11 | Contactos

Una vez que se conocen las construcciones de un lenguaje de programación para controlar el flujo de ejecución del código, es posible manejar cantidades de información cuyo tamaño se desconoce al momento de codificar el programa. En esta práctica no sólo usarás listas para contar con un almacén de datos de tamaño variable, sino que las programarás.

Meta

Que el alumno aprenda a utilizar correctamente las referencias a objetos.

Objetivos

Al finalizar la práctica el alumno será capaz de:

1. Idenficar los efectos colaterales producidos por el acceso a objetos a través de referencias.
2. Aprovechar el uso de referencias para implementar estructuras lineales recursivas de tamaño variable.
3. Programar y utilizar una lista simplemente ligada.

Código Auxiliar 11.1: Contactos

https://github.com/computacion-ciencias/icc-contactos

Antecedentes

En esta práctica programarás tu propia lista de contactos. Cualquier lista de contactos que se respete debe permitirle al usuario guardar tantos contactos como el usuario desee

(mientras haya memoria disponible, claro). Sin embargo ¿cómo podrías guardar 100 registros sin tener que declarar 100 variables? ¿Qué tal si sólo necesitabas 15? ¿Qué tal si llegas a Presidente de la República y tienes un millón de contactos? Bueno... tal vez eso sea un poco más complicado ¿Qué tal si simplemente no sabes? En resumen, el objetivo es poder almacenar un número variable de elementos, sin saber cuántos serán al momento de hacer el programa.

Para lograrlo utilizarás una técnica muy popular que consiste en hacer que cada objeto sepa dónde se encuentra el que sigue y así sucesivamente. Mientras tú tengas la dirección del primer objeto en tu lista, podrás llegar a cualquier otro siguiendo la fila a partir del primero. Esta es la técnica que se utiliza para programar listas.

Listas

Una lista es un tipo abstracto de datos que nos permite tener un número arbitrario de elementos de un tipo dado. Formalmente, la podemos definir como:

Definición 11.1: Lista

Una *lista* es:

- Una lista vacía.
- Un elemento seguido de otra lista.

La definición de una lista es *recursiva*: se define en términos de sí misma. En este caso decimos que se trata de una *recursión estructural*, pues las listas son objetos (estructuras) que contienen objetos del mismo tipo que ellos.

Toda recursión contiene en su definición un *caso base*, es decir, un caso muy sencillo a partir del cual se construyen los más complejos. En este caso el caso base es cuando la lista es una lista vacía, pues esta ya no contiene a otra lista. Puedes visualizarla así:

$$\mathrm{l} = \{\}$$

A partir de ahí se construyen las listas más largas, si los elementos son letras, otros ejemplos de listas son:

$$\mathrm{l}_1 = \{\mathrm{A}\{\}\}$$
$$\mathrm{l}_2 = \{\mathrm{L}\{\mathrm{I}\{\mathrm{S}\{\mathrm{T}\{\mathrm{A}\{\}\}\}\}\}\}$$

Observa que cada lista contiene a un elemento y otra lista dentro, al final siempre queda un lista vacía.

Para programar una estructura como esta en Java haremos uso de los recursos siguientes:

- Las variables cuyo tipo son clases son variables de tipo *referencia*, es decir, no contienen realmente al objeto, sino sólo la dirección que éste tiene en el *heap*.
- Utilizaremos el valor asociado a la ausencia de un objeto con la lista vacía. Este valor especial es `null`. Cuando una variable vale `null` es porque no tiene la dirección de ningún objeto. Pensaremos entonces que una variable de tipo `Lista` que valga `null` correspondería a una lista vacía.
- Podemos definir clases con atributos que tienen como tipo a la misma clase, pues esto sólo significa que el objeto tiene una referencia (dirección) a otro objeto del mismo tipo.

Si nuestra lista contiene objetos de tipo cadena, podemos definir fácilmente a la clase con los atributos siguientes:

Listado 11.1: Lista.java

```
1 public class Lista {
2   private String elemento;
3   private Lista siguiente;
4 }
```

Obseva sin embargo, que las operaciones que realicemos sobre la lista: insertar, buscar, borrar, serán las mismas independientemente que se trate de una lista de `String`, `Correos`, `Estudiantes`, etc. Para programar una lista que funcione para cualquier tipo de contenido aprovecharemos la **herencia**. Como todos los objetos heredan de `Object`, una referencia de tipo `Object` puede contener la dirección de cualquier tipo de dato[1]. De este modo el código anterior se convierte en:

Listado 11.2: Lista.java

```
1 public class Lista {
2   private Object elemento;
3   private Lista siguiente;
4 }
```

Ahora, los métodos que operan sobre las listas tendrán que trabajar de forma distinta dependiendo de si ésta es vacía o no (caso base o caso recursivo). En orientación a objetos nos gusta que este tipo de detalles queden ocultos al programador y que el usuario de una lista no se vea en la necesidad de lidiar con ellos. Después de todo, una lista es una lista, como la lista de alumnos de un grupo. Entonces modificaremos un poco la estructura del código anterior.

En Java, una forma estándar de programar listas es como una *lista simplemente ligada*, haciendo uso de dos tipos de objetos:

[1] Claro, excepto datos primitivos.

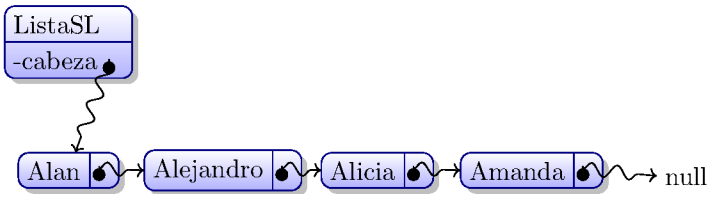

Figura 11.1 Un objeto de tipo `ListaSL` (por *lista simplemente ligada*) tiene una referencia al contenedor con el primer dato. Cada contenedor contiene su dato y la referencia al contenedor con el siguiente dato.

- Una *lista*, que contendrá solamente la dirección de su primer elemento (al cual típicamente se le conoce como `cabeza`), además de los métodos que podrá utilizar el usuario (insertar, consultar, remover, etc.) y
- *contenedores*, que guardarán dentro datos y la dirección del siguiente contenedor. Estos contenedores pueden recibir distintos nombres, pero los más populares son `Nodo` y `Elemento`.

La Figura 11.1 ilustra cómo se visualiza esta esta estructura en la memoria de la computadora.

Recorriendo una lista

Para acceder a un elemento en este tipo de estructura es necesario recorrer a todos los elementos, desde el primero, hasta encontrar al que buscamos:

Listado 11.3: Componentes de una lista simplemente ligada.

```
1  /** Estructura recursiva.
2    * Sólo es visible dentro del paquete. */
3  class Nodo {
4    private Object dato;
5    private Nodo siguiente;
6    // métodos de lectura y escritura ...
7  }
8
9  /** Se debe implementar esta interfaz para establecer la
10     condición del dato que se busca. */
11 public interface Condición {
12   boolean satisface(Object o);
13 }
14
15 /** Clase para usar una lista. */
16 public class Lista {
17   private Nodo cabeza;
18
```

```
19    public Object satisface(Condición c) {
20      Nodo temp = cabeza;           // Buscamos desde la cabeza
21      while(temp != null) {         // Mientras no lleguemos al final
22        if (c.satisface(temp.dato())) {
23          return temp.dato();       // Si lo encontramos termina
24        }
25        temp = temp.siguiente();    // Sigue buscando
26      }
27      return null;                  // No está
28    }
29  }
```


Agregar al final

Para agregar objetos al final de la lista es necesario realizar tres acciones:

- Recorrer a la lista desde el primer elemento hasta llegar al último. Observa que la característica distintiva del último elemento es que su nodo siguiente es `null`.
- Crear un nodo nuevo, que contenga al objeto que se quiere almacenar. Para ello es conveniente que la clase `Nodo` tenga un constructor que reciba el objeto como parámetro y lo asigne al atributo correspondiente.
- Hacer que el último nodo almacene en su atributo `siguiente` la dirección del nodo que acabamos de crear.

Veamos a continuación los fragmentos de código relevantes para esta acción, entendiendo que, dentro de cada clase, aún están los elementos que habíamos listado antes.

Listado 11.4: Agregar al final de una lista simplemente ligada.

```
1  class Nodo {
2    Nodo(Object o) {
3      dato = o;
4    }
5  }
6
7  public class Lista {
8    public void agrega(Object o) {
9      if(cabeza == null) {  // Si aún no hay primer elemento
10       cabeza = new Nodo(o);
11     } else {
12       Nodo temp = cabeza; // Recorrer hasta encontrar dónde insertar
13       while(temp.siguiente() != null) {
14           temp = temp.siguiente();
15       }
16       Nodo nuevo = new Nodo(o);
17       temp.siguiente(nuevo);
```

```
18        }
19      }
20    }
```

Interfaces funcionales y lambdas

En Java, cuando requerimos especificar el algoritmo a ejecutar, más que un conjunto de datos, utilizamos *interfaces funcionales* para pasar objetos capaces de ejecutar el algoritmo que necesitamos. Se caracteriza por contener el encabezado de un solo método que se debe implementar.

Una particularidad muy útil de las interfaces funcionales es que, a partir de Java8, no es necesario implementar dicha interfaz con una clase, sino que se puede utilizar una construcción más sencilla conocida como *lambda*. Una `lambda` se utiliza para especificar ese único método a implementar.

Ejemplo 11.1. *En el Listado 11.3, `Condición` es una interfaz funcional que se utiliza para indicar el código con el cual se decidirá si el objeto en la lista es el que está buscando el usuario.*

Supongamos tenemos una lista de cadenas y buscamos el primer elemento que comience con la palabra `"señal"`. Suponiendo que la clase `Lista` ya tiene un método para agregar en el orden en el que se insertan los datos, el código para buscar el dato se vería como sigue:

```
1  public class UsoLista {
2    public static void llenaLista(Lista l) {
3      l.agrega("Hola.");
4      l.agrega("Espera␣a␣que");
5      l.agrega("envíe␣la");
6      l.agrega("señal␣acordada.");
7    }
8
9    public static void main(String[] args) {
10     Lista l = new Lista();
11     llenaLista(l);
12     String s = (String)l.satisface(
13       o -> ((String)o).startsWith("señal")
14       System.out.format("Encontré:␣\"%s\"%n", s);
15     )
16   }
17 }
```

Al ejecutar el programa la consola mostrará:

```
Encontré: "señal␣acordada."
```

Método toString()

Como dijimos, toda clase de `Java` hereda de la clase `Object` ya sea directa o indirectamente. La clase `Object` tiene un método llamado `toString()` cuyo trabajo es devolver una cadena que represente al objeto y todas las clases heredan este método. Por defecto, esta cadena lista información como la clase del objeto y la dirección de memoria donde se encuentra. Pero si quieres una representación más amigable para los objetos de las clases que programes, lo que debes hacer es *sobreescribir* este método, es decir, declarar un método en tu clase que tenga la misma *firma*. En el código de ese método devuelve la cadena que quieres que represente a tu objeto. Ahora cada vez que `Java` llame al método `toString()` para tu objeto, verás la cadena que tú devolviste. El ejemplo más inmediato es cuando uses `System.out.println()` pues por dentro manda llamar a este método.

Listado 11.5: El método `println()` manda llamar al método toString()

```
1  public class Rosa {
2    public String toString() {
3      return "Flor␣roja␣con␣hermoso␣aroma";
4    }
5  }
6
7  public class DemoToString {
8    public static void main(String[] args) {
9      Rosa rosa = new Rosa();
10     System.out.println(rosa);
11   }
12 }
```

Desarrollo

Actividad 11.1

Lee la documentación del método `System.out.println`.

Entregable: ¿De qué clase es instancia el objeto `out`? ¿Cuántas veces está sobrecargado el método `println`, da algunos ejemplos? https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out.

Actividad 11.2

Entregable: Explica ¿por qué hay una versión de `println` por cada tipo primitivo de `Java`?

Ejercicios

1. Crea una clase `Registro` dentro del paquete `icc.contactos`, cuyos atributos serán `nombre`, `dirección` y `teléfono`. ¿De qué tipo debe ser cada uno de esos atributos? Elígelos tú. Recuerda que estas variables deben ser privadas.

2. Ponle un constructor que reciba estos tres datos como parámetros, en ese orden y los asigne a los atributos correspondientes. Asegúrate de que no sean `null`, si el parámetro valiera `null`, asígnales valores por defecto.

3. Agrega métodos de lectura (*getters*) y escritura (*setters*) para cada atributo, asegúrate de que los valores asignados no sean `null`. Decide cómo se comportará tu programa en caso de que el parámetro valga `null` y documenta tu decisión en el método correspondiente.

4. Sobreescribe el método `toString()`. Agrega un método main a tu clase `Registro`, crea un par de registros con datos e imprímelos. Verifica que funcionen correctamente.

5. Ahora haz que `Registro` implemente la interfaz provista `IComparable` modificando la declaración de la clase:

```
1  package icc.contactos;
2
3  import icc.util.IComparable;
4
5  public class Registro implements IComparable {
6    // Ahora Registro está obligado a implementar el método
       ↪ comparaCon
7  }
```

   Revisa la documentación del método `compareTo(String a)` de la clase `String`, lo necesitarás para determinar el orden correcto, pues querrás que los registros se ordenen alfabéticamente de acuerdo a los nombres que almacenan.

6. Crea ahora la clase `Nodo` dentro del paquete `icc.util`. Como nos interesa programar una lista que mantiene sus datos ordenados, `Nodo` deberá tener un atributo de tipo `IComparable` en lugar del típico `Object`, éste contendrá los datos, y otro atributo de tipo `Nodo`, que nos indicará la dirección del nodo siguiente de la lista. Agrega sus métodos de lectura y escritura correspondientes, pero no les pongas acceso. Esto hará que sólo las clases dentro del mismo `paquete` (directorio) puedan mandarlos llamar. Esto permitirá que tu lista pueda acceder a estos métodos para agregar, consultar, modificar o borrar sus datos – aunque para esta práctica sólo estarás agregando y consultando. Revisa que tu código compile.

7. En este mismo directorio crea ahora la clase `ListaSL`. Tendrá un atributo de tipo `Nodo` llamado `cabeza`. Al inicio esta variable valdrá `null`, indicando que tu lista está vacía.

8. Se te da parte del código de la interfaz de usuario en la clase `IUContactos`, completa el contenido del método `solicitaDatosDeContacto()` para que funcione con tu clase `Registro`.

9. Agrega a tu clase `ListaSL` un método `public String toString()` que devuelva una cadena con los datos almacenados en todos los nodos de la lista. Si la lista está vacía devuelve la cadena con un salto de línea `"\n"`. Si no, itera sobre los nodos de la lista, agregando la representación `toString()` del dato en cada nodo. Ojo: no intentes imprimir en la consola dentro de este método, sólo construye y devuelve la cadena haciendo las concatenaciones necesarias.

   TIP: Revisa la documentación de la clase `StringBuilder`.

10. Agrega el método `public void inserta(IComparable n)`. Este método debe recibir un objeto de tipo `IComparable`. Creará un `Nodo` nuevo y guardará ahí a `n`. El nodo nuevo deberá ser colocado en la posición correcta según lo que devuelva el método `comparaCon` aplicado a sus datos, de modo que la lista esté en orden.

    Esto significa que rara vez insertarás el nodo nuevo al final, podría ir también al inicio o en medio. Para ello deberás encontrar al nodo con el dato que debe quedar justo antes del que estás insertando, si lo hay.

    Aquí es donde tu clase `ListaSL` necesitará acceder a los métodos:

    - `leeDato`,
    - `leeSiguiente` y
    - `asignaSiguiente`

    de la clase `Nodo` para encontrar el registro anterior al nuevo e insertar el registro nuevo en su lugar. NOTA: No es obligatorio que los métodos se llamen así, esta vez tú puedes elegir el nombre.

    Diseña con cuidado cuándo debes modificar el atributo `cabeza` de la lista y qué hacer con los atributos `siguiente` tanto del nodo que debe preceder al que vas a insertar, como del nodo que estás insertando. Ten mucho cuidado de no perder elementos de tu lista o, peor aún, la cabeza.

11. Comienza ahora a completar la clase `icc.contactos.Contactos`, que implemente la interfaz `IContactos`. En realidad esta clase usará la `ListaLS` que acabas de programar para almacenar a los contactos y se encargará de guardar Registros en ella y devolverlos cuando le sean requeridos. Como los métodos de `ListaLS` devuelven objetos de tipo `IComparable`, tendrás que realizar a aquí las audiciones (*castings*) necesarias para recordarle al compilador el tipo original de los objetos que estás almacenando en la lista. Comienza por el método `public void insertaContacto(Registro reg)`.

12. Completa el método `public void imprimeContactos()` que recorra tu lista de contactos y los vaya imprimiendo. Deberán aparecen en orden alfabético. Este es un buen momento para probar tanto la inserción como la impresión de tu lista de contactos. Compila, corre tu código y prueba esas opciones del menú. Si encuentras errores, arréglalos de una vez.

13. Completa el método `public Registro consultar(String nombre)` que recibe el nombre o una parte del nombre del contacto que quieres consultar. Este método deberá devolver el primer registro que contenga la cadena indicada. Revisa el funcionamiento del método `contains` de la clase `String`, lo necesitarás para resolver esta parte. Nota que deberás revisar todos los registros para ver si alguno contiene la cadena indicada, si ninguno la contiene regresa `null`.

14. Compila y corre tu programa. Ya debe estar funcionando, prueba cada opción del menú y verifica que así sea. Si encuentras errores trata de irlos arreglando opción por opción.

15. Documenta todo tu código y corrige cualquier error de indentación en todas las clases.

Entregables

Lo que deberás entregar para esta práctica es:

1. Respuestas a las actividades:
   a) 11.1
   b) 11.2
2. Los archivos correspondientes a los ejercicios.
   - `icc.contactos.Registro.java`
   - `icc.util.Nodo.java`
   - `icc.util.ListaSL.java`
   - Método `solicitaDatosDeContacto()` en `IUContactos.java`
   - `icc.contactos.Contactos` completada
3. La respuesta a la pregunta: Al buscar por nombre ¿qué necesitarías hacer para devolver todos los registros que contengan la cadena indicada, en lugar de sólo el primero? Descríbelo, no es necesario que lo programes.