto de inserción). Todos los objetos nodo existentes permanecen en sus posiciones actuales en memoria.

Las listas enlazadas pueden mantenerse en orden con sólo insertar cada nuevo elemento en el punto apropiado de la lista. (Claro que lleva tiempo localizar el punto de inserción apropiado). Los elementos existentes en la lista no necesitan moverse.

Tip de rendimiento 17.3

La inserción y la eliminación en un arreglo ordenado puede llevar mucho tiempo; todos los elementos que van después del elemento insertado o eliminado deben desplazarse apropiadamente.

Por lo general, los nodos de las listas enlazadas no se almacenan contiguamente en memoria, sino que son adyacentes en forma lógica. La figura 17.2 muestra una lista enlazada con varios nodos. Este diagrama presenta una **lista de enlace simple** (cada nodo contiene una referencia al siguiente nodo en la lista). A menudo, las listas

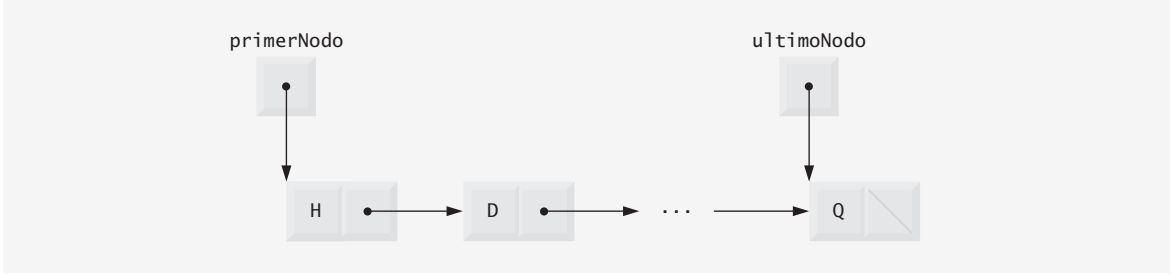

Figura 17.2 | Representación gráfica de una lista enlazada.

enlazadas se implementan como listas de enlace doble (cada nodo contiene una referencia al siguiente nodo en la lista y una referencia al nodo anterior en la lista). La clase `LinkedList` de Java es una implementación de lista de enlace doble.

Tip de rendimiento 17.4

Por lo general, los elementos de un arreglo están contiguos en memoria. Esto permite un acceso inmediato a cualquier elemento del arreglo, ya que la dirección de cualquier elemento puede calcularse directamente como su desplazamiento a partir del inicio del arreglo. Las listas enlazadas no permiten dicho acceso inmediato a sus elementos; para acceder a ellos se tiene que recorrer la lista desde su parte inicial (o desde la parte final en una lista de enlace doble).

El programa de la figuras 17.3 a 17.5 utiliza un objeto de nuestra clase `Lista` para manipular una lista de objetos misceláneos. El programa consta de cuatro clases: `NodoLista` (figura 17.3, líneas 6 a 37), `Lista` (figura 17.3, líneas 40 a 147), `ExcepcionListaVacia` (figura 17.4) y `PruebaLista` (figura 17.5). Las clases `Lista`, `NodoLista` y `ExcepcionListaVacia` están colocadas en el paquete `com.deitel.jhtp7.cap17`, para que puedan reutilizarse a lo largo de este capítulo. Hay una lista enlazada de objetos `NodoLista` encapsulada en cada objeto `Lista`. [*Nota:* muchas de las clases en este capítulo se declaran en el paquete `com.deitel.jhtp7.cap17`. Cada una de estas clases debe compilarse con la opción de línea de comandos `-d` para `javac` Cuando compile las clases que no están en este paquete y cuando ejecute los programas, asegúrese de utilizar la opción `-classpath` para `javac` y `java`, respectivamente].

```
1  // Fig. 17.3: Lista.java
2  // Definiciones de las clases NodoLista y Lista.
3  package com.deitel.jhtp7.cap17;
4
5  // clase para representar un nodo en una lista
6  class NodoLista
7  {
8     // miembros de acceso del paquete; Lista puede acceder a ellos directamente
9     Object datos; // los datos para este nodo
10    NodoLista siguienteNodo; // referencia al siguiente nodo en la lista
11
12    // el constructor crea un objeto NodoLista que hace referencia al objeto
13    NodoLista( Object objeto )
14    {
15       this( objeto, null );
16    } // fin del constructor de NodoLista con un argumento
17
18    // el constructor crea un objeto NodoLista que hace referencia a
19    // un objeto Object y al siguiente objeto NodoLista
20    NodoLista( Object objeto, NodoLista nodo )
21    {
22       datos = objeto;
23       siguienteNodo = nodo;
24    } // fin del constructor de NodoLista con dos argumentos
25
26    // devuelve la referencia a datos en el nodo
27    Object obtenerObject()
28    {
29       return datos; // devuelve el objeto Object en este nodo
30    } // fin del método obtenerObject
31
32    // devuelve la referencia al siguiente nodo en la lista
33    NodoLista obtenerSiguiente()
34    {
35       return siguienteNodo; // obtiene el siguiente nodo
```

Figura 17.3 | Declaraciones de las clases `NodoLista` y `Lista`. (Parte 1 de 3).

```
36      } // fin del método obtenerSiguiente
37   } // fin de la clase NodoLista
38
39   // definición de la clase Lista
40   public class Lista
41   {
42      private NodoLista primerNodo;
43      private NodoLista ultimoNodo;
44      private String nombre; // cadena como "lista", utilizada para imprimir
45
46      // el constructor crea una Lista vacía con el nombre "lista"
47      public Lista()
48      {
49         this( "lista" );
50      } // fin del constructor de Lista sin argumentos
51
52      // el constructor crea una Lista vacía con un nombre
53      public Lista( String nombreLista )
54      {
55         nombre = nombreLista;
56         primerNodo = ultimoNodo = null;
57      } // fin del constructor de Lista con un argumento
58
59      // inserta objeto Object al frente de la Lista
60      public void insertarAlFrente( Object elementoInsertar )
61      {
62         if ( estaVacia() ) // primerNodo y ultimoNodo hacen referencia al mismo objeto
63            primerNodo = ultimoNodo = new NodoLista( elementoInsertar );
64         else // primerNodo hace referencia al nuevo nodo
65            primerNodo = new NodoLista( elementoInsertar, primerNodo );
66      } // fin del método insertarAlFrente
67
68      // inserta objeto Object al final del la Lista
69      public void insertarAlFinal( Object elementoInsertar )
70      {
71         if ( estaVacia() ) // primerNodo y ultimoNodo hacen referencia al mismo objeto
72            primerNodo = ultimoNodo = new NodoLista( elementoInsertar );
73         else // el siguienteNodo de ultimoNodo hace referencia al nuevo nodo
74            ultimoNodo = ultimoNodo.siguienteNodo = new NodoLista( elementoInsertar );
75      } // fin del método insertarAlFinal
76
77      // elimina el primer nodo de la Lista
78      public Object eliminarDelFrente() throws ExcepcionListaVacia
79      {
80         if ( estaVacia() ) // lanza excepción si la Lista está vacía
81            throw new ExcepcionListaVacia( nombre );
82
83         Object elementoEliminado = primerNodo.datos; // obtiene los datos que se van a
                                                        eliminar
84
85         // actualiza las referencias primerNodo y ultimoNodo
86         if ( primerNodo == ultimoNodo )
87            primerNodo = ultimoNodo = null;
88         else
89            primerNodo = primerNodo.siguienteNodo;
90
91         return elementoEliminado; // devuelve los datos del nodo eliminado
92      } // fin del método eliminarDelFrente
93
```

Figura 17.3 | Declaraciones de las clases `NodoLista` y `Lista`. (Parte 2 de 3).

```
94      // elimina el último nodo de la Lista
95      public Object eliminarDelFinal() throws ExcepcionListaVacia
96      {
97         if ( estaVacia() ) // lanza excepción si la Lista está vacía
98            throw new ExcepcionListaVacia( nombre );
99
100        Object elementoEliminado = ultimoNodo.datos; // obtiene los datos que se van a
                                                        eliminar
101
102        // actualiza las referencias primerNodo y ultimoNodo
103        if ( primerNodo == ultimoNodo )
104           primerNodo = ultimoNodo = null;
105        else // localiza el nuevo último nodo
106        {
107           NodoLista actual = primerNodo;
108
109           // itera mientras el nodo actual no haga referencia a ultimoNodo
110           while ( actual.siguienteNodo != ultimoNodo )
111              actual = actual.siguienteNodo;
112
113           ultimoNodo = actual; // actual el nuevo ultimoNodo
114           actual.siguienteNodo = null;
115        } // fin de else
116
117        return elementoEliminado; // devuelve los datos del nodo eliminado
118     } // fin del método eliminarDelFinal
119
120     // determina si la lista está vacía
121     public boolean estaVacia()
122     {
123        return primerNodo == null; // devuelve true si la lista está vacía
124     } // fin del método estaVacia
125
126     // imprime el contenido de la lista
127     public void imprimir()
128     {
129        if ( estaVacia() )
130        {
131           System.out.printf( "%s vacia\n", nombre );
132           return;
133        } // fin de if
134
135        System.out.printf( "La %s es: ", nombre );
136        NodoLista actual = primerNodo;
137
138        // mientras no esté al final de la lista, imprime los datos del nodo actual
139        while ( actual != null )
140        {
141           System.out.printf( "%s ", actual.datos );
142           actual = actual.siguienteNodo;
143        } // fin de while
144
145        System.out.println( "\n" );
146     } // fin del método imprimir
147  } // fin de la clase Lista
```

Figura 17.3 | Declaraciones de las clases `NodoLista` y `Lista`. (Parte 3 de 3).

La clase `NodoLista` (figura 17.3, líneas 6 a 37) declara los campos de acceso del paquete llamados `datos` y `siguienteNodo`. El campo `datos` es una referencia `Object`, por lo que puede hacer referencia a cualquier objeto.

El miembro `siguienteNodo` de `NodoLista` almacena una referencia al siguiente objeto `NodoLista` en la lista enlazada (o `null`, si el nodo es el último en la lista).

En las líneas 42 y 43 de la clase `Lista` (figura 17.3, líneas 40 a 147) se declaran referencias al primer y último objetos `NodoLista` en un objeto `Lista` (`primerNodo` y `ultimoNodo`, respectivamente). Los constructores (líneas 47 a 50 y 53 a 57) inicializan ambas referencias con null. Los métodos más importantes de la clase `Lista` son `insertarAlFrente` (líneas 60 a 66), `insertarAlFinal` (líneas 69 a 75), `eliminarDelFrente` (líneas 78 a 92) y `eliminarDelFinal` (líneas 95 a 118). El método `estaVacia` (líneas 121 a 124) es un *método predicado*, el cual determina si la lista está vacía (es decir, si la referencia al primer nodo de la lista es `null`). Los métodos predicado generalmente evalúan una condición y no modifican el objeto en el que son llamados. Si la lista está vacía, el método `estaVacia` devuelve `true`; en caso contrario, devuelve `false`. El método `imprimir` (líneas 127 a 146) muestra el contenido de la lista. Después de la figura 17.5 hay una discusión detallada sobre los métodos de `Lista`.

El método `main` de la clase `PruebaLista` (figura 17.5) inserta objetos al principio de la lista utilizando el método `insertarAlFrente`, inserta objetos al final de la lista utilizando el método `insertarAlFinal`, elimina objetos de la parte frontal de la lista utilizando el método `eliminarDelFrente` y elimina objetos de la parte final de la lista utilizando el método `eliminarDelFinal`. Después de cada operación de inserción y eliminación, `PruebaLista` invoca al método de `Lista` llamado `imprimir` para mostrar el contenido actual de la lista. Si se trata de eliminar un elemento de una lista vacía, se lanza una excepción del tipo `ExcepcionListaVacia` (figura 17.4), de manera que las llamadas a los métodos `eliminarDelFrente` y `eliminarDelFinal` se colocan en un bloque `try` que va seguido de un manejador de excepciones apropiado. Observe en las líneas 13, 15, 17 y 19 que la aplicación pasa valores literales `int` primitivos a los métodos `insertarAlFrente` e `insertarAlFinal`, aun cuando cada uno de estos métodos se declaró con un parámetro de tipo `Object` (figura 17.3, líneas 60 y 69). En este caso, la JVM realiza la conversión autobox de cada valor literal a un objeto `Integer`, y ese objeto es el que se inserta en la lista. Desde luego que esto se permite debido a que `Object` es una superclase indirecta de `Integer`.

```
1  // Fig. 17.4: ExcepcionListaVacia.java
2  // Definición de la clase ExcepcionListaVacia.
3  package com.deitel.jhtp7.cap17;
4
5  public class ExcepcionListaVacia extends RuntimeException
6  {
7     // constructor sin argumentos
8     public ExcepcionListaVacia()
9     {
10       this( "Lista" ); // llama al otro constructor de ExcepcionListaVacia
11    } // fin del constructor de ExcepcionListaVacia sin argumentos
12
13    // constructor con un argumento
14    public ExcepcionListaVacia( String nombre )
15    {
16       super( nombre + " esta vacia" ); // llama al constructor de la superclase
17    } // fin del constructor de ExcepcionListaVacia con un argumento
18 } // fin de la clase ExcepcionListaVacia
```

Figura 17.4 | Declaración de la clase `ExcepcionListaVacia`.

```
1  // Fig. 17.5: PruebaLista.java
2  // Clase PruebaLista para demostrar las capacidades de Lista.
3  import com.deitel.jhtp7.cap17.Lista;
4  import com.deitel.jhtp7.cap17.ExcepcionListaVacia;
5
6  public class PruebaLista
7  {
```

Figura 17.5 | Manipulaciones de listas enlazadas. (Parte 1 de 2).

```
 8     public static void main( String args[] )
 9     {
10        Lista lista = new Lista(); // crea el contenedor de Lista
11
12        // inserta enteros en lista
13        lista.insertarAlFrente( -1 );
14        lista.imprimir();
15        lista.insertarAlFrente( 0 );
16        lista.imprimir();
17        lista.insertarAlFinal( 1 );
18        lista.imprimir();
19        lista.insertarAlFinal( 5 );
20        lista.imprimir();
21
22        // elimina objetos de lista; imprime después de cada eliminación
23        try
24        {
25           Object objetoEliminado = lista.eliminarDelFrente();
26           System.out.printf( "%s eliminado\n", objetoEliminado );
27           lista.imprimir();
28
29           objetoEliminado = lista.eliminarDelFrente();
30           System.out.printf( "%s eliminado\n", objetoEliminado );
31           lista.imprimir();
32
33           objetoEliminado = lista.eliminarDelFinal();
34           System.out.printf( "%s eliminado\n", objetoEliminado );
35           lista.imprimir();
36
37           objetoEliminado = lista.eliminarDelFinal();
38           System.out.printf( "%s eliminado\n", objetoEliminado );
39           lista.imprimir();
40        } // fin de try
41        catch ( ExcepcionListaVacia excepcionListaVacia )
42        {
43           excepcionListaVacia.printStackTrace();
44        } // fin de catch
45     } // fin de main
46  } // fin de la clase PruebaLista
```

```
La lista es: -1

La lista es: 0 -1

La lista es: 0 -1 1

La lista es: 0 -1 1 5

0 eliminado
La lista es: -1 1 5

-1 eliminado
La lista es: 1 5

5 eliminado
La lista es: 1

1 eliminado
lista vacia
```

Figura 17.5 | Manipulaciones de listas enlazadas. (Parte 2 de 2).

Ahora hablaremos detalladamente sobre cada uno de los métodos de la clase `Lista` (figura 17.3) y proporcionaremos diagramas que muestren las manipulaciones de referencia realizadas por los métodos `insertarAlFrente`, `insertarAlFinal`, `eliminarDelFrente` y `eliminarDelFinal`. El método `insertarAlFrente` (líneas 60 a 66 de la figura 17.3) coloca un nuevo nodo al frente de la lista. Los pasos son:

1. Llamar a `estaVacia` para determinar si la lista está vacía (línea 62).
2. Si la lista está vacía, asignar `primerNodo` y `ultimoNodo` al nuevo `NodoLista` que se inicializó con `elementoInsertar` (línea 63). El constructor de `NodoLista` en las líneas 13 a 16 llama al constructor de `NodoLista` en las líneas 20 a 24 para establecer la variable de instancia `datos`, para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y para establecer la referencia `siguienteNodo` en `null`, ya que éste es el primer y último nodo en la lista.
3. Si la lista no está vacía, el nuevo nodo se "enlaza" en la lista asignando a `primerNodo` un nuevo objeto `NodoLista`, e inicializando ese objeto con `elementoInsertar` y `primerNodo` (línea 65). Cuando se ejecuta el constructor de `NodoLista` (líneas 20 a 24), establece la variable de instancia `datos` para que haga referencia al `elementoInsertar` que se pasa como argumento, y realiza la inserción asignando a la referencia `siguienteNodo` del nuevo nodo al objeto `NodoLista` que se pasa como argumento, y que anteriormente era el primer nodo.

En la figura 17.6, la parte (a) muestra una lista y un nuevo nodo durante la operación `insertarAlFrente` y antes de que el programa enlace el nuevo nodo a la lista. Las flechas punteadas en la parte (b) ilustran el *paso 3* de la operación `insertarAlFrente`, en donde se permite al nodo que contiene `12` convertirse en el primer nuevo nodo en la lista.

El método `insertarAlFinal` (líneas 69 a 75 de la figura 17.3) coloca un nuevo nodo al final de la lista. Los pasos son:

1. Llamar a `estaVacia` para determinar si la lista está vacía (línea 71).
2. Si la lista está vacía, asignar `primerNodo` y `ultimoNodo` al nuevo `NodoLista` que se inicializó con `elementoInsertar` (línea 72). El constructor de `NodoLista` en las líneas 13 a 16 llama al constructor de las líneas 20 a 24 para establecer la variable de instancia `datos`, para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y para establecer la referencia `siguienteNodo` en `null`.
3. Si la lista no está vacía, en la línea 74 se enlaza el nuevo nodo a la lista, asignando a `ultimoNodo` y a `ultimoNodo.siguienteNodo` la referencia al nuevo `NodoLista` que se inicializó con `elementoInsertar`. El constructor de `NodoLista` (líneas 13 a 16) establece la variable de instancia `datos` para hacer referencia al objeto `elementoInsertar` que se pasa como argumento y establece la referencia `siguienteNodo` en `null`, ya que éste es el último nodo en la lista.

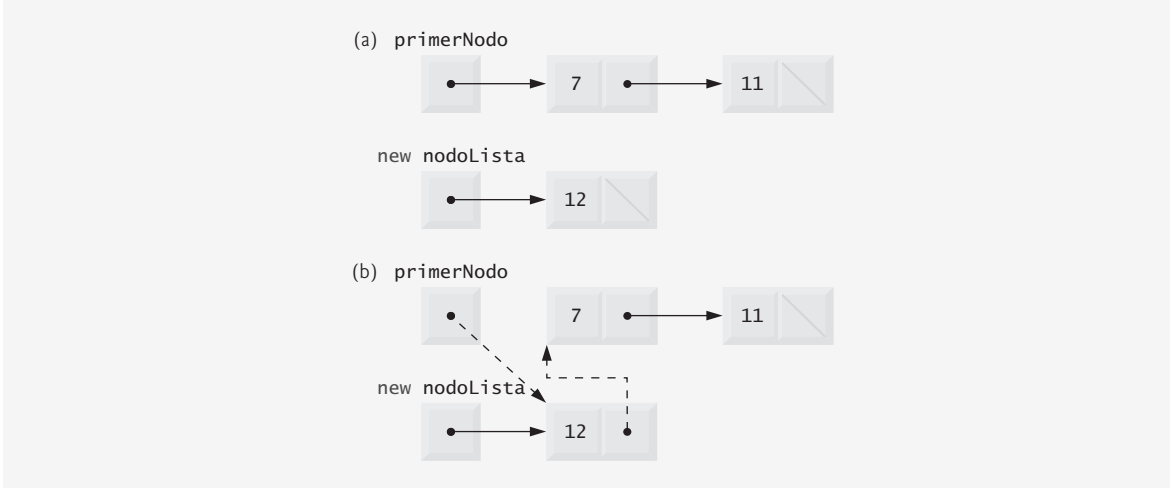

Figura 17.6 | Representación gráfica de la operación `insertarAlFrente`.

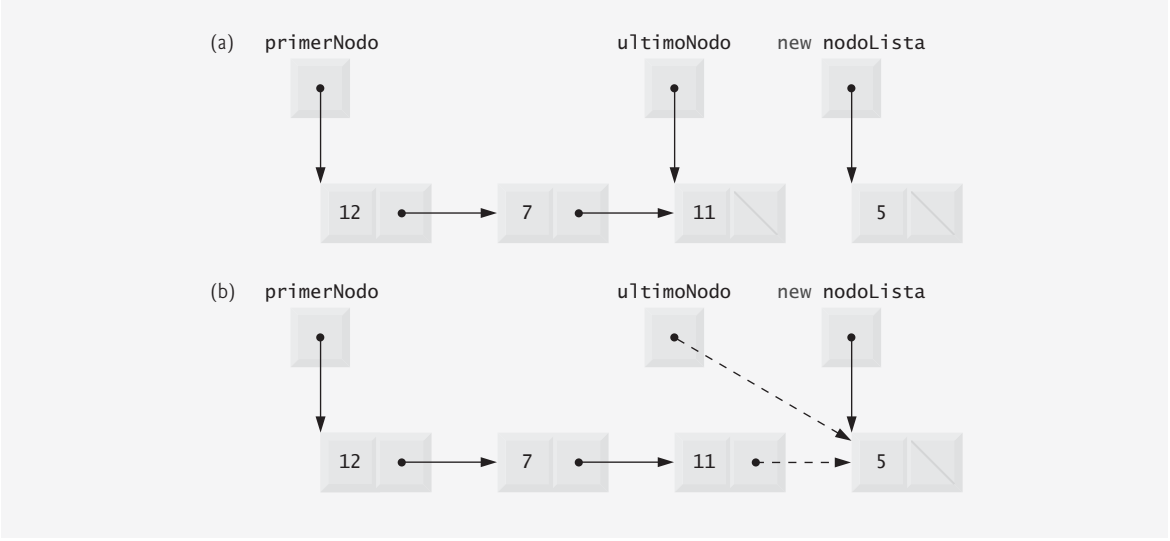

Figura 17.7 | Representación gráfica de la operación `insertarAlFinal`.

En la figura 17.7, la parte (a) muestra una lista y un nuevo nodo durante la operación `insertarAlFrente` y antes de que el programa enlace el nuevo nodo a la lista. Las flechas punteadas en la parte (b) ilustran el *paso 3* del método `insertarAlFinal`, el cual agrega el nuevo nodo al final de una lista que no está vacía.

El método `eliminarDelFrente` (líneas 78 a 92 de la figura 17.3) elimina el primer nodo de la lista y devuelve una referencia a los datos eliminados. El método lanza una excepción `ExcepcionListaVacia` (líneas 80 y 81) si la lista está vacía cuando el programa llama a este método. De no ser así, el método devuelve una referencia a los datos eliminados. Los pasos son:

1. Asignar `primerNodo.datos` (los datos que se van a eliminar de la lista) a la referencia `elementoEliminado` (línea 83).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 86), quiere decir que la lista sólo tiene un elemento en ese momento. Por lo tanto, el método establece a `primerNodo` y `ultimoNodo` en `null` (línea 87) para eliminar el nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, entonces el método deja la referencia `ultimoNodo` como está y asigna el valor de `primerNodo.siguienteNodo` a `primerNodo` (línea 89). Por lo tanto, `primerNodo` hace referencia al nodo que era anteriormente el segundo nodo en la lista.
4. Devolver la referencia `elementoEliminado` (línea 91).

En la figura 17.8, la parte (a) ilustra la lista antes de la operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de referencias.

El método `eliminarDelFinal` (líneas 95 a 118 de la figura 17.3) elimina el último nodo de una lista y devuelve una referencia a los datos eliminados. El método lanza una excepción `ExcepcionListaVacia` (líneas 97 y 98) si la lista está vacía cuando el programa llama a este método. Los pasos son:

1. Asignar `ultimoNodo.datos` (los datos que se van a eliminar de la lista) a `elementoEliminado` (línea 100).
2. Si `primerNodo` y `ultimoNodo` hacen referencia al mismo objeto (línea 103), quiere decir que la lista sólo tiene un elemento en ese momento. Por lo tanto, en la línea 104 se establece a `primerNodo` y `ultimoNodo` en `null` para eliminar ese nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo, crear la referencia `NodoLista` llamada `actual` y asignarla a `primerNodo` (línea 107).

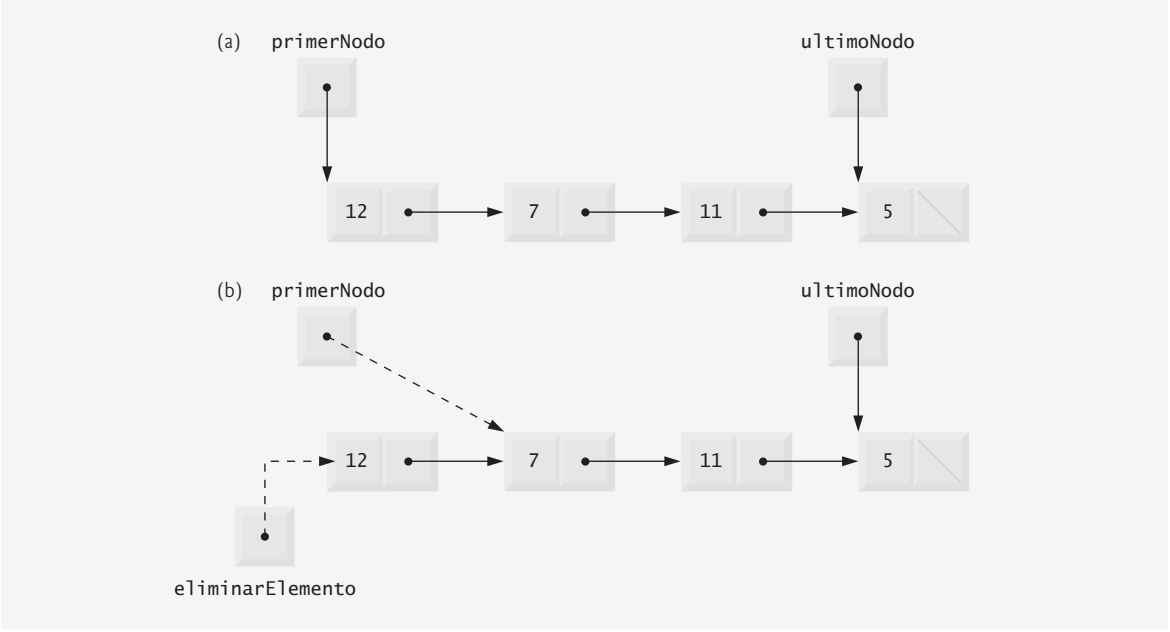

Figura 17.8 | Representación gráfica de la operación `eliminarDelFrente`.

4. Ahora hay que "recorrer la lista" con `actual` hasta que haga referencia al nodo que esté antes del último nodo. El ciclo `while` (líneas 110 y 111) asigna `actual.siguienteNodo` a `actual`, siempre y cuando `actual.siguienteNodo` (el siguiente nodo en la lista) no sea `ultimoNodo`.
5. Después de localizar el penúltimo nodo, asignar `actual` a `ultimoNodo` (línea 113) para actualizar cuál nodo es el último en la lista.
6. Establecer `actual.siguienteNodo` en `null` (línea 114) para eliminar el último nodo de la lista y terminarla con el nodo actual.
7. Devolver la referencia `elementoEliminado` (línea 117).

En la figura 17.9, la parte (a) ilustra la lista antes de la operación de eliminación. Las líneas punteadas y las flechas en la parte (b) muestran las manipulaciones de referencias.

El método `imprimir` (líneas 127 a 146) determina primero si la lista está vacía (líneas 129 a 133). De ser así, `imprimir` muestra un mensaje indicando que la lista está vacía y devuelve el control al método que hizo la llamada. En caso contrario, `imprimir` muestra en pantalla los datos en la lista. En la línea 136 se crea la referencia `NodoLista` llamada `actual` y se inicializa con `primerNodo`. Mientras que `actual` no sea `null`, hay más elementos en la lista. Por lo tanto, en la línea 141 se muestra en pantalla una representación de cadena de `actual.datos`. En la línea 142 se avanza al siguiente nodo en la lista mediante la asignación del valor de la referencia `actual.siguienteNodo` a `actual`. Este algoritmo de impresión es idéntico para listas enlazadas, pilas y colas.

17.7 Pilas

Una pila es una versión restringida de una lista enlazada; pueden agregarse y eliminarse nuevos nodos en una pila solamente desde su parte superior. [*Nota:* una pila no tiene que implementarse mediante el uso de una lista enlazada]. Por esta razón, a una pila se le conoce como **estructura de datos UEPS (último en entrar, primero en salir)**. El miembro de enlace en el nodo inferior (es decir, el último) de la pila se establece en `null` para indicar el fondo de la pila.

Los métodos básicos para manipular una pila son **`push`** (empujar) y **`pop`** (sacar). El método `push` agrega un nuevo nodo a la parte superior de la pila. El método `pop` elimina un nodo de la parte superior de la pila y devuelve los datos del nodo que se quitó.