

## MANEJO DE GRUPOS DE ATRIBUTOS

03



# 1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Utilizar las estructuras contenedoras de tamaño fijo como elementos para modelar una característica de un elemento del mundo que permiten almacenar una secuencia de valores (simples u objetos).
- Utilizar las estructuras contenedoras de tamaño [variable](#) como elementos de modelado que permiten manejar atributos cuyo valor es una secuencia de objetos.
- Utilizar las instrucciones iterativas para manipular estructuras contenedoras y entender que dichas instrucciones se pueden utilizar en otro tipo de problemas.
- Crear una [clase](#) completa en Java utilizando el [ambiente de desarrollo](#) Eclipse.
- Entender la documentación de un conjunto de clases escritas por otros y utilizar dicha documentación para poder incorporar y usar adecuadamente dichas clases en un programa que se está construyendo.

## 2. Motivación

Cuando nos enfrentamos a la construcción del modelo conceptual del mundo del problema, en muchas ocasiones nos encontramos con el concepto de colección o grupo de cosas de la misma [clase](#). Por ejemplo, si retomamos el caso de estudio del empleado presentado en el nivel 1 y lo generalizamos a la administración de todos los empleados de la universidad, es claro que en alguna parte del diagrama de clases debe aparecer el concepto de grupo de empleados. Además, cuando planteemos la solución, tendremos que definir un [método](#) en alguna [clase](#) para añadir un nuevo elemento a ese grupo (ingresó un nuevo empleado a la universidad) o un [método](#) para buscar un empleado de la universidad (por ejemplo, quién es el empleado que tiene mayor salario). De manera similar, si retomamos el caso de estudio del nivel 2 sobre la tienda, lo natural es que una tienda manipule un número arbitrario de productos, y no sólo cuatro de ellos como se definió en el ejemplo. En ese caso, la tienda debe poder agregar un nuevo producto al grupo de los que ya vende, buscar un producto en su catálogo, etc.

En este capítulo vamos a introducir dos conceptos fundamentales de la programación:

1. Las estructuras contenedoras, que nos permiten manejar atributos cuyo valor corresponde a una secuencia de elementos.
2. Las instrucciones repetitivas, que son instrucciones que nos permiten manipular los elementos contenidos en dichas secuencias.

Además, en este nivel estudiaremos la manera de crear objetos y agregarlos a una contenedora, la manera de crear una [clase](#) completa en Java y la forma de leer la descripción de un conjunto de clases desarrolladas por otros, para ser capaces de utilizarlas en nuestros programas.

Vamos a trabajar sobre varios casos de estudio que iremos introduciendo a lo largo del nivel.

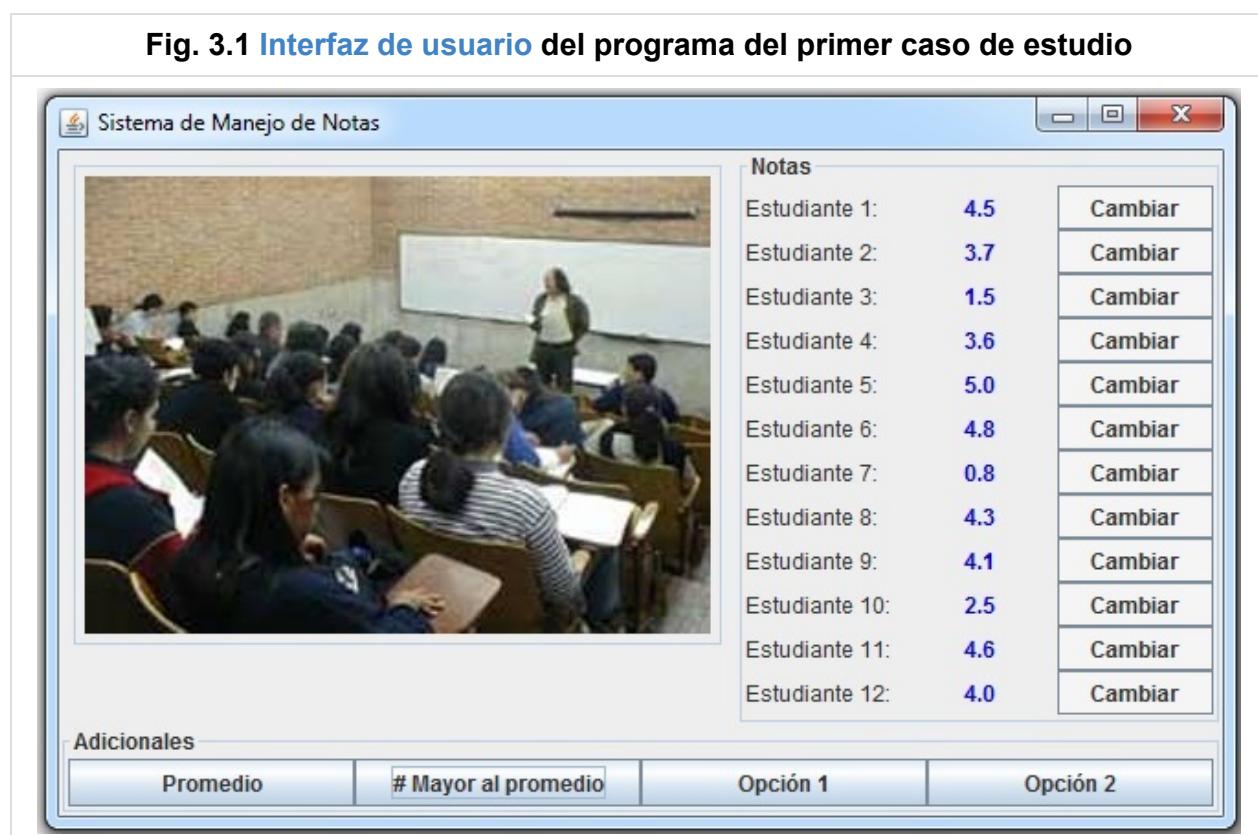
### 3. Caso de Estudio N° 1: Las Notas de un Curso

Considere el problema de administrar las calificaciones de los alumnos de un curso, en el cual hay doce estudiantes, de cada uno de los cuales se tiene la nota definitiva que obtuvo (un valor entre 0,0 y 5,0). Se quiere construir un programa que permita:

1. Cambiar la nota de un estudiante.
2. Calcular el promedio del curso.
3. Establecer el número de estudiantes que está por encima de dicho promedio.

En la [figura 3.1](#) aparece la **interfaz de usuario** que se quiere que tenga el programa.

**Fig. 3.1 Interfaz de usuario del programa del primer caso de estudio**



- En la [ventana](#) del programa aparece la nota de cada uno de los doce estudiantes del curso. La nota con la que comienzan es siempre cero.
- Con el respectivo botón es posible modificar la nota. Al oprimirlo, aparece una [ventana](#) de diálogo en la que se pide la nueva nota.
- En la parte de abajo de la [ventana](#) se encuentran los botones que implementan los requerimientos funcionales: calcular el promedio e indicar el número de estudiantes que están por encima de dicha nota.

## 3.1. Comprensión de los Requerimientos

### Requerimiento funcional 1

Nombre	R1 – Cambiar una nota.
Resumen	Permite cambiar la nota definitiva que tiene asignado un estudiante del curso.
Entradas	(1) El estudiante a quien se le quiere cambiar la nota. (2) La nueva nota del estudiante.
Resultado	Se le ha asignado al estudiante la nueva nota.

### Requerimiento funcional 2

Nombre	R2 – Calcular el promedio.
Resumen	Se quiere calcular el promedio del curso, utilizando la nota que tiene cada estudiante.
Entradas	Ninguna.
Resultado	Promedio de las notas de los doce estudiantes del curso.

### Requerimiento funcional 3

Nombre	R3 – Calcular el número de estudiantes por encima del promedio.
Resumen	Se quiere saber cuántos estudiantes tienen una nota superior a la nota promedio del curso.
Entradas	Ninguna.
Resultado	Número de estudiantes con nota mayor al promedio del curso.

## 3.2. Comprensión del Mundo del Problema

Dado el enunciado del problema, el modelo conceptual se puede definir con una [clase](#) llamada *Curso*, la cual tendría doce atributos de tipo double para representar las notas de cada uno de los estudiantes, tal como se muestra en la [figura 3.2](#).

**Fig. 3.2 Modelo conceptual de las calificaciones de los estudiantes**

## Curso

```
double nota1  
double nota2  
double nota3  
double nota4  
double nota5  
double nota6  
double nota7  
double nota8  
double nota9  
double nota10  
double nota11  
double nota12
```

Aunque este modelado es correcto, los métodos necesarios para resolver el problema resultarían excesivamente largos y dispendiosos. Cada [expresión](#) aritmética para calcular cualquier valor del curso tomaría muchas líneas de código. Además, imagine si en vez de 12 notas tuviéramos que manejar 50 ó 100. Terminaríamos con algoritmos imposibles de leer y de mantener. Necesitamos una manera mejor de hacer este modelado y ésta es la motivación de introducir el concepto de [estructura contenedora](#).

## 4. Contenedoras de Tamaño Fijo

Lo ideal, en el caso de estudio, sería tener un sólo [atributo](#) (llamado por ejemplo notas), en donde pudiéramos referirnos a uno de los valores individuales por un número que corresponda a su posición en el grupo (por ejemplo, la quinta nota). Ese tipo de atributos que son capaces de agrupar una secuencia de valores se denominan contenedoras y la idea se ilustra en la [figura 3.3](#)). Vale la pena aclarar que la sintaxis usada en la figura no corresponde a la sintaxis de UML, sino que solamente la usamos para ilustrar la idea de una [estructura contenedora](#).

**Fig. 3.3 Modelo conceptual de las calificaciones con una contenedora**

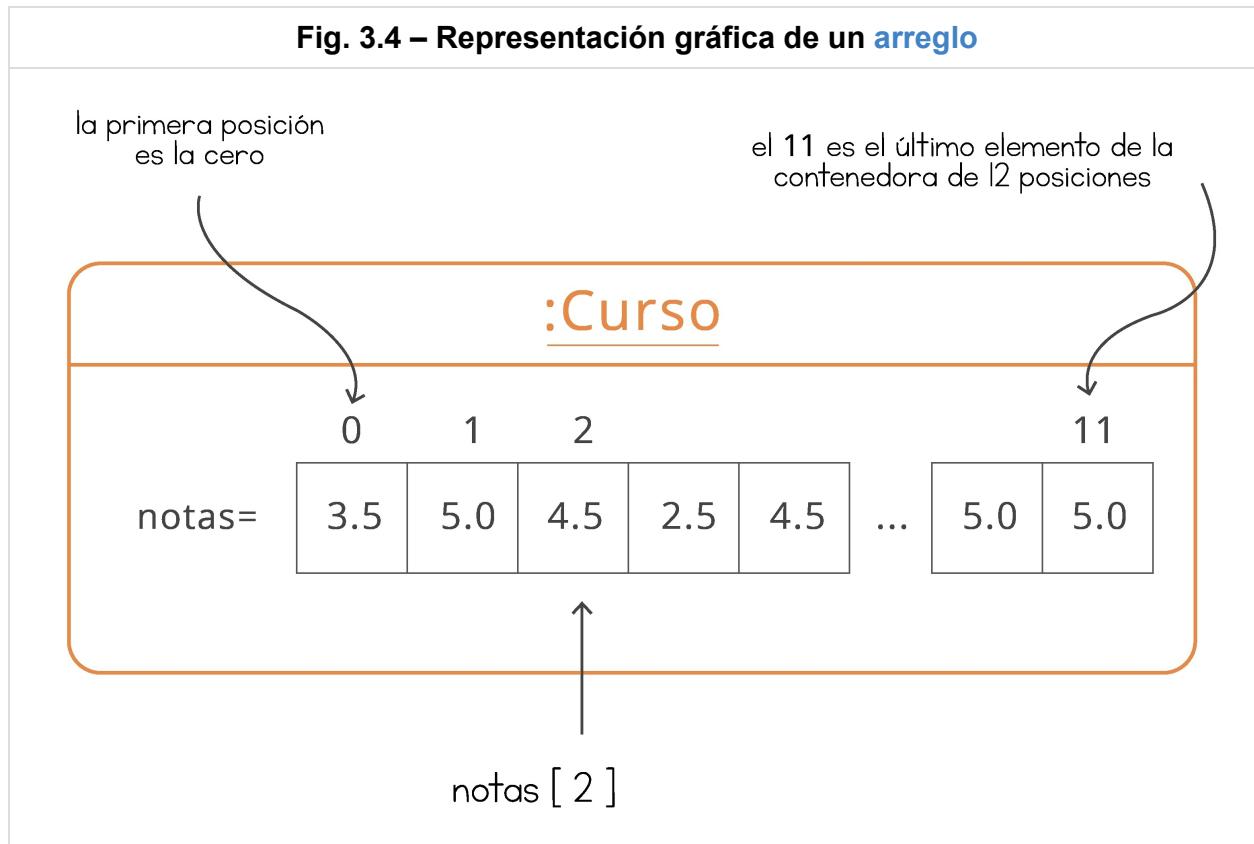
## Curso

**double nota =**

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

- En lugar de tener 12 atributos de tipo real, vamos a tener un sólo **atributo** llamado "notas" el cual contendrá en su interior las 12 notas que queremos representar.
- Cada uno de los elementos del **atributo** "notas" se puede referenciar utilizando la sintaxis `notas[x]`, donde `x` es el número del estudiante a quien corresponde la nota (comenzando en 0).
- Con esta representación podemos manejar de manera más simple y general el grupo de notas de los estudiantes.

Un [objeto](#) de la [clase](#) Curso se vería como aparece en la [figura 3.4](#). Allí se puede apreciar que las posiciones dentro de una contenedora se comienzan a numerar a partir del valor 0 y que los elementos individuales se refieren a través de su posición. Cada nota va en una posición distinta de la contenedora de tipo `double` llamada notas.



En las secciones que siguen veremos la manera de declarar (en UML y en Java) un [atributo](#) que corresponda a una contenedora, lo mismo que a manipular los valores allí incluidos.

## 4.1 Declaración de un Arreglo

En Java, las estructuras contenedoras de tamaño fijo se denominan arreglos (arrays en inglés), y se declaran como se muestra en el ejemplo 1. Los arreglos se utilizan para modelar una característica de una [clase](#) que corresponde a un grupo de elementos, de los cuales se conoce su número. Si no supiéramos, por ejemplo, el número de estudiantes del curso en el caso de estudio, deberíamos utilizar una [contenedora de tamaño variable](#), que es el tema de una sección posterior de este capítulo.

### Ejemplo 1

**Objetivo:** Mostrar la sintaxis usada en Java para declarar un [arreglo](#).

En este ejemplo se hace la declaración del **arreglo** de notas, como parte de la **clase** Curso del caso de estudio.

```
public class Curso
{
    //-----
    // Constantes
    //-----
    public final static int TOTAL_EST = 12;

    //-----
    // Atributos
    //-----
    private double[] notas;
    ...
}
```

- Es conveniente declarar el número de posiciones del **arreglo** como una **constante** (`TOTAL_EST`). Eso facilita realizar más tarde modificaciones al programa. Si en vez de 12 hay que manejar 15 estudiantes, bastaría con cambiar dicho valor.
- En el momento de declarar el **atributo** "notas", usamos la sintaxis "`[]`" para indicar que va a contener un grupo de valores.
- El tamaño del **arreglo** será determinado en el momento de la inicialización del **arreglo**, en el **método** constructor. Por ahora no hay que decir nada al respecto.
- En la declaración le decimos al **compilador** que todos los elementos del **arreglo** son de tipo `double`.
- Recuerde que los elementos de un **arreglo** se comienzan a referenciar a partir de la posición 0.

## 4.2 Inicialización de un Arreglo

Al igual que con cualquier otro **atributo** de una **clase**, es necesario inicializar los arreglos en el **método** constructor antes de poderlos utilizar. Para hacerlo se debe definir el tamaño del **arreglo**, o sea el número de elementos que va a contener. Esta inicialización es obligatoria, puesto que es en ese momento que le decimos al computador cuántos valores debe manejar en el **arreglo**, lo que corresponde al espacio en memoria que debe reservar. Veamos en el ejemplo 2 cómo se hace esto para el caso de estudio.

Si tratamos de acceder a un elemento de un **arreglo** que no ha sido inicializado, vamos a obtener el error de ejecución: `java.lang.NullPointerException`

### Ejemplo 2

**Objetivo:** Mostrar la manera de inicializar un [arreglo](#) en Java.

En este ejemplo mostramos, en el contexto del caso de estudio, la manera de inicializar el [arreglo](#) de notas dentro del constructor de la [clase](#) Curso.

```
public Curso( )
{
    notas = new double[ TOTAL_EST ] ;
}
```

- Se utiliza la instrucción `new` como con cualquier otro [objeto](#), pero se le especifica el número de valores que debe contener el [arreglo](#) (`TOTAL_EST`, que es una [constante](#) de valor 12).
- Esta construcción reserva el espacio para el [arreglo](#), pero el valor de cada uno de los elementos del [arreglo](#) sigue siendo indefinido. Esto lo arreglaremos más adelante.

El lenguaje Java provee un [operador](#) especial (`length`) para los arreglos, que permite consultar el número de elementos que éstos contienen. En el caso de estudio, la [expresión](#) `notas.length` debe dar el valor 12, independientemente de si los valores individuales ya han sido o no inicializados, puesto que en el [método](#) constructor de la [clase](#) se reservó dicho espacio de memoria.

## 4.3. Acceso a los Elementos del Arreglo

Un índice es un valor entero que nos sirve para indicar la posición de un elemento en un [arreglo](#). Los índices van desde 0 hasta el número de elementos menos 1. En el caso de estudio la primera nota tiene el índice 0 y la última, el índice 11. Para tomar o modificar el valor de un elemento particular de un [arreglo](#) necesitamos dar su índice, usando la sintaxis que aparece en el siguiente [método](#) de la [clase](#) Curso y que, en el caso general, se puede resumir como `<arreglo>[<índice>]`.

```
public void noHaceNadaUtil( double valor )
{
    int indice = 10;
    notas[ 0 ] = 3.5;
    if( valor < 2.5 && notas.length == TOTAL_EST )
    {
        notas[ indice ] = notas[ 0 ];
        notas[ 0 ] = valor + 1.0;
    }
    else
    {
        notas[ indice ] = notas[ 0 ] - valor;
    }
}
```

- Este [método](#) sólo lo utilizamos para ilustrar la sintaxis que se utiliza en Java para manipular los elementos de un [arreglo](#).
- Para asignar un valor a una casilla del [arreglo](#), usamos la sintaxis `notas[ x ] = valor` , donde x es el índice que nos indica una posición.
- Para obtener el valor de una casilla, usamos la misma sintaxis (`notas[ x ]` ).  
`notas.length` nos da el número de casillas del [arreglo](#).

De esta manera podemos asignar cualquier valor de tipo `double` a cualquiera de las casillas del [arreglo](#), o tomar el valor que allí se encuentra.

Cuando dentro de un [método](#) tratamos de acceder una casilla con un índice no válido (menor que 0 o mayor o igual que el número de casillas), obtenemos el error de ejecución: `java.lang.ArrayIndexOutOfBoundsException`

Es importante destacar que, hasta este momento, lo único que hemos ganado con la introducción de los arreglos es no tener que usar atributos individuales para representar una característica que incluye un grupo de elementos. Es más cómodo tener un sólo [atributo](#) con todos esos elementos en su interior. Las verdaderas ventajas de usar arreglos las veremos a continuación, al introducir las instrucciones repetitivas.

# 5. Instrucciones Repetitivas

## 5.1. Introducción

En muchos problemas notamos una regularidad que sugiere que su solución puede lograrse repitiendo un paso que vaya transformando gradualmente el estado del mundo modelado y acercándose a la solución. Instintivamente es lo que hacemos cuando subimos unas escaleras: repetimos el paso de subir un escalón hasta que llegamos al final. Otro ejemplo posible es si suponemos que tenemos en una hoja de papel una lista de palabras sin ningún orden y nos piden buscar si la palabra "casa" está en la lista. El [algoritmo](#) que seguimos para realizar esta tarea puede ser descrito de la siguiente manera:

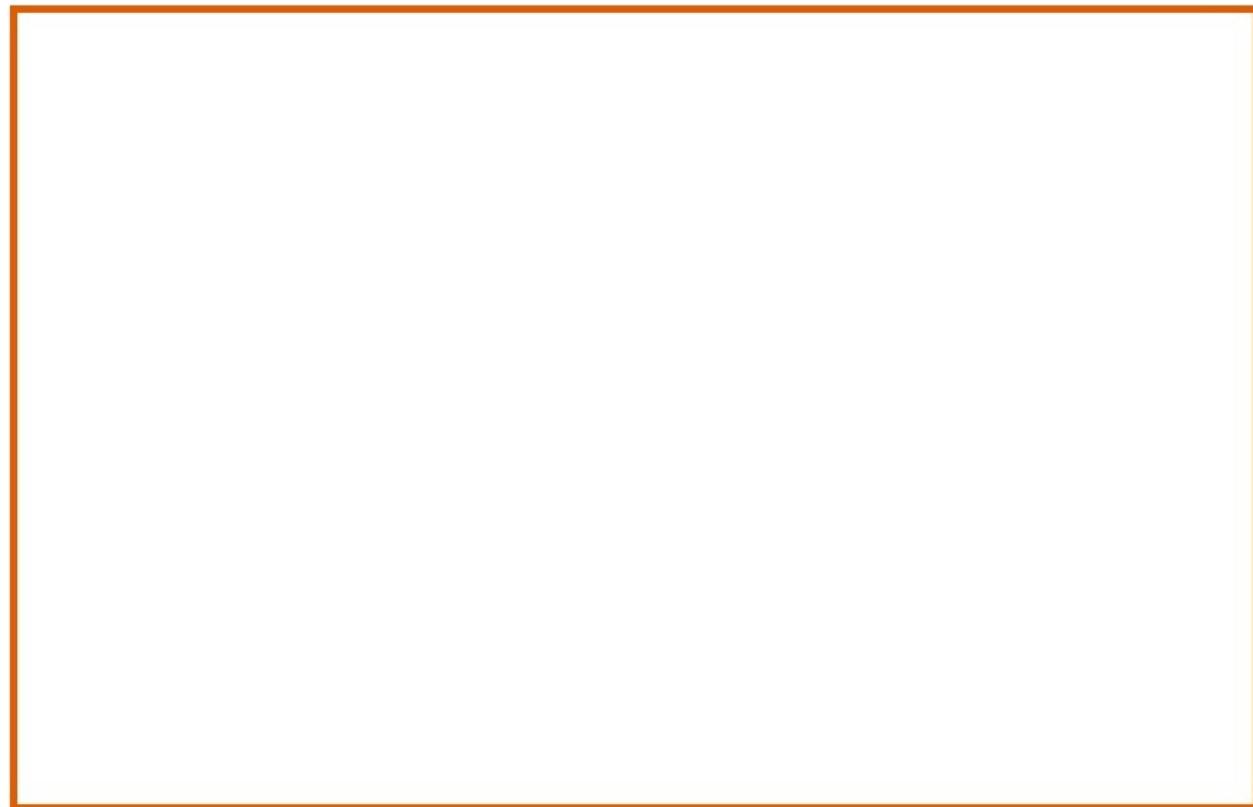
1. Verifique si la primera palabra es igual a "casa".
2. Si lo es, no busque más. Si no lo es, busque la segunda palabra.
3. Verifique si la segunda palabra es igual a "casa".
4. Si lo es, no busque más. Si no lo es, busque la tercera palabra.
5. Repita el procedimiento palabra por palabra, hasta que la encuentre o hasta que no haya más palabras para buscar.

### Tarea 1

**Objetivo:** Explicar el significado de la instrucción repetitiva y usarla para definir un [algoritmo](#) que resuelva un problema simple.

Suponga que en el ejemplo anterior, ya no queremos buscar una palabra sino contar el número total de letras que hay en todas las palabras de la hoja.

Escriba el [algoritmo](#) para resolver el problema:



## 5.2. Calcular el Promedio de las Notas

Para resolver el segundo requerimiento del caso de estudio (R2 - calcular el promedio de las notas), debemos calcular la suma de todas las notas del curso para luego dividirlo por el número de estudiantes. Esto se puede hacer con el [método](#) que se muestra a continuación:

```
public double promedio( )
{
    double suma = notas[ 0 ] + notas[ 1 ] + notas[ 2 ] +
                  notas[ 3 ] + notas[ 4 ] + notas[ 5 ] +
                  notas[ 6 ] + notas[ 7 ] + notas[ 8 ] +
                  notas[ 9 ] + notas[ 10 ] + notas[ 11 ];
    return suma / TOTAL_EST;
}
```

- Primero sumamos las notas de todos los estudiantes y guardamos el valor en la [variable](#) `suma`.
- El promedio corresponde a dividir dicho valor por el número de estudiantes, representado con la [constante](#) `TOTAL_EST`.

Si planteamos el problema de manera iterativa, podemos escribir el mismo [método](#) de la siguiente manera, en la cual, en cada paso, acumulamos el valor del siguiente elemento:

```

public double promedio( )
{
    double suma = 0.0;
    int indice = 0;
    suma += notas[ indice ];
    indice++;
    return suma / TOTAL_EST;
}

```

- Esta solución también calcula el promedio del curso, pero en lugar de hacer referencia directa a las doce casillas del **arreglo**, utiliza un índice que va desplazando desde 0 hasta 11.
- Por supuesto que es más clara la solución anterior, pero queremos utilizar este ejemplo para introducir las instrucciones iterativas, que expresan esta misma idea de "desplazar" un índice, pero usando una sintaxis mucho más compacta.
- Lo primero que debemos notar es que vamos a ejecutar 12 veces (`TOTAL_EST` veces para ser exactos) un grupo de instrucciones.
- Ese grupo de instrucciones es: `suma += notas[ indice ]; indice++ ;`
- Después de ejecutar 12 veces esas dos instrucciones, en la **variable** `suma` tendremos el valor total, listo para dividirlo por el número de estudiantes.
- El índice comienza teniendo el valor 0 y termina teniendo el valor 11. De esta manera, cada vez que hacemos referencia al elemento `notas[indice]`, estamos hablando de una casilla distinta del **arreglo**.

Allí repetimos 12 veces una pareja de instrucciones, una vez por cada elemento del [arreglo](#). Basta un poco de reflexión para ver que lo que necesitamos es poder decir que esas dos instrucciones se deben repetir tantas veces como notas haya en el [arreglo](#). Las instrucciones repetitivas nos permiten hacer eso de manera sencilla. En el siguiente [método](#) se ilustra el uso de la instrucción `while` para el mismo problema del cálculo del promedio.

```
public double promedio( )
{
    double suma = 0.0;
    int indice = 0;
    while( indice < TOTAL_EST )
    {
        suma += notas[ indice ]; indice++;
    }
    return suma / TOTAL_EST;
}
```

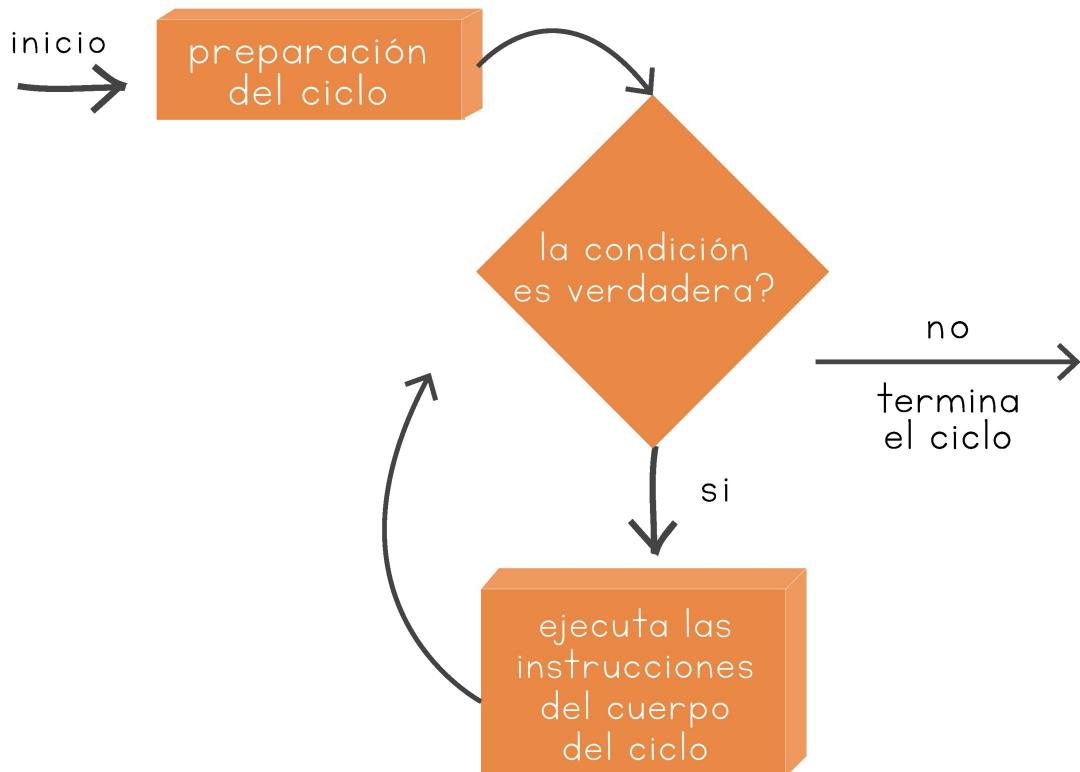
- La estructura del [método](#) sigue siendo la misma, con la única diferencia de que en lugar de repetir 12 veces la pareja de instrucciones, las incluimos dentro de la instrucción `while`, que se encarga de ejecutar repetidamente las instrucciones que tiene en su interior.
- La instrucción `while` sirve para decirle al computador que "mientras que" una [condición](#) se cumpla, siga ejecutando las instrucciones que están por dentro.
- La [condición](#) en el ejemplo es `indice < TOTAL_EST`, que equivale a decirle que "mientras que" el índice no llegue a 12, vuelva a ejecutar la pareja de instrucciones que tiene asociadas.

Ahora veremos las partes de las instrucciones repetitivas y su significado.

## 5.3. Componentes de una Instrucción Repetitiva

La [figura 3.5](#) ilustra la manera en que se ejecuta una instrucción repetitiva. Primero, y por una sola vez, se ejecutan las instrucciones que vamos a llamar de inicio o preparación del ciclo. Allí se le da el valor inicial al índice y a las variables en las que queremos acumular los valores durante el recorrido. Luego, se evalúa la [condición](#) del ciclo. Si es falsa, se ejecutan las instrucciones que se encuentran después del ciclo. Si es verdadera, se ejecutan las instrucciones del cuerpo del ciclo para finalmente volver a repetir el mismo proceso. Cada repetición, que incluye la evaluación de la [condición](#) y la ejecución del cuerpo del ciclo, recibe el nombre de [iteración](#) o [bucle](#).

Fig. 3.5 Ejecución de una instrucción repetitiva



Usualmente en un lenguaje de programación hay varias formas de escribir una instrucción repetitiva. En Java existen varias formas, pero en este libro sólo vamos a presentar dos de ellas: la instrucción `for` y la instrucción `while`.

### 5.3.1. Las Instrucciones `for` y `while`

Una instrucción repetitiva con la instrucción `while` se escribe de la siguiente manera:

```

<inicio>
while( <condición> )
{
  <cuerpo>
  <avance>
}
  
```

- Las instrucciones de preparación del ciclo van antes de la instrucción repetitiva.
- La **condición** que establece si se debe repetir de nuevo el ciclo va siempre entre paréntesis.
- El avance del ciclo es una parte opcional, en la cual se modifican los valores de algunos de los elementos que controlan la salida del ciclo (avanzar el índice con el que

se recorre un [arreglo](#) sería parte de esta sección).

Una instrucción repetitiva con la instrucción `for` se escribe de la siguiente manera:

```
<inicio1>
for( <inicio2>; <condición>; <avance> )
{
    <cuerpo>
}
```

- El inicio va separado en dos partes: en la primera, va la declaración y la inicialización de las variables que van a ser utilizadas después de terminado el ciclo (la [variable](#) `suma`, por ejemplo, en el [método](#) del promedio). En la segunda parte de la zona de inicio van las variables que serán utilizadas únicamente dentro de la instrucción repetitiva (la [variable](#) `índice`, por ejemplo, que sólo sirve para desplazarse recorriendo las casillas del [arreglo](#)).
- La segunda parte del inicio, lo mismo que el avance del ciclo, se escriben en el encabezado de la instrucción `for`.

## Ejemplo 3

**Objetivo:** Mostrar la manera de utilizar la instrucción iterativa `for`.

En este ejemplo se presenta una [implementación](#) del [método](#) que calcula el promedio de notas del caso de estudio, en la cual se utiliza la instrucción `for`.

```
public double promedio( )
{
    double suma = 0.0;
    for(int indice = 0; indice < TOTAL_EST; indice++)
    {
        suma += notas[ indice ];
    }
    return suma / TOTAL_EST;
}
```

- Puesto que la [variable](#) "`suma`" será utilizada por fuera del cuerpo del ciclo, es necesario declararla antes del `for`.
- La [variable](#) "`índice`" es interna al ciclo, por eso se declara dentro del encabezado.
- El avance del ciclo consiste en incrementar el valor del "`índice`".
- En este ejemplo, los corchetes del `for` son opcionales, porque sólo hay una instrucción dentro del cuerpo del ciclo.

Vamos a ver en más detalle cada una de las partes de la instrucción y las ilustraremos con algunos ejemplos.

### 5.3.2. El Inicio del Ciclo

El objetivo de las instrucciones de inicio o preparación del ciclo es asegurarnos de que vamos a empezar el proceso repetitivo con las variables de trabajo en los valores correctos. En nuestro caso, una [variable](#) de trabajo la utilizamos como índice para movernos por el [arreglo](#) y la otra para acumular la suma de las notas:

- La suma antes de empezar el ciclo debe ser cero: `double suma = 0.0;`
- El índice a partir del cual vamos a iterar debe ser cero: `int indice = 0;`

### 5.3.3. La Condición para Continuar

El objetivo de la [condición](#) del ciclo es identificar el caso en el cual se debe volver a hacer una nueva [iteración](#). Esta [condición](#) puede ser cualquier [expresión](#) lógica: si su evaluación da verdadero, significa que se deben ejecutar de nuevo las instrucciones del ciclo. Si es falsa, el ciclo termina y se continúa con la instrucción que sigue después de la instrucción repetitiva.

Típicamente, cuando se está recorriendo un [arreglo](#) con un índice, la [condición](#) del ciclo dice que se debe volver a iterar mientras el índice sea menor que el número total de elementos del [arreglo](#). Para indicar este número, se puede utilizar la [constante](#) que define su tamaño (`TOTAL_EST`) o el [operador](#) que calcula el número de elementos de un [arreglo](#) (`notas.length`).

Dado que los arreglos comienzan en 0, la [condición](#) del ciclo debe usar el [operador](#) `<` y el número de elementos del [arreglo](#). Son errores comunes comenzar los ciclos con el índice en 1 o tratar de terminar con la [condición](#) `indice <= notas.length`.

### 5.3.4. El Cuerpo del Ciclo

El cuerpo del ciclo contiene las instrucciones que se van a repetir en cada [iteración](#). Estas instrucciones indican:

- La manera de modificar algunas de las variables de trabajo para ir acercándose a la solución del problema. Por ejemplo, si el problema es encontrar la suma de las notas de todos los estudiantes del curso, con la instrucción `suma += notas[indice]` agregamos un nuevo valor al acumulado.
- La manera de modificar los elementos del [arreglo](#), a medida que el índice pasa por cada casilla. Por ejemplo, si queremos sumar una décima a todas las notas, lo

hacemos con la instrucción `notas[indice] += 0.1`.

### 5.3.5. El Avance del Ciclo

Cuando se recorre un [arreglo](#), es necesario mover el índice que indica la posición en la que estamos en un momento dado (`indice++`). En algún punto (en el avance o en el cuerpo) debe haber una instrucción que cambie el valor de la [condición](#) para que finalmente ésta sea falsa y se detenga así la ejecución de la instrucción iterativa. Si esto no sucede, el programa se quedará en un ciclo infinito.

Si construimos un ciclo en el que la [condición](#) nunca sea falsa (por ejemplo, si olvidamos escribir las instrucciones de avance del ciclo), el programa dará la sensación de que está bloqueado en algún lado, o podemos llegar al error:

`java.lang.OutOfMemoryError`

## Tarea 2

**Objetivo:** Practicar el desarrollo de métodos que tengan instrucciones repetitivas.

Para el caso de estudio de las notas de los estudiantes escriba los métodos de la [clase](#) Curso que resuelven los problemas planteados.

Calcular el número de estudiantes que sacaron una nota entre 3,0 y 5,0:

```
public int cuantosPasaron( )
{
    }

}
```

Calcular la mayor nota del curso:

```
public double mayorNota( )
{
}
}
```

Contar el número de estudiantes que sacaron una nota inferior a la del estudiante que está en la posición del **arreglo** que se entrega como **parámetro**. Suponga que el **parámetro posEst** tiene un valor comprendido entre **0** y **TOTAL\_EST - 1**.

```
public int cuantosPeoresQue( int posEst )
{
}
}
```

Aumentar el 5% todas las notas del curso, sin que ninguna de ellas sobrepase el valor 5,0:

```
public void hacerCurva( )
{
}
}
```

## 5.4. Patrones de Algoritmo para Instrucciones Repetitivas

Cuando trabajamos con estructuras contenedoras, las soluciones de muchos de los problemas que debemos resolver son similares y obedecen a ciertos esquemas ya conocidos (¿cuántas personas no habrán resuelto ya los mismos problemas que estamos aquí resolviendo?). En esta sección pretendemos identificar tres de los patrones que más se repiten en el momento de escribir un ciclo, y con los cuales se pueden resolver todos los problemas del caso de estudio planteados hasta ahora. Lo ideal sería que, al leer un problema que debemos resolver (el [método](#) que debemos escribir), pudiéramos identificar el patrón al cual corresponde y utilizar las guías que existen para resolverlo. Eso simplificaría enormemente la tarea de escribir los métodos que tienen ciclos.

Un patrón de [algoritmo](#) se puede ver como una solución genérica para un tipo de problemas, en la cual el programador sólo debe resolver los detalles particulares de su problema específico.

En esta sección vamos a introducir tres patrones que se diferencian por el tipo de recorrido que hacemos sobre la secuencia.

### 5.4.1. Patrón de Recorrido Total

En muchas ocasiones, para resolver un problema que involucra una secuencia, necesitamos recorrer todos los elementos que ésta contiene para lograr la solución. En el caso de estudio de las notas tenemos varios ejemplos de esto:

- Calcular la suma de todas las notas.
- Contar cuántos en el curso obtuvieron la nota 3,5.
- Contar cuántos estudiantes aprobaron el curso.
- Contar cuántos en el curso están por debajo del promedio (conociendo este valor).
- Aumentar en 10% todas las notas inferiores a 2,0.

¿Qué tienen en común los algoritmos que resuelven esos problemas? La respuesta es que la solución requiere siempre un recorrido de todo el [arreglo](#) para poder cumplir el objetivo que se está buscando: debemos pasar una vez por cada una de las casillas del [arreglo](#).

Esto significa:

1. Que el índice para iniciar el ciclo debe empezar en cero.
2. Que la [condición](#) para continuar es que el índice sea menor que la longitud del [arreglo](#).
3. Que el avance consiste en sumarle uno al índice.

Esa estructura que se repite en todos los algoritmos que necesitan un **recorrido total** es lo que denominamos el **esqueleto del patrón**, el cual se puede resumir con el siguiente fragmento de código:

```
for( int indice = 0; indice < arreglo.length; indice++ )
{
    <cuerpo>
}
```

- Es común que en lugar de la **variable** " `indice` " se utilice una **variable** llamada " `i` ". Esto hace el código un poco más compacto.
- En lugar del **operador** " `length` ", se puede utilizar también la **constante** que indica el número de elementos del **arreglo**.
- Los corchetes del " `for` " sólo son necesarios si el cuerpo tiene más de una instrucción.

Lo que cambia en cada caso es lo que se quiere hacer en el cuerpo del ciclo. Aquí hay dos variantes principales. En la primera, algunos de los elementos del **arreglo** van a ser modificados siguiendo una regla (por ejemplo, aumentar en 10% todas las notas inferiores a 2,0). Lo único que se hace en ese caso es reemplazar el del esqueleto por las instrucciones que hacen la modificación pedida a un elemento del **arreglo** (el que se encuentra en la posición `indice`). Esa variante se ilustra en el ejemplo 4.

## Ejemplo 4

**Objetivo:** Mostrar la primera variante del patrón de **recorrido total**.

En este ejemplo se presenta la **implementación** del **método** de la **clase** `Curso` que aumenta en 10% todas las notas inferiores a 2,0.

```
public void hacerCurva( )
{
    for( int i = 0; i < notas.length; i++ )
    {
        if( notas[ i ] < 2.0 )
            notas[ i ] = notas[ i ] * 1.1;
    }
}
```

- El esqueleto del patrón de **algoritmo** de **recorrido total** se copia dentro del cuerpo del **método**.
- Se reemplaza el cuerpo del patrón por la **instrucción condicional** que hace la modificación pedida.
- En el cuerpo se indica la modificación que debe sufrir el elemento que está siendo

referenciado por el índice con el que se recorre el [arreglo](#).

La segunda variante corresponde a calcular alguna [propiedad](#) sobre el conjunto de elementos del [arreglo](#) (por ejemplo, contar cuántos estudiantes aprobaron el curso). Esta variante implica cuatro decisiones que definen la manera de completar el esqueleto del patrón:

1. Cómo acumular la información que se va llevando a medida que avanza el ciclo.
2. Cómo inicializar dicha información.
3. Cuál es la [condición](#) para modificar dicho acumulado en el punto actual del ciclo.
4. Cómo modificar el acumulado.

En el ejemplo 5 se ilustra esta variante.

## Ejemplo 5

**Objetivo:** Mostrar la segunda variante del patrón de [recorrido total](#).

En este ejemplo se presenta la aplicación del patrón de [algoritmo de recorrido total](#), para el problema de contar el número de estudiantes que aprobaron el curso.

- ¿Cómo acumular información?

Vamos a utilizar una [variable](#) de tipo entero llamada `vanAprobando`, que va llevando durante el ciclo el número de estudiantes que aprobaron el curso.

- ¿Cómo inicializar el acumulado?

La [variable](#) `vanAprobando` se debe inicializar en 0, puesto que inicialmente no hemos encontrado todavía ningún estudiante que haya pasado el curso.

- ¿[Condición](#) para cambiar el acumulado?

Cuando `notas[ indice ]` sea mayor o igual a 3,0, porque quiere decir que hemos encontrado otro estudiante que pasó el curso.

- ¿Cómo modificar el acumulado?

El acumulado se modifica incrementándolo en 1.

```

public int cuantosAprobaron( )
{
    int vanAprobando = 0;

    for( int i = 0; i < notas.length; i++ )
    {
        if( notas[ i ] >= 3.0 ) vanAprobando++;
    }

    return vanAprobando;
}

```

- Las cuatro decisiones tomadas anteriormente van a definir la manera de completar el esqueleto del [algoritmo](#) definido por el patrón.
- Las decisiones 1 y 2 definen el inicio del ciclo.
- Las decisiones 3 y 4 ayudan a construir el cuerpo del mismo.

En resumen, si el problema planteado corresponde al patrón de [recorrido total](#), se debe identificar la variante y luego tomar las decisiones que definen la manera de completar el esqueleto.

## Tarea 3

**Objetivo:** Generar habilidad en el uso del patrón de [algoritmo de recorrido total](#).

Escriba los métodos de la [clase](#) Curso que resuelven los siguientes problemas, los cuales corresponden a las dos variantes del patrón de [algoritmo de recorrido total](#).

Escriba un [método](#) para modificar las notas de los estudiantes de la siguiente manera: a todos los que obtuvieron más de 4,0, les quita 0,5. A todos los que obtuvieron menos de 2,0, les aumenta 0,5. A todos los demás, les deja la nota sin modificar:

```

public void cambiarNotas( )
{
}

```

Escriba un [método](#) que retorne la menor nota del curso:

```
public double menorNota ( )  
{  
}  
}
```

Escriba un [método](#) que indique en qué rango hay más notas en el curso: rango 1 de 0,0 a 1,99, rango 2 de 2,0 a 3,49, rango 3 de 3,5 a 5,0:

```
public int rangoConMasNotas( )  
{  
}  
}
```

## 5.4.2. Patrón de Recorrido Parcial

En algunos problemas de manejo de secuencias no es necesario recorrer todos los elementos para lograr el objetivo propuesto. Piense en la solución de los siguientes problemas:

- Informar si algún estudiante obtuvo la nota 5,0.
- Buscar el primer estudiante con nota igual a cero.
- Indicar si más de 3 estudiantes perdieron el curso.
- Aumentar el 10% en la nota del primer estudiante que haya sacado más de 4,0.

En todos esos casos hacemos un recorrido del [arreglo](#), pero éste debe terminar tan pronto hayamos resuelto el problema. Por ejemplo, el [método](#) que informa si algún estudiante obtuvo cinco en la nota del curso debe salir del proceso iterativo tan pronto localice el

primer estudiante con esa nota. Sólo si no lo encuentra, va a llegar hasta el final de la secuencia.

Un **recorrido parcial** se caracteriza porque existe una **condición** que debemos verificar en cada **iteración** para saber si debemos detener el ciclo o volver a repetirlo.

En este patrón, debemos adaptar el esqueleto del patrón anterior para que tenga en cuenta la **condición** de salida, de la siguiente manera:

```
boolean termino = false;

for( int i = 0; i < arreglo.length && !termino; i++ )
{
    <cuerpo>

    if( <ya se cumplió el objetivo> )
        termino = true;
}
```

- Primero, declaramos una **variable** de tipo `boolean` para controlar la salida del ciclo, y la inicializamos en `false`.
- Segundo, en la **condición** del ciclo usamos el valor de la **variable** que acabamos de definir: si su valor es verdadero, no debe volver a iterar.
- Tercero, en algún punto del ciclo verificamos si el problema ya ha sido resuelto (si ya se cumplió el objetivo). Si ése es el caso, cambiamos el valor de la **variable** a verdadero.

```
for( int i = 0; i < arreglo.length && !<condición>; i++ )
{
    <cuerpo>
}
```

Este patrón de esqueleto es más simple que el anterior, pero sólo se debe usar si la **expresión** que indica que ya se cumplió el objetivo del ciclo es sencilla.

Cuando se aplica el patrón de **recorrido parcial**, el primer paso que se debe seguir es identificar la **condición** que indica que el problema ya fue resuelto. Con esa información se puede tomar la decisión de cuál esqueleto de **algoritmo** es mejor usar.

## Ejemplo 6

**Objetivo:** Mostrar el uso del patrón de **recorrido parcial** para resolver un problema.

En este ejemplo se presentan tres soluciones posibles al problema de decidir si algún estudiante obtuvo cinco en la nota del curso.

```

public boolean alguienConCinco( )
{
    boolean termino = false;

    for( int i = 0; i < notas.length && !termino; i++ )
    {
        if( notas[ i ] == 5.0 )
            termino = true;
    }

    return termino;
}

```

- La **condición** para no seguir iterando es que se encuentre una nota igual a 5,0 en la posición `i`.
- Al final del **método**, se retorna el valor de la **variable** "`termino`", que indica si el objetivo se cumplió. Esto funciona en este caso particular, porque dicha **variable** dice que en el **arreglo** se encontró una nota igual al valor buscado.

```

public boolean alguienConCinco( )
{
    int i = 0;
    while( i < notas.length && notas[ i ] != 5.0 )
    {
        i++;
    }
    return i < notas.length;
}

```

- Esta es la segunda solución posible, y evita el uso de la **variable** "`termino`", pero tiene varias consecuencias sobre la instrucción iterativa.
- En lugar de la instrucción `for` es más conveniente usar la instrucción `while`.
- La **condición** de continuación en el ciclo es que la  $i$ -ésima nota sea diferente de 5,0.
- El **método** debe retornar verdadero si la **variable** `i` no llegó hasta el final del **arreglo**, porque esto querría decir que encontró en dicha posición una nota igual a cinco.

```

public boolean alguienConCinco( )
{
    for( int i = 0; i < notas.length; i++ )
    {
        if( notas[ i ] == 5.0 )
            return true;
    }
    return false;
}

```

- Esta es la tercera solución posible. Si dentro del ciclo ya tenemos la respuesta del [método](#), en lugar de utilizar la [condición](#) para salir del ciclo, la usamos para salir de todo el [método](#).
- En la última instrucción retorna falso, porque si llega a ese punto quiere decir que no encontró ninguna nota con el valor buscado.
- Esta manera de salir de un ciclo, terminando la ejecución del [método](#) en el que éste se encuentra, se debe usar con algún cuidado, puesto que se puede producir código difícil de entender.

Hay muchas soluciones posibles para resolver un problema. Un patrón de [algoritmo](#) sólo es una guía que se debe adaptar al problema específico y al estilo preferido del programador.

Para el patrón de [recorrido parcial](#) aparecen las mismas dos variantes que para el patrón de [recorrido total](#) (ver ejemplo 7):

- En la primera variante se modifican los elementos del [arreglo](#) hasta que una [condición](#) se cumpla (por ejemplo, encontrar las tres primeras notas con 1,5 y asignarles 2,5). En ese caso, en el cuerpo del [método](#) va la modificación que hay que hacerle al elemento que se encuentra en el índice actual, pero se debe controlar que cuando haya llegado a la tercera modificación termine el ciclo.
- En la segunda variante, se deben tomar las mismas cuatro decisiones que se tomaban con el patrón de [recorrido total](#), respecto de la manera de acumular la información para calcular la respuesta que está buscando el [método](#).

## Ejemplo 7

**Objetivo:** Mostrar el uso del patrón de [recorrido parcial](#), en sus dos variantes.

En este ejemplo se presentan dos métodos de la [clase](#) Curso, en cada uno de los cuales se ilustra una de las variantes del patrón de [recorrido parcial](#).

Encontrar las primeras tres notas iguales a 1,5 y asignarles 2,5:

```

public void subirNotas( )
{
    int numNotas = 0;
    for( int i = 0; i < notas.length && numNotas < 3; i++ )
    {
        if( notas[ i ] == 1,5 )
        {
            numNotas++;
            notas[ i ] = 2,5;
        }
    }
}

```

Este **método** corresponde a la primera variante, porque hace una modificación de los elementos del **arreglo** hasta que una **condición** se cumpla. En el **método** del ejemplo, debemos contar el número de modificaciones que hacemos, para detenernos al llegar a la tercera.

Retornar la posición en la secuencia de la tercera nota con valor 5,0. Si dicha nota no aparece al menos 3 veces, el **método** debe retornar el valor -1:

```

public int tercerCinco( )
{
    int cuantosCincos = 0;
    int posicion = -1;
    for( int i = 0; i < notas.length && posicion == -1; i++ )
    {
        if( notas[ i ] == 5,0 )
        {
            cuantosCincos++;
            if( cuantosCincos == 3 )
            {
                posicion = i;
            }
        }
    }
    return posicion;
}

```

- ¿Cómo acumular información? En este caso necesitamos dos variables para acumular la información: la primera para llevar el número de notas iguales a 5,0 que han aparecido (`cuantosCincos`), la segunda para indicar la posición de la tercera nota 5,0 (`posicion`).
- ¿Cómo inicializar el acumulado? La **variable** `cuantosCincos` debe comenzar en 0. La **variable** `posicion` debe comenzar en menos 1.
- ¿**Condición** para cambiar el acumulado? Si la nota actual es 5,0 debemos cambiar

nuestro acumulado.

- ¿Cómo modificar el acumulado? Debe cambiar la variable `cuantoscincos`, incrementándose en 1. Si es el tercer 5,0 de la secuencia, la variable `posicion` debe cambiar su valor, tomando el valor del índice actual.

## Tarea 4

**Objetivo:** Generar habilidad en el uso del patrón de algoritmo de recorrido parcial.

Escriba los métodos de la clase Curso que resuelven los siguientes problemas, los cuales corresponden a las dos variantes del patrón de algoritmo de recorrido parcial.

Reemplazar todas las notas del curso por 0,0, hasta que aparezca la primera nota superior a 3,0.

```
public void notasACero( )
{
    }

}
```

Calcular el número mínimo de notas del curso necesarias para que la suma supere el valor 30, recorriéndolas desde la posición 0 en adelante. Si al sumar todas las notas no se llega a ese valor, el método debe retornar -1.

```
public int sumadasDanTreinta( )
{
    }

}
```

### 5.4.3. Patrón de Doble Recorrido

El último de los patrones que vamos a ver en este capítulo es el de [doble recorrido](#). Este patrón se utiliza como solución de aquellos problemas en los cuales, por cada elemento de la secuencia, se debe hacer un recorrido completo. Piense en el problema de encontrar la nota que aparece un mayor número de veces en el curso. La solución evidente es tomar la primera nota y hacer un recorrido completo del [arreglo](#) contando el número de veces que ésta vuelve a aparecer. Luego, haríamos lo mismo con los demás elementos del [arreglo](#) y escogeríamos al final aquélla que aparezca un mayor número de veces.

El esqueleto básico del [algoritmo](#) con el que se resuelven los problemas que siguen este patrón es el siguiente:

```
for( int indice1 = 0; indice1 < arreglo.length; indice1++ )
{
    for( int indice2 = 0; indice2 < arreglo.length; indice2++ )
    {
        <cuerpo del ciclo interno>
    }

    <cuerpo del ciclo externo>
}
```

- El ciclo de afuera está controlado por la [variable](#) " `indice1` ", mientras que el ciclo interno utiliza la [variable](#) " `indice2` ".
- Dentro del cuerpo del ciclo interno se puede hacer referencia a la [variable](#) " `indice1` ".

Las variantes y las decisiones son las mismas que identificamos en los patrones anteriores. La estrategia de solución consiste en considerar el problema como dos problemas independientes, y aplicar los patrones antes vistos, tal como se muestra en el ejemplo 8.

## Ejemplo 8

**Objetivo:** Mostrar el uso del patrón de [algoritmo](#) de [recorrido total](#).

En este ejemplo se muestra el [método](#) de la [clase](#) `Curso` que retorna la nota que aparece un mayor número de veces. Para escribirlo procederemos por etapas, las cuales se describen en la parte derecha.

```

public double masVecesAparece( )
{
    double notaMasVecesAparece = 0.0;

    for( int i = 0; i < notas.length; i++ )
    {
        for( int j = 0; j < notas.length; j++ )
        {
            //Por completar
        }
    }

    return notaMasVecesAparece;
}

```

- Primera etapa: armar la estructura del **método** a partir del esqueleto del patrón.
- Utilizamos las variables `i` y `j` para llevar los índices en cada uno de los ciclos.
- Decidimos que el resultado lo vamos a dejar en una **variable** llamada `notasMasVecesAparece`, la cual retornamos al final del **método**.
- Una vez construida la base del **método**, identificamos los dos problemas que debemos resolver en su interior: (1) contar el número de veces que aparece en el **arreglo** el valor que está en la casilla `i`; (2) encontrar el mayor valor entre los que son calculados por el primer problema.

```

public double masVecesAparece( )
{
    double notaMasVecesAparece = 0.0;

    for( int i = 0; i < notas.length; i++ )
    {
        double notaBuscada = notas[ i ]; int contador = 0;

        for( int j = 0; j < notas.length; j++ )
        {
            if( notas[ j ] == notaBuscada )
                contador++;
        }

        //Por completar
    }

    return notaMasVecesAparece;
}

```

- Segunda etapa: Resolvemos el primero de los problemas identificados, usando para

eso el ciclo interno.

- Para facilitar el trabajo, vamos a dejar en la **variable** `notaBuscada` la nota para la cual queremos contar el número de ocurrencias. Dicha **variable** la inicializamos con la nota de la casilla `i`.
- Usamos una segunda **variable** llamada `contador` para acumular allí el número de veces que aparezca el valor buscado dentro del **arreglo**. Dicho valor será incrementado cuando `notaBuscada == notas[ j ]`.
- Al final del ciclo, en la **variable** contador quedará el número de veces que el valor de la casilla `i` aparece en todo el **arreglo**.

```
public double masVecesAparece( )
{
    double notaMasVecesAparece = 0.0;

    int numeroVecesAparece = 0;

    for( int i = 0; i < notas.length; i++ )
    {
        double notaBuscada = notas[ i ];
        int contador = 0;

        for( int j = 0; j < notas.length; j++ )
        {
            if( notas[ j ] == notaBuscada )
                contador++;
        }

        if( contador > numeroVecesAparece )
        {
            notaMasVecesAparece = notaBuscada;
            numeroVecesAparece = contador;
        }
    }

    return notaMasVecesAparece;
}
```

- Tercera etapa: Usamos el ciclo externo para encontrar la nota que más veces aparece.
- Usamos para eso dos variables: `notaMasVecesAparece` que indica la nota que hasta el momento más veces aparece, y `numeroVecesAparece` para saber cuántas veces aparece dicha nota.
- Luego definimos el caso en el cual debemos cambiar el acumulado: si encontramos un valor que aparezca más veces que el que teníamos hasta el momento (`contador > numeroVecesAparece`) debemos actualizar los valores de nuestras variables.

En general, este patrón dice que para resolver un problema que implique un [doble recorrido](#), primero debemos identificar los dos problemas que queremos resolver (uno con cada ciclo) y, luego, debemos tratar de resolverlos independientemente, usando los patrones de [recorrido total](#) o parcial.

Si para resolver un problema se necesita un tercer ciclo anidado, debemos escribir métodos separados que ayuden a resolver cada problema individualmente, tal como se plantea en el nivel 4, porque la solución directa es muy compleja y propensa a errores.

## Tarea 5

**Objetivo:** Generar habilidad en el uso del patrón de [algoritmo de doble recorrido](#).

Escriba el [método](#) de la [clase](#) Curso que resuelve el siguiente problema, que corresponde al patrón de [algoritmo de doble recorrido](#).

Calcular una nota del curso (si hay varias que lo cumplan puede retornar cualquiera) tal que la mitad de las notas sean menores o iguales a ella.

```
public double notaMediana( )  
{  
  
}  
}
```

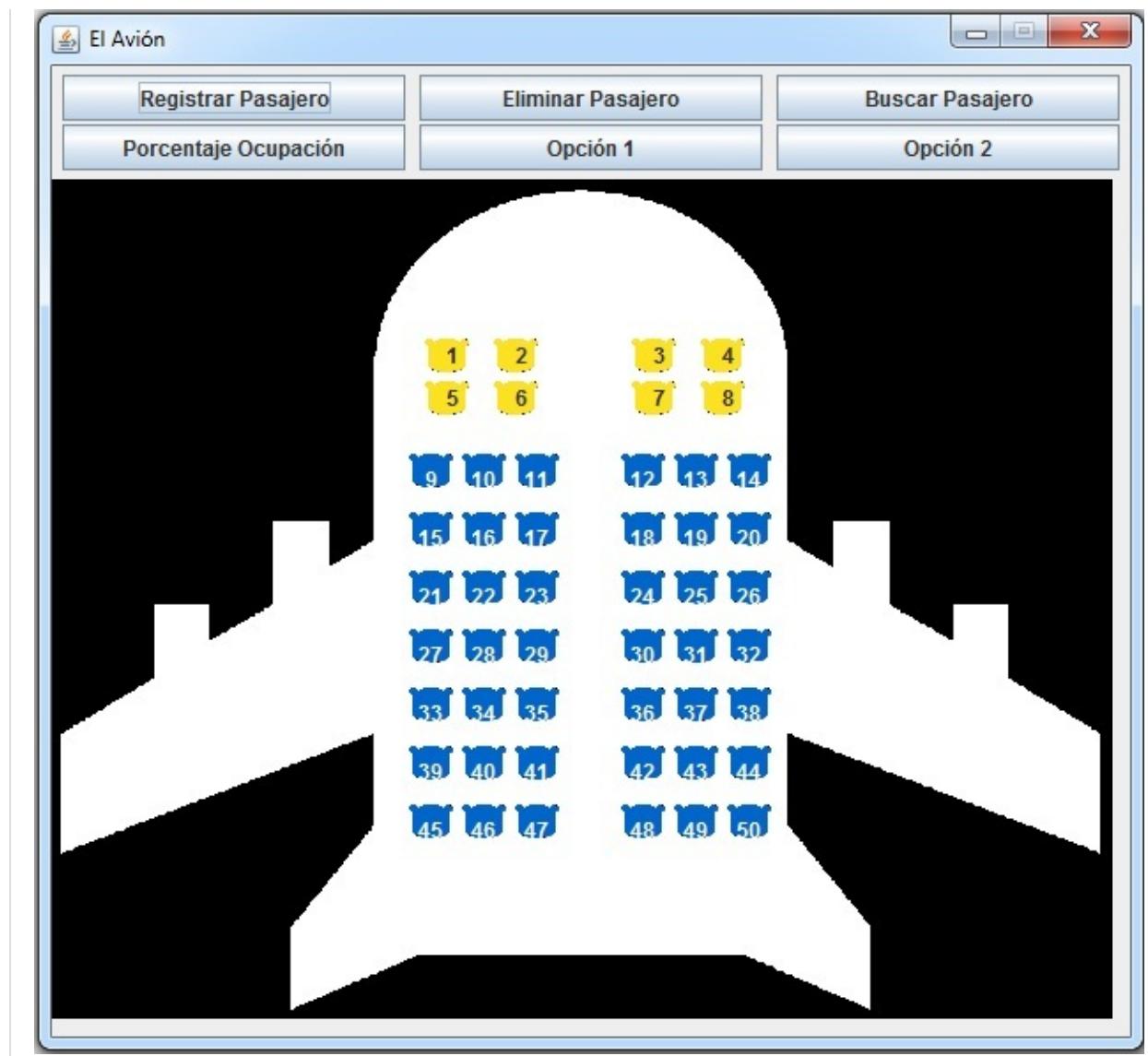
## 6. Caso de Estudio N° 2: Reservas en un Vuelo

Un cliente quiere que construyamos un programa para manejar las reservas de un vuelo. Se sabe que el avión tiene 50 sillas, de las cuales 8 son de [clase ejecutiva](#) y las demás de [clase económica](#). Las sillas ejecutivas se acomodan en filas de cuatro, separadas en el medio por el corredor. Las sillas económicas se acomodan en filas de seis, tres a cada lado del corredor.

Cuando un pasajero llega a solicitar una silla, indica sus datos personales y sus preferencias con respecto a la posición de la silla en el avión. Los datos del pasajero que le interesan a la aerolínea son el nombre y la cédula. Para dar la ubicación deseada, el pasajero indica la [clase](#) y la ubicación de la silla. Esta puede ser, en el caso de las ejecutivas, [ventana](#) y pasillo, y en el de las económicas, [ventana](#), pasillo y centro. La [asignación](#) de la silla en el avión se hace en orden de llegada, tomando en cuenta las preferencias anteriores y las disponibilidades.

La [interfaz de usuario](#) del programa a la que se llegó después de negociar con el cliente se muestra en la [figura 3.6](#).

**Fig. 3.6 Interfaz de usuario para el caso de estudio del avión**



- En la parte superior del avión aparecen las 8 sillas ejecutivas.
- En la parte inferior, aparecen las 42 sillas económicas, con un corredor en la mitad.
- Se ofrecen las distintas opciones del programa a través de los botones que se pueden observar en la parte superior de la [ventana](#).
- Cuando una silla está ocupada, ésta aparecerá indicada en el dibujo del avión con un color especial.
- Cada silla tiene asignado un número que es único. La silla 7, por ejemplo, está en primera [clase](#), en el corredor de la segunda fila.

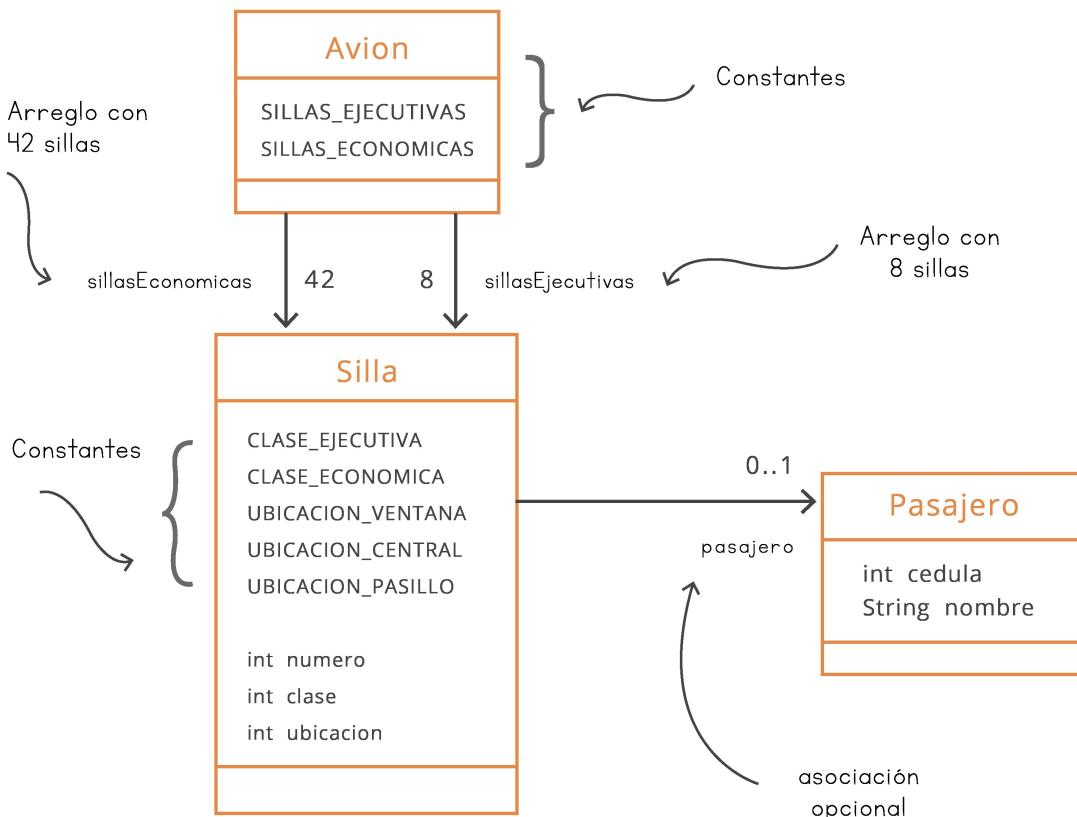
## 6.1. Comprensión de los Requerimientos

Nombre	R1 - Asignar una silla a un pasajero.
Resumen	Se quiere asignar una silla según las preferencias del pasajero. Estas son la <a href="#">clase</a> (ejecutiva o económica) y la ubicación ( <a href="#">ventana</a> , centro o pasillo). En la <a href="#">asignación</a> se deben registrar los datos del pasajero.
Entradas	(1) nombre del pasajero, (2) cédula del pasajero, (3) <a href="#">clase</a> de la silla que desea, (4) ubicación de la silla que desea.
Resultados	Si existe una silla con las características de <a href="#">clase</a> y ubicación solicitadas por el pasajero, ésta queda asignada a dicho pasajero.

## 6.2. Comprensión del Mundo del Problema

Podemos identificar tres entidades distintas en el mundo: avión, silla y pasajero. Lo cual nos lleva al diagrama de clases que se muestra en la [figura 3.7](#).

**Fig. 3.7 Diagrama de clases para el caso de estudio del avión**



En este diagrama se puede leer lo siguiente:

- Una silla puede ser ejecutiva o económica (dos constantes definidas en la [clase](#) Silla), puede estar localizada en pasillo, corredor o centro (tres constantes de la [clase](#) Silla), y

tiene un identificador único que es un valor numérico.

- Entre Silla y Pasajero hay una **asociación** opcional (0..1). Si la **asociación** está presente se interpreta como que la silla está ocupada y se conoce el pasajero que allí se encuentra. Si no está presente (vale null) se interpreta como que la silla está disponible.
- Un pasajero se identifica con la cédula y tiene un nombre.
- Un avión tiene 8 sillas ejecutivas (**constante** SILLAS\_EJECUTIVAS de la **clase** Avion) y 42 sillas económicas (**constante** SILLAS\_ECONOMICAS de la **clase** Avion). Fíjese cómo se expresa la cardinalidad de una **asociación** en UML.

## 6.3. Diseño de la Solución

Vamos a dividir el proyecto en 3 paquetes, siguiendo la **arquitectura** planteada en el primer nivel del libro. Los paquetes son:

```
uniandes.cupi2.avion.interfaz
uniandes.cupi2.avion.test
uniandes.cupi2.avion.mundo
```

La principal decisión de **diseño** del programa se refiere a la manera de representar el grupo de sillas del avión. Para esto vamos a manejar dos arreglos de objetos. Uno con 8 posiciones que tendrá los objetos de la **clase** Silla que representan las sillas de la **clase** ejecutiva, y otro **arreglo** de 42 posiciones con los objetos para representar las sillas económicas.

En las secciones que siguen presentaremos las distintas clases del modelo del mundo que constituyen la solución. Comenzamos por la **clase** más sencilla (la **clase** Pasajero) y terminamos por la **clase** que tiene la **responsabilidad** de manejar los grupos de atributos (la **clase** Avion), en donde tendremos la oportunidad de utilizar los patrones de **algoritmo** vistos en las secciones anteriores.

## 6.4. La Clase Pasajero

Tarea 6

**Objetivo:** Hacer la declaración en Java de la **clase** Pasajero.

Complete la declaración de la **clase** Pasajero, incluyendo sus atributos, el constructor y los métodos que retornan la cédula y el nombre. Puede guiarse por el diagrama de clases que aparece en la **figura 3.7**.

```
public class Pasajero
{
    //-----
    // Atributos
    //-----

    //-----
    // Constructor
    //-----
    public Pasajero( int unaCedula, String unNombre )
    {

    }

    //-----
    // Métodos
    //-----
    public int darCedula( )
    {

    }

    }
    public String darNombre( )
    {
```

```
    }  
}
```

## 6.5. La Clase Silla

### Tarea 7

**Objetivo:** Completar la declaración de la [clase](#) Silla.

Complete las declaraciones de los atributos y las constantes de la [clase](#) Silla y desarrolle los métodos que se le piden para esta [clase](#).

```
public class Silla  
{  
    //-----  
    // Constantes  
    //-----  
    public final static int CLASE_EJECUTIVA = 1;  
    public final static int CLASE_ECONOMICA = 2;  
    public final static int VENTANA = 1;  
  
    ...  
}
```

- Se declaran dos constantes para [atributo clase](#) de la silla CLASE\_ECONOMICA).
- Se declaran tres constantes para representar las tres posiciones posibles de una silla ([VENTANA](#),CENTRAL, PASILLO).

```
public class Silla
{
    ...
    //-----
    // Atributos
    //-----
    private int numero;

    private int clase;
    private int ubicacion;
    private Pasajero pasajero;

    ...
}
```

- Se declaran en la **clase** cuatro atributos: (1) el número de la silla, (2) la **clase** de la silla, (3) su ubicación y (4) el pasajero que opcionalmente puede ocupar la silla.
- El **atributo** " `pasajero` " debe tener el valor `null` si no hay ningún pasajero asignado a la silla.

```
public Silla( int num, int clas, int ubica )
{
    numero = num;
    clase = clas;
    ubicacion = ubica;
    pasajero = null;
}
```

- En el constructor se inicializan los atributos a partir de los valores que se reciben como **parámetro**.
- Se inicializa el **atributo** `pasajero` en `null`, para indicar que la silla se encuentra vacía.

```
public class Silla
{
    ...
    public void asignarPasajero( Pasajero pas )
    {
        ...
    }
    ...
}
```

- Asigna la silla al pasajero "pas".

```
public class Silla
{
    ...
    public void desasignarSilla ( )
    {
        ...
    }
    ...
}
```

- Quita al pasajero que se encuentra en la silla, dejándola desocupada.

```
public class Silla
{
    ...
    public boolean sillaAsignada( )
    {
        ...
    }
    ...
}
```

- Informa si la silla está ocupada.

```
public class Silla
{
    ...
    public int darNumero( )
    {
        ...
    }
    ...
}
```

- Retorna el número de la silla.

```
public class Silla
{
    ...
    public int darClase( )
    {
        ...
    }
    ...
}
```

- Retorna la **clase** de la silla.

```
public class Silla
{
    ...
    public int darUbicacion( )
    {
        ...
    }
    ...
}
```

- Retorna la ubicación de la silla.

```
public class Silla
{
    ...
    public Pasajero darPasajero( )
    {
        ...
    }
    ...
}
```

- Retorna el pasajero de la silla.

## 6.6. La Clase Avion

## Ejemplo 9

**Objetivo:** Mostrar las declaraciones y el constructor de la [clase](#) Avion.

En este ejemplo se presentan las declaraciones de los atributos y las constantes de la [clase](#) Avion, lo mismo que su [método](#) constructor.

```
public class Avion
{
    //-----
    // Constantes
    //-----
    public final static int SILLAS_EJECUTIVAS = 8;
    public final static int SILLAS_ECONOMICAS = 42;
    ...
}
```

- Con dos constantes representamos el número de sillas de cada una de las clases.

```
public class Avion
{
    ...
    //-----
    // Atributos
    //-----
    private Silla[] sillasEjecutivas;
    private Silla[] sillasEconomicas;

    ...
}
```

- La [clase](#) Avion tiene dos contenedoras de tamaño fijo de sillas: una, de 42 posiciones, con las sillas de [clase](#) económica, y otra, de 8 posiciones, con las sillas de [clase](#) ejecutiva.
- Se declaran los dos arreglos, utilizando la misma sintaxis que utilizamos en el caso de las notas del curso.
- La única diferencia es que en lugar de contener valores de tipo simple, van a contener objetos de la [clase](#) Silla.

A continuación aparece un fragmento del constructor de la [clase](#). En las primeras dos instrucciones del constructor, creamos los arreglos, informando el número de casillas que deben contener. Para eso usamos las constantes definidas en la [clase](#).

Después de haber reservado el espacio para los dos arreglos, procedemos a crear los objetos que representan cada una de las sillas del avión y los vamos poniendo en la respectiva casilla.

Esta inicialización se podría haber hecho con varios ciclos, pero el código resultaría un poco difícil de explicar.

```
public Avion( )
{
    sillasEjecutivas = new Silla[ SILLAS_EJECUTIVAS ];
    sillasEconomicas = new Silla[ SILLAS_ECONOMICAS ];

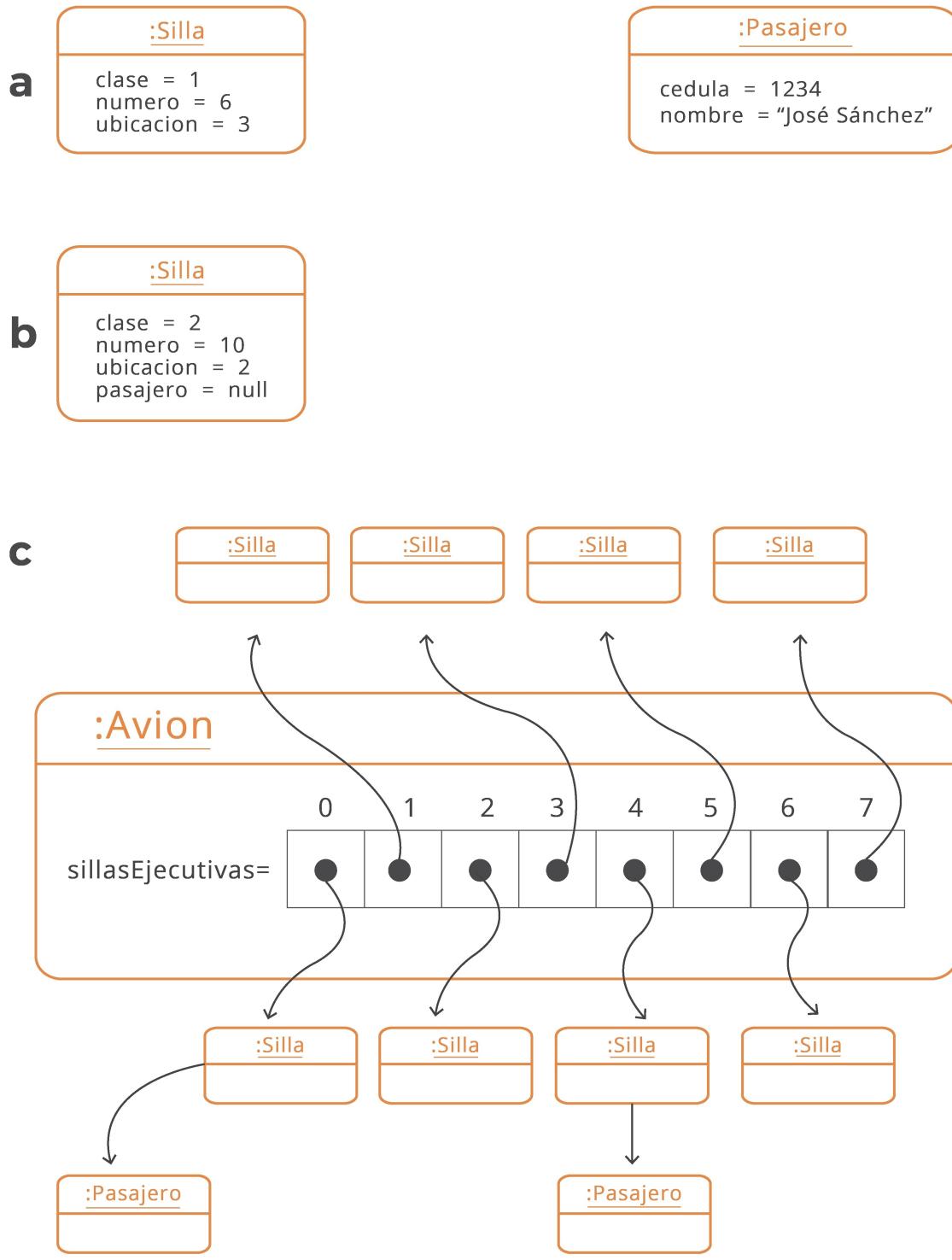
    // Creación de las sillas de clase ejecutiva
    sillasEjecutivas[ 0 ] = new Silla( 1, Silla.CLASE_EJECUTIVA, Silla.VENTANA );
    sillasEjecutivas[ 1 ] = new Silla( 2, Silla.CLASE_EJECUTIVA, Silla.PASILLO );
    sillasEjecutivas[ 2 ] = new Silla( 3, Silla.CLASE_EJECUTIVA, Silla.PASILLO );
    sillasEjecutivas[ 3 ] = new Silla( 4, Silla.CLASE_EJECUTIVA, Silla.VENTANA );
    sillasEjecutivas[ 4 ] = new Silla( 5, Silla.CLASE_EJECUTIVA, Silla.VENTANA );
    sillasEjecutivas[ 5 ] = new Silla( 6, Silla.CLASE_EJECUTIVA, Silla.PASILLO );
    sillasEjecutivas[ 6 ] = new Silla( 7, Silla.CLASE_EJECUTIVA, Silla.PASILLO );
    sillasEjecutivas[ 7 ] = new Silla( 8, Silla.CLASE_EJECUTIVA, Silla.VENTANA );

    // Creación de las sillas de clase económica
    sillasEconomicas[ 0 ] = new Silla( 9, Silla.CLASE_ECONOMICA, Silla.VENTANA );
    sillasEconomicas[ 1 ] = new Silla( 10, Silla.CLASE_ECONOMICA, Silla.CENTRAL );
    sillasEconomicas[ 2 ] = new Silla( 11, Silla.CLASE_ECONOMICA, Silla.PASILLO );
    ...
}
```

Ya con las declaraciones hechas y con el constructor implementado, estamos listos para comenzar a desarrollar los distintos métodos de la [clase](#). Pero antes de empezar, queremos hablar un poco de las diferencias que existen entre un [arreglo](#) de valores de tipo simple (como el del caso de estudio de las notas) y un [arreglo](#) de objetos (como el del caso del avión).

Para empezar, en la [figura 3.8a](#) se muestra una instancia de la [clase](#) Silla ocupada por un pasajero. En la [figura 3.8b](#) se muestra un [objeto](#) de la [clase](#) Silla que se encuentra vacía. En la [figura 3.8c](#) se ilustra un posible contenido del [arreglo](#) de sillas ejecutivas (usando un [diagrama de objetos](#)).

**Fig. 3.8 Ejemplo del contenido del arreglo de sillas ejecutivas**



- **Figura 3.8a:** en la silla de primera `clase` número 6, situada en el corredor, está sentado el Sr. José Sánchez con cédula No. 1234.
- **Figura 3.8b:** la silla de `clase` económica número 10, situada en el centro, está desocupada.
- **Figura 3.8c:** cada casilla del `arreglo` tiene un `objeto` de la `clase` `Silla` (incluso si la silla está desocupada).

- Las sillas ocupadas tienen una **asociación** con el **objeto** que representa al pasajero que la ocupa. \* En los arreglos de objetos se almacenan referencias a los objetos, en lugar de los objetos mismos.
- Con la sintaxis `sillasEjecutivas[x]` podemos hacer referencia al **objeto** de la **clase** Silla que se encuentra en la casilla x.
- Si queremos llegar hasta el pasajero que se encuentra en alguna parte del avión, debemos siempre pasar por la silla que ocupa. No hay otra manera de "navegar" hasta él.

Ya teniendo una visualización del **diagrama de objetos** del caso de estudio, es más fácil contestar las siguientes preguntas:

¿Cómo se llama un <b>método</b> de un <b>objeto</b> que está en un <b>arreglo</b> ?	Por ejemplo, dentro de la <b>clase Avion</b> , para preguntar si la silla que está en la posición 0 del <b>arreglo</b> de sillas ejecutivas está ocupada, se utiliza la sintaxis: <code>sillasEjecutivas[0].sillaAsignada()</code> . Esta sintaxis es sólo una extensión de la sintaxis que ya veníamos utilizando. Lo único que se debe tener en cuenta es que cada vez que hacemos referencia a una casilla, estamos hablando de un <b>objeto</b> , más que de un valor simple.
¿Los objetos que están en un <b>arreglo</b> se pueden guardar en una <b>variable</b> ?	Tanto las variables como las casillas de los arreglos guardan únicamente referencias a los objetos. Si se hace la siguiente <b>asignación</b> : <code>Silla sillaTemporal = sillasEjecutivas[0];</code> tanto la variable <code>sillaTemporal</code> como la casilla 0 del <b>arreglo</b> estarán haciendo referencia al mismo <b>objeto</b> . Debe quedar claro que el <b>objeto</b> no se duplica, sino que ambos nombres hacen referencia al mismo <b>objeto</b> .
¿Qué pasa con el <b>objeto</b> que está siendo referenciado desde una casilla si asigno <code>null</code> a esa posición del <b>arreglo</b> ?	Si guardó una referencia a ese <b>objeto</b> en algún otro lado, puede seguir usando el <b>objeto</b> a través de dicha referencia. Si no guardó una referencia en ningún lado, el recolector de basura de Java detecta que ya no lo está usando y recupera la memoria que el <b>objeto</b> estaba utilizando. ¡Adiós <b>objeto</b> !

## Ejemplo 10

**Objetivo:** Mostrar la sintaxis que se usa para manipular arreglos de objetos.

En este ejemplo se muestra el código de un **método** de la **clase Avion** que permite eliminar todas las reservas del avión. No forma parte de los requerimientos funcionales, pero nos va a permitir mostrar una aplicación del patrón de **recorrido total**.

```

public void eliminarReservas( )
{
    for( int i = 0; i < SILLAS_EJECUTIVAS; i++ )
    {
        sillasEjecutivas[ i ].desasignarSilla( );
    }

    for( int i = 0; indice < SILLAS_ECONOMICAS; i++ )
    {
        sillasEconomicas[ i ].desasignarSilla( );
    }
}

```

- Este **método** elimina todas las reservas que hay en el avión.
- Note que podemos utilizar la misma **variable** como índice en los dos ciclos. La razón es que en la instrucción `for`, al terminar de ejecutar el ciclo, se destruyen las variables declaradas dentro de él y, por esta razón, podemos volver a utilizar el mismo nombre para la **variable** del segundo ciclo.
- El **método** utiliza el patrón de **recorrido total** dos veces, una por cada uno de los arreglos del avión.

Ya vimos toda la teoría concerniente al manejo de los arreglos (estructuras contenedoras de tamaño fijo). Lo que sigue es aplicar los patrones de **algoritmo** que vimos unas secciones atrás, para implementar los métodos de la **clase** Avion.

## Tarea 8

**Objetivo:** Desarrollar los métodos de la **clase** Avión que nos permitan implementar los requerimientos funcionales del caso de estudio.

Para cada uno de los problemas que se plantean a continuación, escriba el **método** que lo resuelve. No olvide identificar primero el patrón de **algoritmo** que se necesita y usar las guías que se dieron en secciones anteriores.

Calcular el número de sillas ejecutivas ocupadas en el avión:

```
public int contarSillasEjecutivasOcupadas( )
{
}

}
```

Localizar la silla en la que se encuentra el pasajero identificado con la cédula que se entrega como **parámetro**. Si no hay ningún pasajero en **clase ejecutiva** con esa cédula, el **método** retorna `null`.

```
public Silla buscarPasajeroEjecutivo( int cedula )
{
}

}
```

Localizar una silla económica disponible, en una localización dada (**ventana**, centro o pasillo). Si no existe ninguna, el **método** retorna `null`:

```
public Silla buscarSillaEconomicaLibre( int ubicacion )
{
}

}
```

Asignar al pasajero que se recibe como **parámetro** una silla en **clase** económica que esté libre (en la ubicación pedida). Si el proceso tiene éxito, el **método** retorna verdadero. En caso contrario, retorna falso:

```
public boolean asignarSillaEconomica( int ubicacion, Pasajero pasajero )
{
}
}
```

Anular la reserva en **clase** ejecutiva que tenía el pasajero con la cédula dada. Retorna verdadero si el proceso tiene éxito:

```
public boolean anularReservaEjecutivo( int cedula )
{
}
}
```

Contar el número de puestos disponibles en una **ventana**, en la zona económica del avión:

```
public int contarVentanasEconomica( )
{
}
}
```

Informar si en la zona económica del avión hay dos personas que se llamen igual. Patrón de doble recorrido:

```
public boolean hayDosHomonimosEconomica( )  
{  
  
}  
}
```

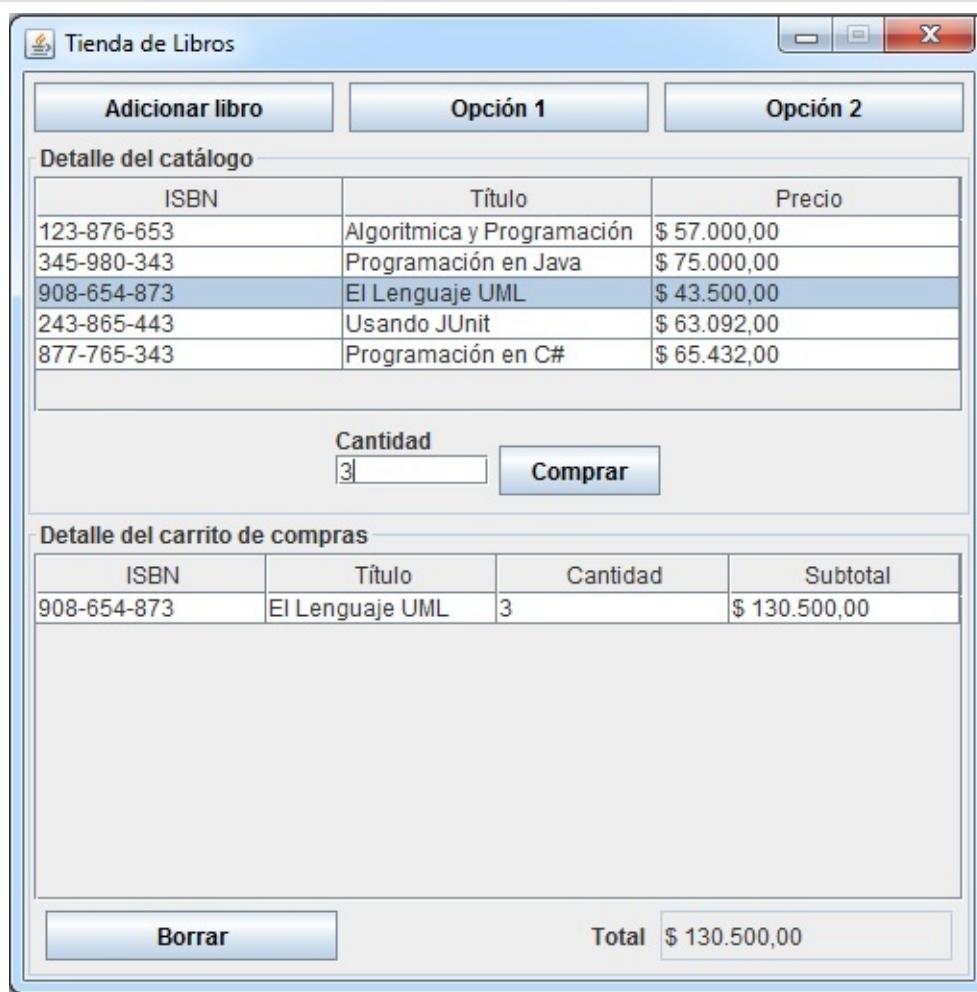
## 7. Caso de Estudio N° 3: Tienda de Libros

Se quiere construir una aplicación para una tienda virtual de libros. La tienda tiene un catálogo o colección de libros que ofrece para la venta. Los libros tienen un ISBN que los identifica de manera única, un título y un precio con el que se venden.

Cuando un cliente llega a la tienda virtual a comprar libros, utiliza un carrito de compras. En el carrito de compras va adicionando los libros que quiere comprar. El cliente puede llevar más de un ejemplar de cada libro. Al revisar la cuenta, el cliente debe poder ver el subtotal de cada libro según la cantidad de ejemplares que lleve de él, además del total de la compra, que es igual a la suma de los subtotales.

En la [figura 3.9](#) aparece la **interfaz de usuario** que se tiene prevista para el programa que se va a construir.

**Fig. 3.9 Interfaz de usuario de la tienda de libros**



- La interfaz está dividida en dos zonas: una para que el usuario pueda ver el catálogo de libros disponibles en la tienda (donde también puede adicionar libros), y una zona

para manejar el carrito de compras del cliente (donde puede comprar).

- En la imagen del ejemplo, aparecen cinco libros en el catálogo. Para adicionar libros a la tienda, se usa el botón que aparece en la parte superior izquierda.
- En la zona de abajo aparece la información de los libros que el cliente lleva en su carrito de compras. En la imagen del ejemplo, el cliente lleva en su carrito 3 ejemplares del libro titulado "El Lenguaje UML". El monto total de la compra es hasta el momento de \$130.500.
- Con el botón Comprar se pueden añadir libros al carrito de compras. Se debe dar la cantidad de ejemplares y seleccionar del catálogo uno de los libros.
- Con el botón Borrar se elimina del carrito de compras el libro que esté seleccionado.

## 7.1. Comprensión de los Requerimientos

Los requerimientos funcionales de este caso de estudio son tres:

1. Adicionar un nuevo libro al catálogo.
2. Agregar un libro al carro de compras del cliente.
3. Retirar un libro del carro de compras.

### Tarea 9

**Objetivo:** Entender el problema del caso de estudio.

Lea detenidamente el enunciado del caso de estudio y complete la documentación de los tres requerimientos funcionales.

#### Requerimiento funcional 1

Nombre	R1 - Adicionar un nuevo libro al catálogo.
Resumen	Se quiere adicionar un nuevo libro al catálogo para vender en la tienda.
Entradas	(1) título del libro, (2) ISBN del libro, (3) precio del libro.
Resultado	El catálogo ha sido actualizado y contiene el nuevo libro.

#### Requerimiento funcional 2

	
Nombre	
Resumen	
Entradas	
Resultado	

**Requerimiento funcional 3**

<b>Nombre</b>	
Resumen	
Entradas	
Resultado	

## 7.2. Comprensión del Mundo del Problema

En el mundo del problema podemos identificar cuatro entidades (ver figura 3.10):

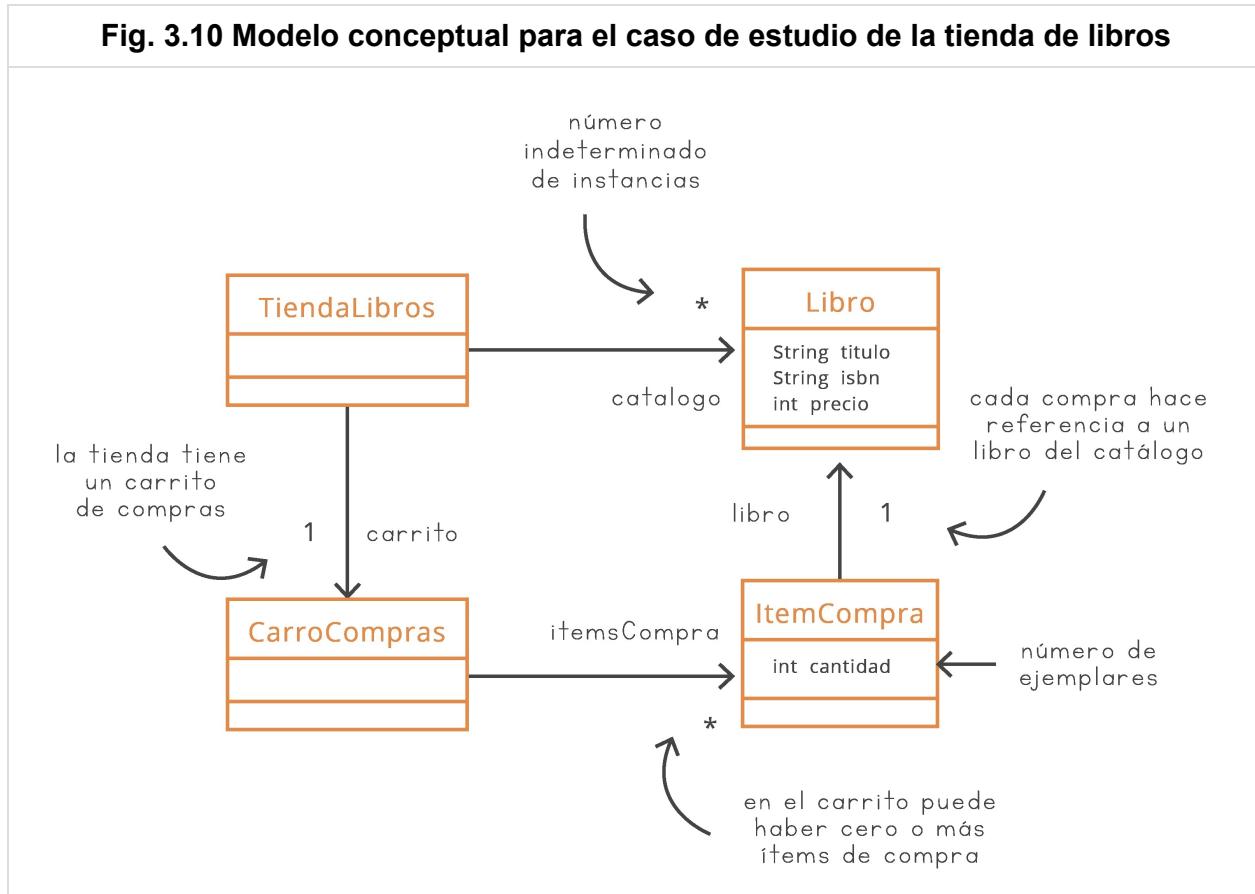
- La tienda de libros ([clase](#) TiendaLibros)
- Un libro ([clase](#) Libro)
- El carrito de compras del usuario ([clase](#) CarroCompras)
- Una compra de un libro que va dentro del carrito ([clase](#) ItemCompra)

Todas las características de las entidades identificadas en el modelo conceptual se pueden modelar con los elementos que hemos visto hasta ahora en el libro, con [excepción](#) del grupo de libros del catálogo y el grupo de compras que hay en el carrito. La dificultad que tenemos es que no podemos predecir la cardinalidad de dicho grupo de elementos y, por esta razón, el modelado con arreglos puede no ser el más adecuado.

¿En qué se diferencia del caso del avión? La diferencia radica en que el avión tiene unas dimensiones predefinidas (42 sillas en *clase* económica y 8 en *clase* ejecutiva) que no van a cambiar durante la ejecución del programa (no existe un requerimiento de agregar una silla al avión). En el caso de la tienda de libros, se plantea que el catálogo puede tener cualquier cantidad de libros y que el cliente puede comprar cualquier cantidad de ellos. Si usáramos arreglos para representar dicha información, ¿de qué dimensión deberíamos crearlos?

¿Qué hacemos si se llena el *arreglo* de libros del catálogo?

**Fig. 3.10 Modelo conceptual para el caso de estudio de la tienda de libros**



La solución a ese problema será el tema de esta parte final del nivel, en la cual presentamos las contenedoras de tamaño **variable**, la manera en que se usan a nivel de modelado del mundo y la forma en que se incorporan en los programas escritos en Java.

Por ahora démosle una mirada al diagrama de clases de la [figura 3.10](#) y recorramos cada una de las entidades identificadas:

- Una tienda de libros tiene un catálogo (así se llama la *asociación*), que corresponde a un grupo de longitud indefinida de libros (representado por el **\***). También tiene un carrito de compras.
- Un libro tiene tres atributos: un título, un ISBN y un precio.
- Un carrito de compras tiene un grupo de longitud indefinida de ítems de compra (libros que piensa comprar el usuario).
- Cada ítem de compra tiene una cantidad (el número de ejemplares que va a llevar de

un libro) y el libro del catálogo que quiere comprar.

## 8. Contenedoras de Tamaño Variable

En muchos problemas necesitamos representar grupos de atributos para los cuales no conocemos su tamaño máximo. En el caso de la tienda de libros, por ejemplo, el catálogo podría tener 100 ó 10.000 libros distintos. Para poder representar y manejar ese tipo de características, tenemos las contenedoras de tamaño [variable](#).

En el diagrama de clases de UML, las asociaciones que tienen dicha característica se representan con una cardinalidad indefinida, usando los símbolos \* o 0..N, tal como se mostró en la [figura 3.10](#).

Para implementarlas en Java, no existen elementos en el lenguaje como los arreglos, sino que es necesario utilizar algunas clases que fueron construidas con este fin.

¿Cuál es la diferencia? La principal diferencia es que para manipular las contenedoras de tamaño [variable](#) debemos utilizar la misma sintaxis que utilizamos para manejar cualquier otra [clase](#). No hay una sintaxis especial para obtener un elemento (como `[ ]` en los arreglos), ni contamos con operadores especiales (`length`).

En Java existen varias clases que nos permiten manejar contenedoras de tamaño [variable](#), todas ellas disponibles en el [paquete](#) llamado `java.util`. En este libro vamos a utilizar la [clase](#) `ArrayList`, que es eficiente e incluye toda la funcionalidad necesaria para manipular grupos de objetos. La mayor restricción que vamos a encontrar es que no permite manejar grupos de atributos de tipo simple, sino únicamente grupos de objetos. En este nivel vamos a estudiar únicamente los principales métodos de esa [clase](#), aquéllos que ofrecen las funcionalidades típicas para manejar esta [clase](#) de estructuras. Si desea conocer la descripción de todos los métodos disponibles, lo invitamos a consultar la documentación que aparece en el sitio web del lenguaje Java.

Por simplicidad, vamos a llamar [vector](#) a cualquier [implementación](#) de una estructura [contenedora de tamaño variable](#).

Al igual que con los arreglos, comenzamos ahora el recorrido para estudiar la manera de declarar un [atributo](#) de la [clase](#) `ArrayList`, la manera de tener acceso a sus elementos, la forma de modificarlo, etc. Para esto utilizaremos el caso de estudio de la tienda de libros.

### 8.1. Declaración de un Vector

Puesto que un **vector** es una **clase** común y corriente de Java, la sintaxis para declararlo es la misma que hemos utilizado en los niveles anteriores. En el ejemplo 11 se explican las declaraciones de las clases TiendaLibros y CarroCompras.

## Ejemplo 11

**Objetivo:** Mostrar la sintaxis usada en Java para declarar un **vector**.

En este ejemplo se muestran las declaraciones de las clases TiendaLibros y CarroCompras, las cuales contienen atributos de tipo **vector**.

```
package uniandes.cupi2.carrocompralibro.mundo;

import java.util.*;

public class TiendaLibros
{
    //-----
    // Atributos
    //-----
    private ArrayList catalogo;
    private CarroCompras carrito;
    ...
}
```

- Para poder usar la **clase** `ArrayList` es necesario importar su declaración, indicando el **paquete** en el que ésta se encuentra (`java.util`). Esto se hace con la instrucción `import` de Java.
- Dicha instrucción va después de la declaración del **paquete** de la **clase** y antes de su encabezado.
- En la **clase** `TiendaLibros` se declaran dos atributos: el catálogo, que es un **vector**, y el carrito de compras, que es un **objeto**.

```
package uniandes.cupi2.carrocompralibro.mundo;

import java.util.*;

public class CarroCompras
{
    //-----
    // Atributos
    //-----
    private ArrayList itemsCompra;
    ...
}
```

- En la **clase** CarroCompras se declara el grupo de ítems de compra como un **vector**.
- Se debe de nuevo importar el **paquete** en donde se encuentra la **clase** ArrayList, usando la instrucción `import` .
- Fíjese que la declaración de un **vector** utiliza la misma sintaxis que se usa para declarar cualquier otro **atributo** de la **clase**.

## 8.2 Inicialización y Tamaño de un Vector

En el constructor es necesario inicializar los vectores, al igual que hacemos con todos los demás atributos de una **clase**. Hay dos diferencias entre crear un **arreglo** y crear un **vector**:

- En los vectores se utiliza la misma sintaxis de creación de cualquier otro **objeto** (`new ArrayList()`), mientras que los arreglos utilizan los `[ ]` para indicar el tamaño (`new Clase[TAMANIO]`).
- En los vectores no es necesario definir el número de elementos que va a tener, mientras que en los arreglos es indispensable hacerlo.

### Ejemplo 12

**Objetivo:** Mostrar la manera de inicializar un **vector**.

En este ejemplo se muestran los métodos constructores de las clases TiendaLibros y CarroCompras, las cuales contienen atributos de tipo **vector**.

```
public TiendaLibros( )
{
    catalogo = new ArrayList( );
    carrito = new CarroCompras( );
}
```

- La misma sintaxis que se usa para crear un **objeto** de una **clase** (como el carrito de compras) se utiliza para crear un **vector** (el catálogo de libros).
- No hay necesidad de especificar el número de elementos que el **vector** va a contener.

```
public CarroCompras( )
{
    itemsCompra = new ArrayList( );
}
```

- Al crear un **vector** se reserva un espacio **variable** para almacenar los elementos que vayan apareciendo. Inicialmente hay 0 objetos en él.

Dos métodos de la [clase](#) `ArrayList` nos permiten conocer el número de elementos que en un momento dado hay en un [vector](#):

- `isEmpty()` : es un [método](#) que retorna verdadero si el [vector](#) no tiene elementos y falso en caso contrario. Por ejemplo, en la [clase](#) `CarroCompras`, después de llamar el constructor, la invocación del [método](#) `itemsCompra.isEmpty()` retorna verdadero.
- `size()` : es un [método](#) que retorna el número de elementos que hay en el [vector](#). Para el mismo caso planteado anteriormente, `itemsCompra.size()` es igual a 0.

Si adaptamos el esqueleto de los patrones de [algoritmo](#) para el manejo de vectores, lo único que va a cambiar es la [condición](#) de continuar en el ciclo. En lugar de usar la operación `length` de los arreglos, debemos utilizar el [método](#) `size()` de los vectores, tal como se muestra en el siguiente fragmento de [método](#) de la [clase](#) `TiendaLibros`.

```
public void esqueleto( )
{
    for( int i = 0; i < catalogo.size(); i++ )
    {
        // cuerpo del ciclo
    }
}
```

Las posiciones en los vectores, al igual que en los arreglos, comienzan en 0.

La [condición](#) para continuar en el ciclo se escribe utilizando el [método](#) `size()` de la [clase](#) `ArrayList`, en lugar del [operador](#) `length` de los arreglos. Note que los paréntesis son necesarios.

La siguiente tabla ilustra el uso de los métodos de manejo del tamaño de un [vector](#) en el caso de estudio:

Clase	Expresión	Interpretación
TiendaLibros	<code>catalogo.size()</code>	Número de libros disponibles en el catálogo.
TiendaLibros	<code>catalogo.size() == 10</code>	¿Hay 10 libros en el catálogo?
TiendaLibros	<code>catalogo.isEmpty()</code>	¿Está vacío el catálogo?
CarroCompras	<code>itemsCompra.size()</code>	Número de libros en el pedido del usuario.

## 8.3. Acceso a los Elementos de un Vector

Los elementos de un **vector** se referencian por su posición en la estructura, comenzando en la posición cero. Para esto se utiliza el **método** `get(pos)`, que recibe como **parámetro** la posición del elemento que queremos recuperar y nos retorna el **objeto** que allí se encuentra. Al recuperar un **objeto** de un **vector**, es necesario hacer explícita la **clase** a la cual éste pertenece, usando la sintaxis mostrada en el siguiente ejemplo.

## Ejemplo 13

**Objetivo:** Ilustrar el uso del **método** que nos permite recuperar un **objeto** de un **vector**.

En este ejemplo se ilustra el uso del **método** de acceso a los elementos de un **vector**.

Vamos a suponer que en la **clase** `Libro` existe el **método** `darPrecio()`, que retorna el precio del libro. Este **método** suma el precio de todos los libros del catálogo.

```
public int inventario( )
{
    int sumaPrecios = 0;
    for( int i = 0; i < catalogo.size( ); i++ )
    {
        Libro libro = ( Libro )catalogo.get( i );
        sumaPrecios += libro.darPrecio( );
    }
    return sumaPrecios;
}
```

- Es importante notar que al recuperar un **objeto** de un **vector**, se debe hacer explícito su tipo. En la instrucción en la que usamos el **método** `get( i )`, es necesario aplicar el **operador** `(Libro)` antes de hacer la **asignación** a una **variable** temporal.
- Es una buena idea guardar siempre en una **variable** temporal la referencia al **objeto** recuperado, para simplificar el código.

Cuando dentro de un **método** tratamos de acceder una posición en un **vector** con un índice no válido (menor que 0 o mayor o igual que el número de objetos que en ese momento se encuentren en el **vector**), obtenemos el error de ejecución:  
`java.lang.IndexOutOfBoundsException`.

Si no utilizamos el **operador** que indica la **clase** del **objeto** que acabamos de recuperar de un **vector**, obtenemos el siguiente mensaje del **compilador** (para el **método** del ejemplo 13): *Type mismatch: cannot convert from Object to Libro*.

Recuerde que al utilizar el **método** `get(pos)`, lo único que estamos obteniendo es una referencia al **objeto** que se encuentra referenciado desde la posición pos del **vector**. No se hace ninguna copia del **objeto**, ni se lo desplaza a ningún lado.

## 8.4. Agregar Elementos a un Vector

Los elementos de un [vector](#) se pueden agregar al final del mismo o insertar en una posición específica. Los métodos para hacerlo son los siguientes:

- **add(objeto)**: es un [método](#) que permite agregar al final del [vector](#) el [objeto](#) que se pasa como [parámetro](#). No importa cuántos elementos haya en el [vector](#), el [método](#) siempre sabe cómo buscar espacio para agregar uno más.
- **add(indice, objeto)**: es un [método](#) que permite insertar un [objeto](#) en la posición indicada por el índice especificado como [parámetro](#). Esta operación hace que el elemento que se encontraba en esa posición se desplace hacia la posición siguiente, lo mismo que el resto de los objetos en la estructura.

### Ejemplo 14

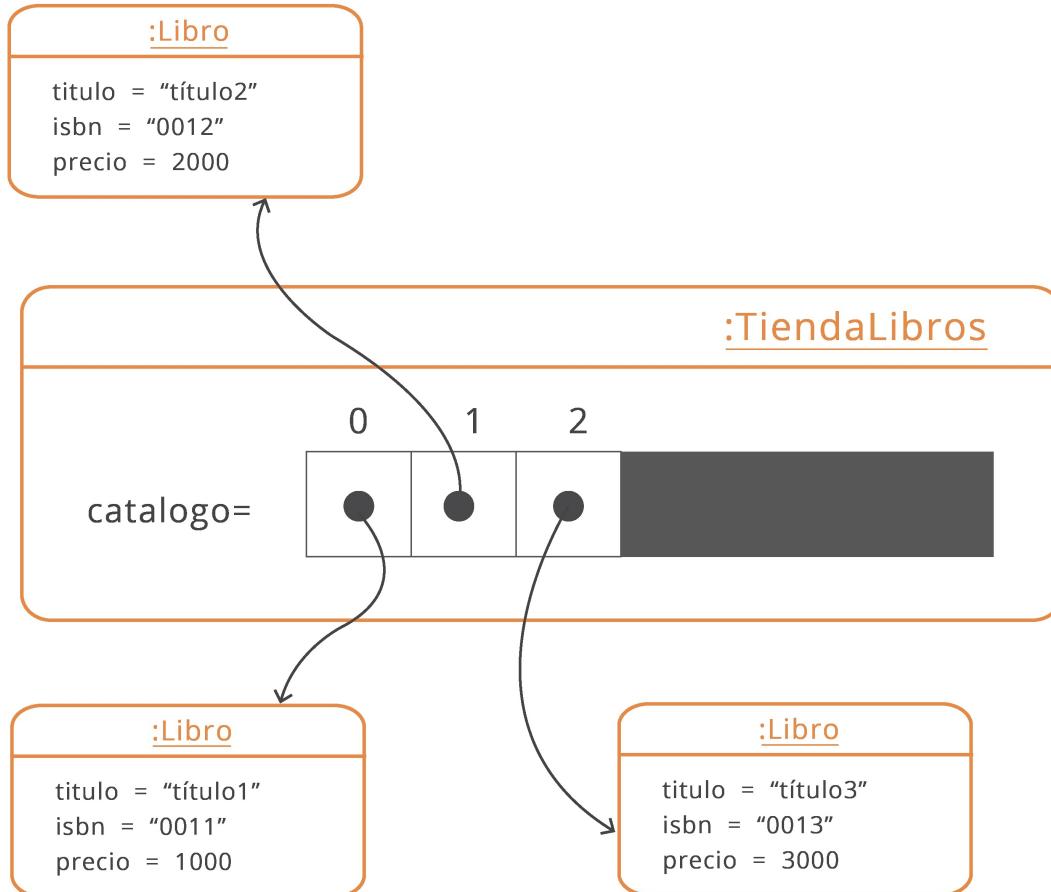
**Objetivo:** Mostrar el uso del [método](#) que agrega objetos a un [vector](#).

En este ejemplo se ilustra el uso de los métodos que permiten agregar elementos a un [vector](#). El siguiente es un [método](#) de la [clase](#) TiendaLibros que añade tres libros al catálogo.

```
public void agregarTresLibros( )
{
    Libro lb1 = new Libro( "título1", "0011", 1000 );
    Libro lb2 = new Libro( "título2", "0012", 2000 );
    Libro lb3 = new Libro( "título3", "0013", 3000 );

    catalogo.add( lb2 );
    catalogo.add( lb3 );
    catalogo.add( 0, lb1 );
}
```

- En el [método](#) se crean inicialmente los tres libros. Luego se agrega el segundo de los libros (`lb2`). Como el [vector](#) estaba vacío, el nuevo elemento queda en la posición 0 del catálogo. Después se añade el tercer libro (`lb3`), que queda en la posición 1. Finalmente se inserta el primer libro (`lb1`) en la posición 0, lo que desplaza el libro 2 a la posición 1 y el libro 3 a la posición 2.



- En este **diagrama de objetos** se puede apreciar el estado del catálogo después de ejecutar este **método**.
- Si usamos el **método size()** para el catálogo, debe responder 3.
- En el dibujo dejamos en gris las casillas posteriores a la 2, para indicar que el **vector** las puede ocupar cuando las necesite.

## 8.5. Reemplazar un Elemento en un Vector

Cuando se quiere reemplazar un **objeto** por otro en un **vector**, se utiliza el **método set()**, que recibe como parámetros el índice del elemento que se debe reemplazar y el **objeto** que debe tomar ahora esa posición.

Este **método** es muy útil para ordenar un **vector** o para clasificar bajo algún concepto los elementos que allí se encuentran. En el ejemplo 15 aparece un **método de la clase** **TiendaLibros** que permite intercambiar dos libros del catálogo, dadas sus posiciones en el **vector** que los contiene.

### Ejemplo 15

**Objetivo:** Mostrar la manera de reemplazar un [objeto](#) en un [vector](#).

En este ejemplo se ilustra el uso del [método](#) que reemplaza un [objeto](#) por otro en un [vector](#). El [método](#) de la [clase](#) TiendaLibros recibe las posiciones en el catálogo de los libros que debe intercambiar.

```
public void intercambiar( int pos1, int pos2 )
{
    Libro lb1 = ( Libro ) catalogo.get( pos1 );
    Libro lb2 = ( Libro ) catalogo.get( pos2 );

    catalogo.set( pos1, lb2 );
    catalogo.set( pos2, lb1 );
}
```

Cuando se intercambian los elementos en cualquier estructura es indispensable guardar al menos uno de ellos en una [variable](#) temporal. En este [método](#) decidimos usar dos variables por claridad. En este [método](#) suponemos que las dos posiciones dadas son válidas (que tienen valores entre 0 y `catalogo.size( ) -1`).

El [método](#) `set()` no hace sino reemplazar la referencia al [objeto](#) que se encuentra almacenada en la casilla. Se puede ver simplemente como la manera de asignar un nuevo valor a una casilla. La referencia que allí se encontraba se pierde, a menos que haya sido guardada en algún otro lugar.

## 8.6. Eliminar un Elemento de un Vector

De la misma manera que es posible agregar elementos a un [vector](#), también es posible eliminarlos. Piense en el caso de la tienda de libros. Si el usuario decide sacar un elemento de su carrito de compras, nosotros en el programa debemos quitar del respectivo [vector](#) el [objeto](#) que lo representaba. Después de eliminada la referencia a un [objeto](#), esta posición es ocupada por el elemento que se encontraba después de él en el [vector](#).

El [método](#) de la [clase](#) `ArrayList` que se usa para eliminar un elemento se llama `remove()` y recibe como [parámetro](#) la posición del elemento que se quiere eliminar (un valor entre 0 y el número de elementos menos 1). Al usar esta operación, se debe tener en cuenta que el tamaño de la estructura disminuye en 1, por lo que se debe tener cuidado en el momento de definir la [condición](#) de continuación de los ciclos.

Es importante recalcar que el hecho de quitar un [objeto](#) de un [vector](#) no implica necesariamente su destrucción. Lo único que estamos haciendo es eliminando una referencia al [objeto](#). Si queremos mantenerlo vivo, basta con guardar su referencia en otro lado, por ejemplo en una [variable](#).

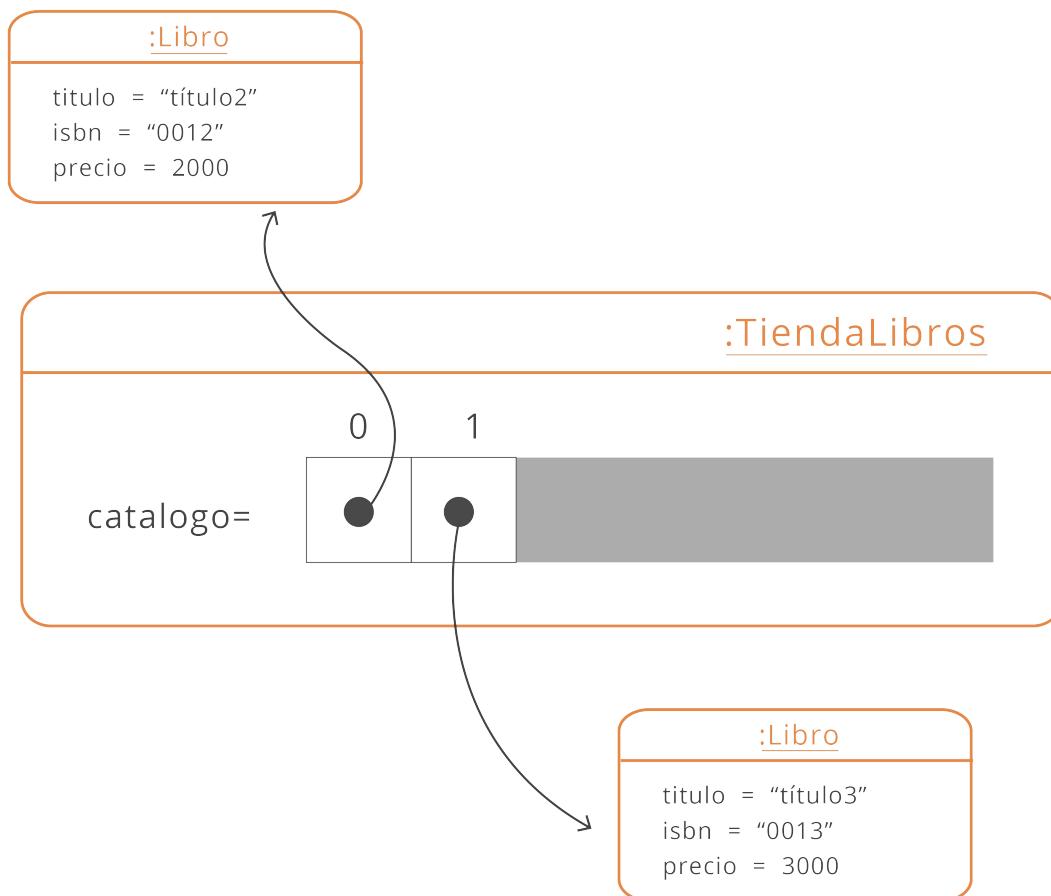
## Ejemplo 16

**Objetivo:** Mostrar la manera de utilizar el **método** que elimina un **objeto** de un **vector**.

En este ejemplo presentamos un **método** de la **clase** TiendaLibros que elimina el primer libro del catálogo. Ilustramos el resultado usando el **diagrama de objetos** del ejemplo 14.

```
public void eliminarPrimerLibro( )
{
    catalogo.remove( 0 );
}
```

Este **método** elimina del catálogo la referencia al primer libro de la tienda. Después de su ejecución, todos los libros se catálogo.



- Si ejecutamos este **método** sobre el **diagrama de objetos** del ejemplo 14, obtenemos el diagrama que aparece en esta figura.
- El libro que estaba en la posición 1 pasa a la posición 0, y el libro de la posición 2 pasa a la posición 1.
- Ahora `catalogo.size( )` es igual a 2.

Ya que hemos terminado de ver los principales métodos con los que contamos para manejar los elementos de un [vector](#), vamos a comenzar a escribir los métodos de la [clase](#) del caso de estudio. Comenzamos con las declaraciones de las clases simples y seguimos con los métodos que manejan los vectores.

## 8.7. Construcción del Programa del Caso de Estudio

### 8.7.1. La Clase Libro

La [clase](#) Libro sólo es responsable de manejar sus tres atributos. Para esto cuenta con un [método](#) constructor y tres métodos analizadores:

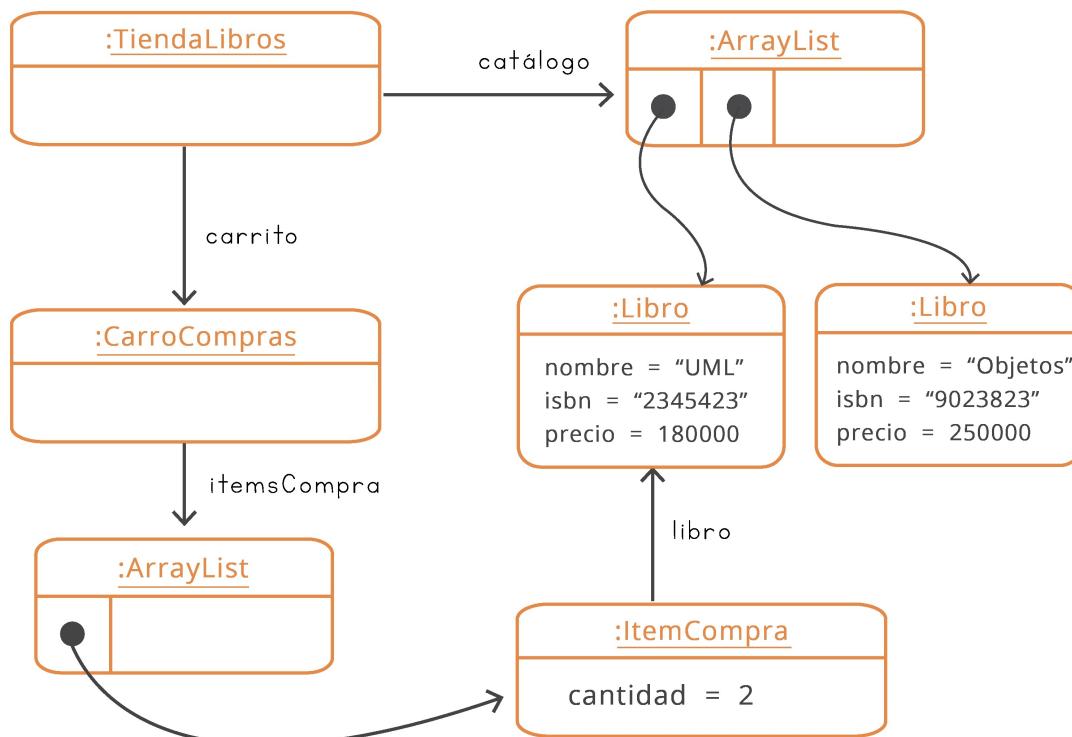
<code>Libro( String unTitulo, String unISBN, int unPrecio )</code>	<b>Método constructor.</b>
<code>String darTitulo( )</code>	Retorna el título del libro.
<code>String darISBN( )</code>	Retorna el ISBN del libro.
<code>String darPrecio( )</code>	Retorna el precio del libro.

### 8.7.2. La Clase ItemCompra

Cada [objeto](#) de la [clase](#) ItemCompra tiene una [asociación](#) con un libro y almacena el número de ejemplares que el usuario va a comprar de dicho libro. Aquí es importante resaltar que los objetos de la [clase](#) Libro serán compartidos por el catálogo y por los ítems de compra, como se ilustra en el [diagrama de objetos](#) de la [figura 3.11](#).

Los métodos de esta [clase](#) se resumen en la siguiente tabla:

<code>ItemCompra( Libro unLibro, int unaCantidad )</code>	<b>Método constructor.</b>
<code>Libro darLibro( )</code>	Retorna el libro pedido.
<code>String darISBNItem( )</code>	Retorna el ISBN del libro pedido.
<code>int darCantidadSolicitada( )</code>	Retorna el número de ejemplares.
<code>void cambiarCantidadSolicitada( int nuevaCantidad )</code>	Cambia el número de ejemplares.
<code>int calcularSubtotalItem( )</code>	Calcula el subtotal.

**Fig. 3.11 Diagrama de objetos para ilustrar el caso de la tienda de libros**

En la figura 3.11 se puede apreciar el caso en el que el usuario tiene en su carrito de compras dos ejemplares del libro "UML", el cual forma parte también del catálogo. En este diagrama decidimos mostrar los vectores como objetos externos a las clases que los usan. Esta representación se ajusta más a la realidad que la que usamos en ejemplos anteriores, aunque es menos simple. Ambas maneras de mostrar el [diagrama de objetos](#) son válidas. Observe, por ejemplo, que el [objeto](#) llamado [catalogo](#) es una [asociación](#) hacia un [objeto](#) de la [clase](#) [ArrayList](#), que mantiene las referencias a los objetos que representan los libros.

### 8.7.3. La Clase TiendaLibros

En la tarea 10 vamos a desarrollar algunos de los métodos de la [clase](#) [TiendaLibros](#). Sus principales responsabilidades se resumen en la siguiente tabla:

TiendaLibros( )	Método constructor.
void adicionarLibroCatalogo( Libro nuevoLibro )	Añade un libro dado al catálogo. Si el libro ya está en el catálogo, el <b>método</b> no hace nada.
void crearNuevaCompra( )	Inicializa el carrito de compras, eliminando todos los libros del pedido actual.
Libro buscarLibro( String isbn )	Localiza un libro del catálogo dado su ISBN. Si no lo encuentra retorna <code>null</code> .

## Tarea 10

**Objetivo:** Desarrollar los métodos de la **clase** TiendaLibros que nos permiten implementar los requerimientos funcionales del caso de estudio.

Para cada uno de los problemas que se plantean a continuación, escriba el **método** que lo resuelve. No olvide identificar primero el patrón de **algoritmo** que se necesita y usar las guías que se dieron en secciones anteriores.

Localizar un libro en el catálogo, dado su ISBN. Si no lo encuentra, el **método** debe retornar `null`:

```
public Libro buscarLibro( String isbn )
{



}
```

Agregar un libro en el catálogo, si no existe ya un libro con ese ISBN. Utilice el **método** anterior:

```
public void adicionarLibroCatalogo( Libro nuevoLibro )  
{  
  
}  
}
```

## 8.7.4. La Clase CarroCompras

La [clase](#) CarroCompras es responsable de agregar un ítem de compra, borrar un ítem de la lista y calcular el valor total que debe pagar el usuario por los libros.

Al igual que en el caso de los arreglos, si antes de usar un [vector](#) no lo hemos creado adecuadamente, se va a generar el error de ejecución: *java.lang.NullPointerException*.

### Tarea 11

**Objetivo:** Desarrollar los métodos de la [clase](#) CarroCompras.

Para cada uno de los problemas que se plantean a continuación, escriba el [método](#) que lo resuelve.

Retornar, si existe, un ítem de compra donde esté el libro con el ISBN dado:

```
public ItemCompra buscarItemCompraLibro( String isbnBuscado )  
{  
  
}  
}
```

Agregar una cantidad de ejemplares de un libro al pedido actual. El [método](#) debe considerar el caso en el que ese libro ya se encuentre en el pedido, caso en el cual sólo debe incrementar el número de ejemplares. Si el libro no se encuentra, el [método](#) debe crear un

nuevo ItemCompra.

```
public void adicionarCompra( Libro libro, int cantidad )
{
}
```

Calcular el monto total de la compra del usuario. Para esto debe tener en cuenta el precio de cada libro y el número de ejemplares de cada uno que hay en el pedido.

```
public int calcularValorTotalCompra( )
{
}
```

Eliminar del pedido el libro que tiene el ISBN dado como [parámetro](#). Si no hay ningún libro con ese ISBN, el [método](#) no hace nada.

```
public void borrarItemCompra( String isbn )
{
}
```



## 9. Uso de Ciclos en Otros Contextos

Aunque hasta este momento sólo hemos mostrado las instrucciones iterativas como una manera de manejar información que se encuentra en estructuras contenedoras, dichas instrucciones también se usan muy comúnmente en otros contextos. En el ejemplo 17 mostramos su uso para calcular el valor de una función aritmética.

### Ejemplo 17

**Objetivo:** Mostrar el uso de las instrucciones iterativas en un contexto diferente al de manipulación de estructuras contenedoras.

En este ejemplo presentamos la manera de escribir un [método](#) para calcular el factorial de un número. La función factorial aplicada a un número entero  $n$  (en matemáticas a ese valor se le representa como  $n!$ ) se define como el producto de todos los valores enteros positivos menores o iguales al valor en cuestión. Planteado de otra manera, tenemos que:

- $\text{factorial}(1)$  es igual a 1.
- $\text{factorial}(n) = n * \text{factorial}(n - 1)$ .

Por ejemplo,  $\text{factorial}(5) = 5 * 4 * 3 * 2 * 1 = 120$

Esta función define también que  $\text{factorial}(0) = 1$ .

Si queremos construir un [método](#) capaz de calcular dicho valor, podemos utilizar una instrucción iterativa, como se muestra a continuación.

```

package uniandes.cupi2.matematicas;
public class Matematica
{
    public static int factorial( int num )
    {
        if( num == 0 )
            return 1;
        else
        {
            int acum = 1;

            for( int i = 1; i <= num; i++ )
                acum = acum * i;

            return acum;
        }
    }
}

```

El [método](#) lo declaramos de manera especial ( `static` ) y su modo de uso es como aparece más abajo en este mismo ejemplo.

El primer caso que tenemos es que el valor del [parámetro](#) sea 0. La respuesta en ese caso es 1. Hasta ahí es fácil.

En el caso general, debemos multiplicar todos los valores desde 1 hasta el valor que recibimos como [parámetro](#) e ir acumulando el resultado en una [variable](#) llamada "`acum`". Al final el [método](#) retorna dicho valor.

Esta solución no es otra que el patrón de [recorrido total](#) aplicado a la secuencia de números. Aunque no estén almacenados en un [arreglo](#), se pueden imaginar uno después del otro, con el índice recorriéndolos de izquierda a derecha. Este uso de las instrucciones iterativas no tiene una teoría distinta a la vista en este capítulo.

```

int fact = Matematica.factorial( i );

```

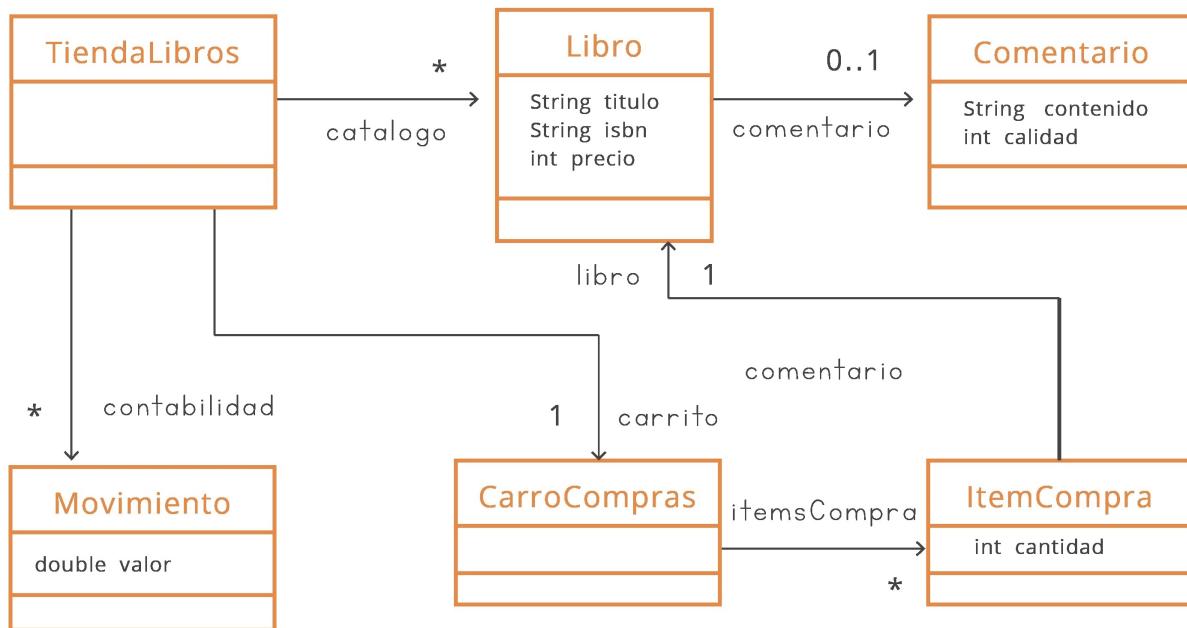
La llamada del [método](#) se hace utilizando esta sintaxis. Como es una función aritmética que no está asociada con ningún elemento del mundo, debemos usar el nombre de la [clase](#) para hacer la invocación.

# Creación de una Clase en Java

## Tarea 12

**Objetivo:** Agregar una nueva [clase](#) en un programa escrito en Java.

En esta tarea vamos a extender el caso de estudio de la tienda de libros, agregando dos clases nuevas, en un [paquete](#) distinto a los ya definidos. Siga los pasos que se detallan a continuación:



Este es el diagrama de clases que queremos construir. Hay dos clases adicionales: una para modelar el concepto de movimiento contable (con el valor de cada venta que se hace en la tienda) y otra con un comentario que se hace opcionalmente sobre cada libro. Tome nota de las nuevas asociaciones que aparecen.

1. Ejecute Eclipse y abra el proyecto de la [tienda de libros](#). Localice el directorio en el cual se guardan los programas fuente.
2. Vamos a crear los archivos de las clases Movimiento y Comentario en un nuevo [paquete](#) llamado `uniandes.cupi2.carrocompralibro.extension`. Para esto, debemos crear primero el [paquete](#). Para crear un [paquete](#) en Java, seleccione la opción *File/New/Package* del menú principal o la opción *New/Package* del menú emergente que aparece al hacer clic derecho sobre el directorio de fuentes.
3. Una vez creado el [paquete](#), podemos crear la [clase](#) allí dentro, seleccionando la opción *File/New/Class* del menú principal o la opción *New/Class* del menú emergente que aparece al hacer clic derecho sobre el [paquete](#) de clases elegido. En la [ventana](#) que

abre el asistente de creación de clases, podemos ver el directorio de fuentes y el **paquete** donde se ubicará la **clase**. Allí debemos teclear el nombre de la **clase**. Al oprimir el botón *Finish*, el editor abrirá la **clase** y le permitirá completarla con sus atributos y métodos. Siguiendo el proceso antes mencionado, cree las clases Movimiento y Comentario.

4. El siguiente paso es agregar los atributos que van a representar las asociaciones hacia esas clases. Abra para esto la **clase** Tienda. Agregue el **atributo** de la **clase** Comentario tal como se describe en el diagrama de clases. ¿Por qué el **compilador** no reconoce la nueva **clase**? Sencillamente porque está en otro **paquete**, el cual debemos importar. Añada la instrucción para importar las clases del nuevo **paquete**. Esta importación puede hacerla manualmente o utilizando el comando Control+Mayús+O para que el editor agregue automáticamente todas las importaciones que necesite.
5. Agregue el **atributo** contabilidad a la **clase** TiendaLibro, representándolo como un **vector**. En este caso sólo vamos a necesitar importar la **clase** Movimiento cuando construyamos el **método** que utiliza dicho **vector**, puesto que es la primera vez que hacemos referencia directa a esta **clase**.
6. Las clases antes mencionadas también se habrían podido crear desde cualquier editor de texto simple (por ejemplo, el bloc de notas). Basta con crear el **archivo**, salvarlo en el directorio que representa el **paquete** y, luego, entrar a Eclipse y utilizar la opción **Refresh** del menú emergente que aparece al hacer clic derecho sobre el proyecto.
7. En la **clase** Comentario agregue el constructor y dos métodos para recuperar el contenido del comentario y el índice de calidad otorgado.
8. En la **clase** Movimiento escriba el constructor y un **método** para recuperar el monto de la transacción.
9. En la **clase** Libro, añada un **método** que agregue un comentario al libro y otro que lo retorne.
10. En la **clase** TiendaLibros, añada los siguientes métodos: (a) un **método** para agregar un movimiento contable, (b) un **método** para calcular el monto total de las ventas y (c) un **método** que calcule el número total de libros del catálogo que tienen un comentario. No olvide modificar el **método** para hacer una venta, de manera que al final de la transacción agregue a la contabilidad de la tienda el respectivo movimiento contable.

# 10. Hojas de Trabajo

## 10.1. Hoja de Trabajo Nº 1: Un Parqueadero

Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

**Enunciado.** Analice el siguiente enunciado e identifique el mundo del problema, lo que se quiere que haga el programa y las restricciones para desarrollarlo.

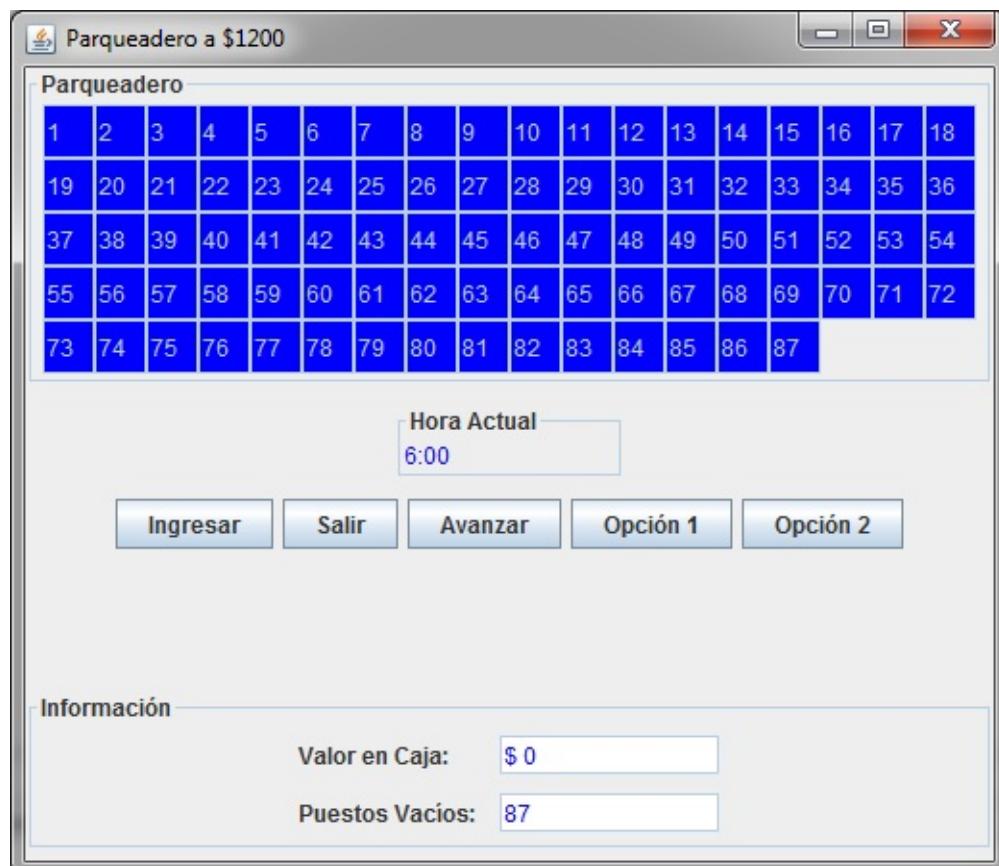
Se quiere construir una aplicación para administrar un parqueadero (lugar de estacionamiento para autos). Dicho parqueadero tiene 87 puestos numerados del 1 al 87. En cada puesto se puede aparcar un sólo automóvil (que representaremos con una clase llamada Carro), el cual se identifica por su placa. El parqueadero tiene una tarifa por hora o fracción de hora, que cambia cada vez que el gobierno lo autoriza.

De cada vehículo aparcado se debe conocer la hora en la que entró, que corresponde a un valor entre 6 y 20, dado que el parqueadero está abierto entre 6 de la mañana y 8 de la noche.

Se espera que la aplicación que se quiere construir permita hacer lo siguiente:

- (1) A un automóvil que llega, decirle el puesto en el que se debe aparcar (si hay cupo).
- (2) A un automóvil que sale, decirle cuánto debe pagar.
- (3) Al administrador del parqueadero, decirle cuánto dinero se ha recogido en el día.
- (4) Al administrador del parqueadero, decirle cuántos puestos libres quedan.

La siguiente es la [interfaz de usuario](#) propuesta para el programa, donde los puestos ocupados deben aparecer en un color distinto.



**Requerimientos funcionales.** Describa los cuatro requerimientos funcionales de la aplicación que haya identificado en el enunciado.

## Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 4

Nombre	
Resumen	
Entradas	
Resultado	

**Modelo del mundo.** Complete el diagrama de clases con los atributos, las constantes y las asociaciones.

**Diagrama UML: Parqueadero**



**Diagrama UML: Puesto**



### Diagrama UML: Carro



**Declaración de arreglos.** Para las siguientes clases, escriba la declaración de los atributos indicados en el comentario (como contenedoras del tipo dado), así como las constantes necesarias para manejarlos.

```
public class Parqueadero
{
    //-----
    // Constantes
    //-----
    /** Indica el número de puestos en el parqueadero */

    //-----
    // Atributos
    //-----
    /** Arreglo de puestos */

}
```

**Inicialización de arreglos.** Escriba el constructor de la [clase](#) para inicializar las contenedoras declaradas en el punto anterior.

```
public Parqueadero( )  
{  
  
}  
}
```

**Patrones de algoritmos.** Desarrolle los siguientes métodos de la [clase](#) Parqueadero, identificando el tipo de patrón de [algoritmo](#) al que pertenece y siguiendo las respectivas guías

## Método 1

Contar y retornar el número total de puestos ocupados.

```
public int totalPuestosOcupados( )  
{  
  
}  
}
```

## Método 2

Informar si en el parqueadero hay un automóvil cuya placa comience con la letra dada como [parámetro](#).

```
public boolean existePlacaIniciaCon( char letra )  
{  
  
}  
}
```

## Método 3

Retornar el número de automóviles en el parqueadero que llegaron antes del mediodía.

```
public int totalCarrosIngresoManana( )  
{  
  
}  
}
```

## Método 4

Retornar el último automóvil en ingresar al parqueadero. Si el parqueadero está vacío retorna null.

```
public Carro carroLlegadaMasReciente( )  
{  
  
}  
}
```

## Método 5

Informar si en algún lugar del parqueadero hay dos puestos libres consecutivos. Esto se hace cuando el vehículo que se quiere aparcar es muy grande.

```
public boolean dosPuestosLibresConsecutivos( )  
{  
  
}  
}
```

## Método 6

Informar si hay dos automóviles en el parqueadero con la misma placa.

```
public boolean placaRepetida( )  
{  
  
}  
}
```

## 10.2 Hoja de Trabajo Nº 2: Lista de Contactos

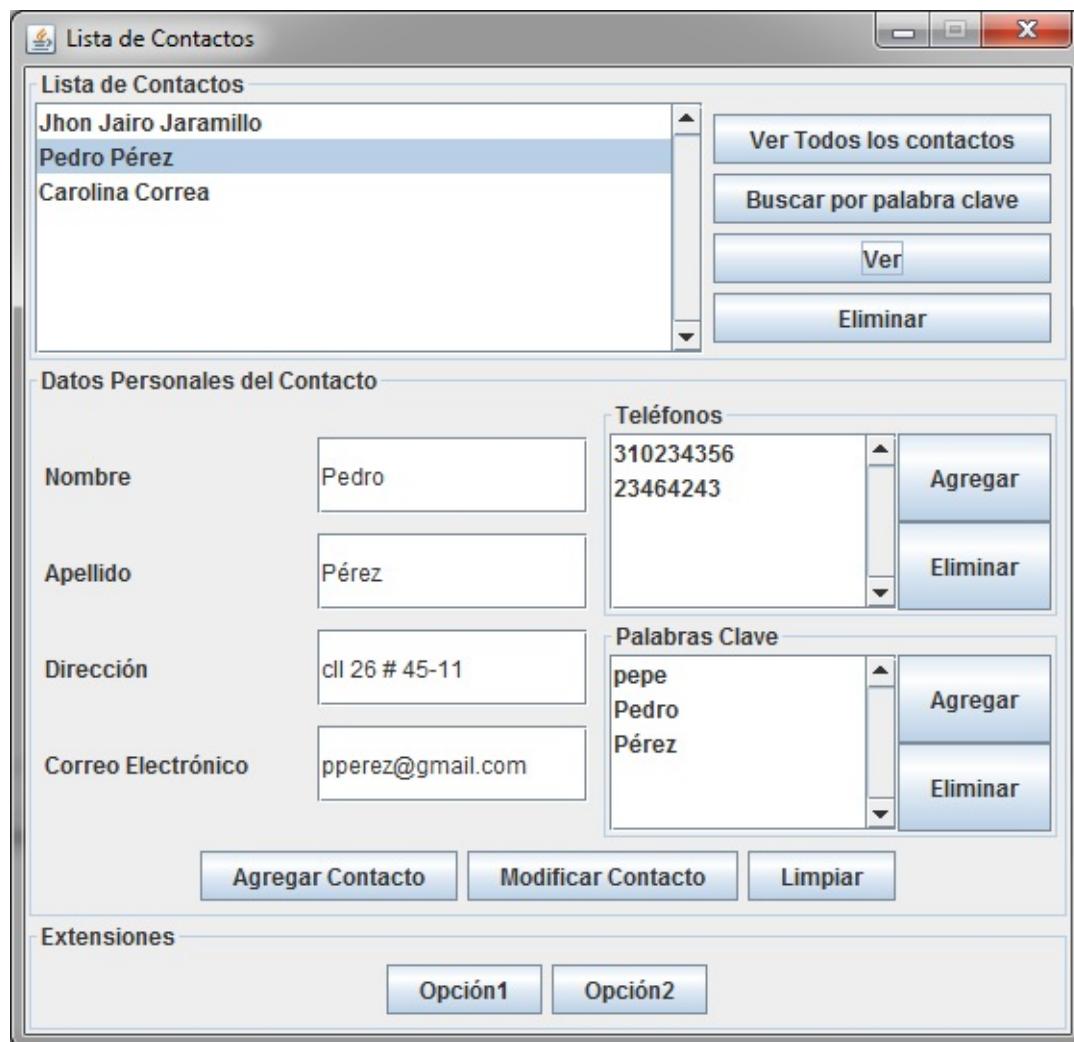
Descargue esta hoja de trabajo a través de los siguientes enlaces: [Descargar PDF](#) | [Descargar Word](#).

**Enunciado.** Analice el siguiente enunciado e identifique el mundo del problema, lo que se quiere que haga el programa y las restricciones para desarrollarlo.

Se quiere construir un programa para manejar la lista de contactos de una persona. Un contacto tiene nombre, apellido, una dirección, un correo electrónico, varios teléfonos y un conjunto de palabras clave que se utilizan para facilitar su búsqueda. El nombre completo (nombre + apellido) de cada contacto debe ser único. Tanto el nombre como el apellido se usan como palabras clave para las búsquedas.

En el programa de contactos se debe poder (1) agregar un nuevo contacto, (2) eliminar un contacto ya existente, (3) ver la información detallada de un contacto, (4) modificar la información de un contacto y (5) buscar contactos usando las palabras clave.

La siguiente es la [interfaz de usuario](#) propuesta para el programa de la lista de contactos.



**Requerimientos funcionales.** Describa los cinco requerimientos funcionales de la aplicación que haya identificado en el enunciado.

## Requerimiento Funcional 1

<b>Nombre</b>	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 4

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 5

Nombre	
Resumen	
Entradas	
Resultado	

**Modelo del mundo.** Complete el diagrama de clases con los atributos, las constantes y las asociaciones.

**Diagrama UML: ListaDeContactos**



**Diagrama UML: Contacto**



**Declaración de arreglos.** Para las siguientes clases, escriba la declaración de los atributos indicados en el comentario (como contenedoras del tipo dado).

```
public class Contacto
{
    //-----
    // Atributos
    //-----
    private String nombre;
    private String apellido;
    private String direccion;
    private String correo;

    /** Vector de teléfonos del contacto */

    /** Vector de palabras clave del contacto */

}
```

```
public class ListaDeContactos
{
    //-----
    // Atributos
    //-----
    /** Vector de contactos */

}
```

**Inicialización de arreglos.** Escriba el constructor de las clases dadas.

```
public Contacto ()  
{  
}  
}
```

```
public ListaDeContactos ()  
{  
}  
}
```

**Patrones de algoritmos.** Desarrolle los siguientes métodos de la [clase](#) indicada, identificando el tipo de patrón de [algoritmo](#) al que pertenece y siguiendo las respectivas guías.

## Metodo

[Clase:](#) **Contacto**

Contar el número de palabras clave que empiezan por la letra dada como [parámetro](#).

```
public int totalPalabrasInicianCon( char letra )
{
}
}
```

## Metodo 2

### Clase: Contacto

Informar si el contacto tiene algún teléfono que comienza por el prefijo dado como parámetro.

```
public boolean existeTelefonoIniciaCon( String prefijo )
{
}

}
```

## Metodo 3

**Clase:** Contacto

Retornar la primera palabra clave que termina con la cadena dada.

```
public String darPalabraTerminaCon( String cadena )
{
}

}
```

## Metodo 4

**Clase:** Contacto

Contar el número de palabras clave que son prefijo (parte inicial) de otras palabras clave.

```
public int totalPalabrasPrefijo( )
{
}
}
```

## Metodo 5

**Clase:** ListaDeContacto

Contar el número de contactos cuyo nombre es igual al recibido como [parámetro](#).

```
public int totalContactosConNombre( String nombre )
{
}
}
```

## Metodo 6

**Clase:** ListadeContactos

Informar si hay dos contactos en la lista con la misma dirección de correo electrónico.

```
public boolean correoRepetido( )  
{  
  
}  
}
```

## Metodo 7

**Clase:** **ListadeContactos**

Retornar el contacto con el mayor número de palabras clave.

```
public Contacto contactoConMasPalabras( )  
{  
  
}  
}
```