Programación orientada a objetos

Capítulo 4

Agrupar objetos

- Separación de código	
Tema 4. Agrupar objetos. Semana 4	
1- Librerías de clases 2- Clases genéricas 3- Colecciones de tamaño flexible: ArrayList a. Procesamiento de colecciones b. Estructuras de control: los bucles for-each while c. Acceso mediante índices e iteradores 4- Colecciones de tamaño fijo: Array a. Creación y declaración de arrays b. Uso de arrays c. Estructuras de control: el bucle for	1- Estudiar el capítulo 4 del libro base para la Unidad Didáctica I 2- Realizar los ejercicios en el entorno BlueJ sugeridos en el libro base 3- Definición de estructuras de almacenamiento e implementación de los métodos y constructores necesarios para la realización de la práctica

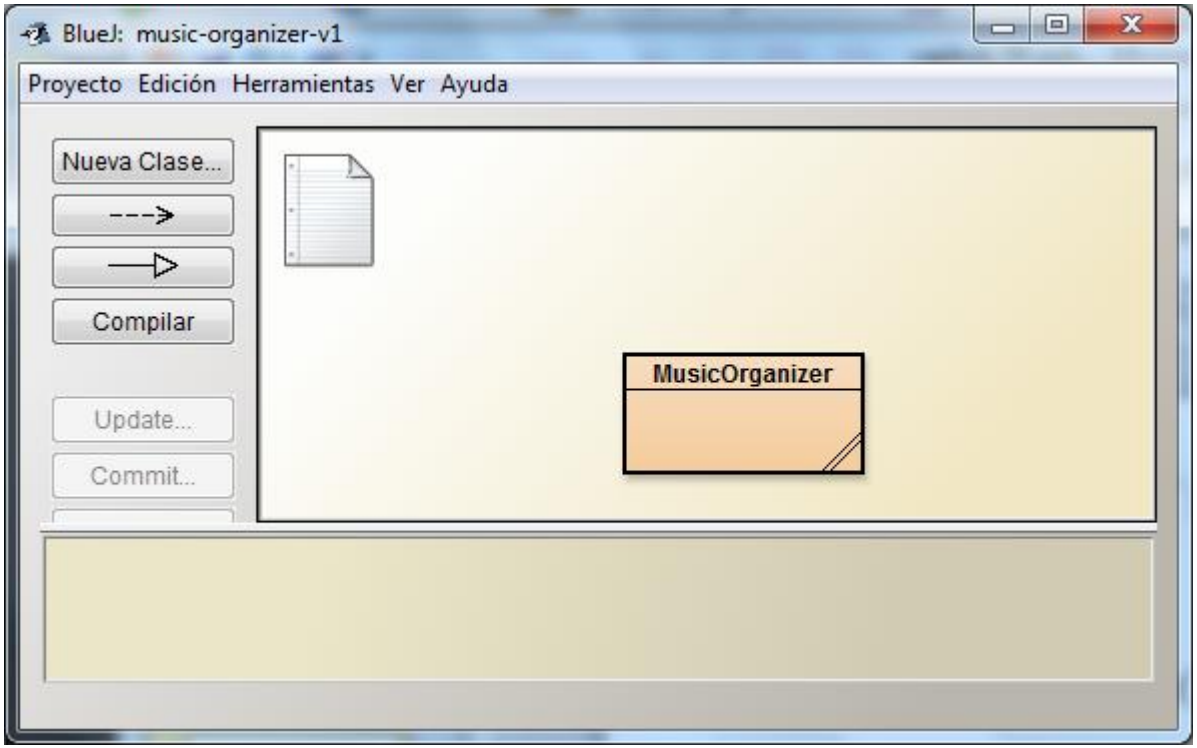
4.2 La Colección de objetos

Las **colecciones de objetos** son objetos que pueden almacenar un número arbitrario de otros objetos.

Agrupar cosas para referirnos y manejarlas de forma conjunta

4.3 Un organizador para archivos de música

- Music-organizer-v1
- Librerías de clases
  - Las librerías o paquetes de Java contiene cientos de clases



4.4 Utilización de una clase librería

```
import java.util.ArrayList;

/**
 * A class to hold details of audio files.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2011.07.31
 */
public class MusicOrganizer
{
    // An ArrayList for storing the file names of music files.
    private ArrayList<String> files;

    /**
     * Create a MusicOrganizer
     */
    public MusicOrganizer()
    {
        files = new ArrayList<String>();
    }

    /**
     * Add a file to the collection.
     * @param filename The file to be added.
     */
    public void addFile(String filename)
    {
        files.add(filename);
    }

    /**
```

```
    /**
     * Return the number of files in the collection.
     * @return The number of files in the collection.
     */
    public int getNumberOfFiles()
    {
        return files.size();
    }

    /**
     * List a file from the collection.
     * @param index The index of the file to be listed.
     */
    public void listFile(int index)
    {
        if(index >= 0 && index < files.size()) {
            String filename = files.get(index);
            System.out.println(filename);
        }
    }

    /**
     * Remove a file from the collection.
     * @param index The index of the file to be removed.
     */
    public void removeFile(int index)
    {
        if(index >= 0 && index < files.size()) {
            files.remove(index);
        }
    }
}
```

4.4.1 Importación de una clase librería

La primera línea de esta clase muestra el modo en que obtenemos el acceso a una clase de una biblioteca de Java mediante la *sentencia import*:

```
import java.util.ArrayList;
```

```
private ArrayList<String> notas;
```

Aquí vemos una nueva construcción: la mención de String entre símbolos de menor (<) y de mayor (>): `<String>`.

La clase `ArrayList` declara muchos métodos pero en este momento, sólo usaremos tres de ellos para implementar la funcionalidad que requerimos: `add`, `size` y `get`.

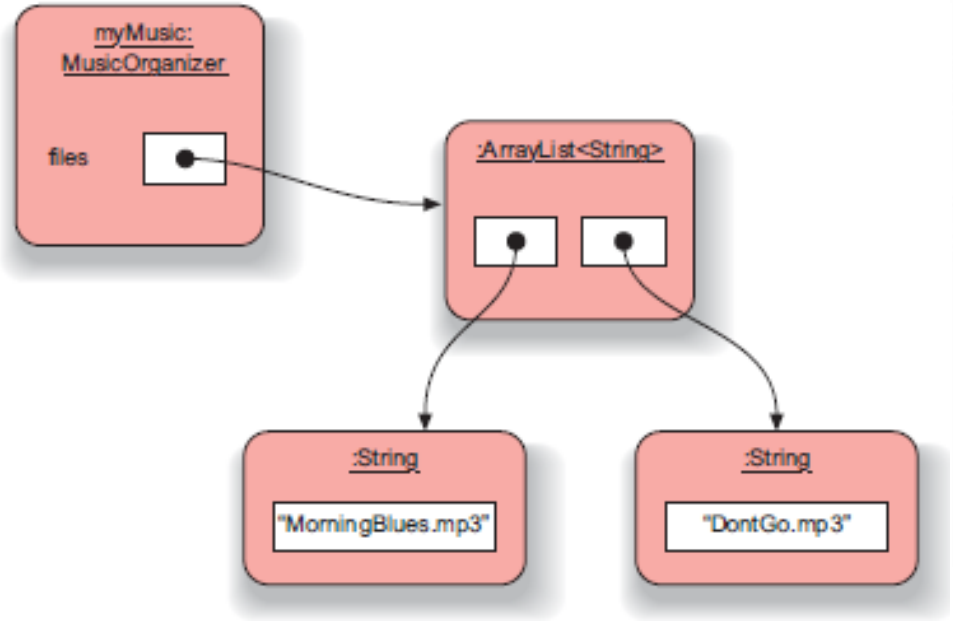
This allows us to use the so-called *diamond notation* as follows:

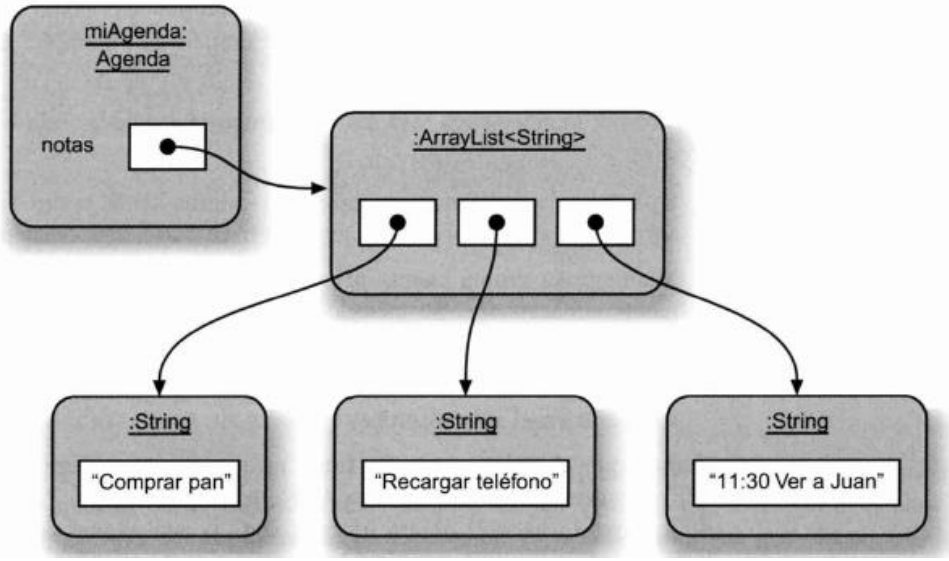
```
files = new ArrayList<>();
```

4.5 Estructura de objetos con “colecciones”

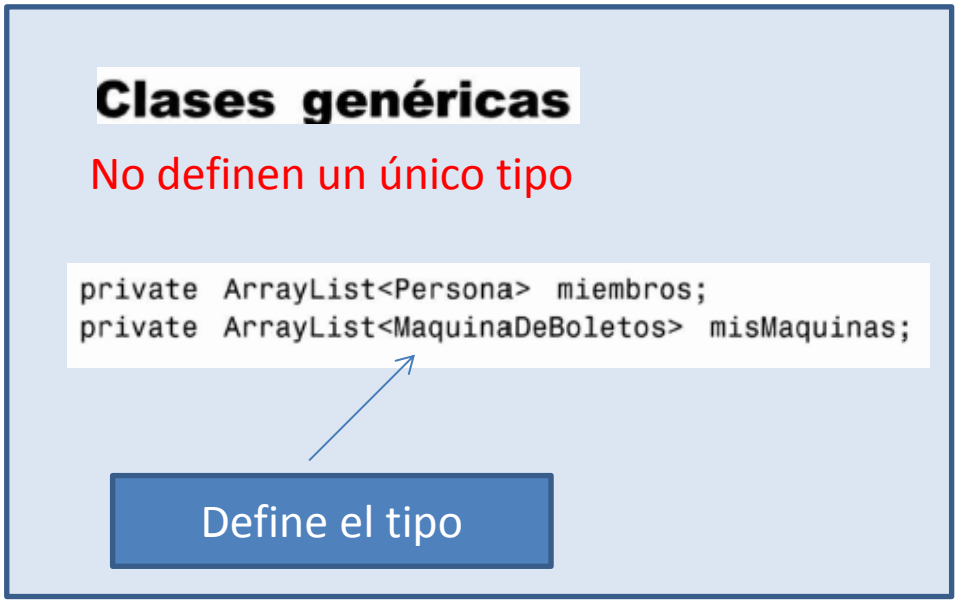
Existen por lo menos tres características importantes de la clase `ArrayList` que debería observar:

- Es capaz de aumentar su capacidad interna tanto como se requiera: cuando se agregan más elementos, simplemente hace suficiente espacio para ellos.
- Mantiene su propia cuenta privada de la cantidad de elementos que tiene actualmente almacenados. Su método `size` devuelve el número de objetos que contiene actualmente.
- Mantiene el orden de los elementos que se agregan, por lo que más tarde se pueden recuperar en el mismo orden.



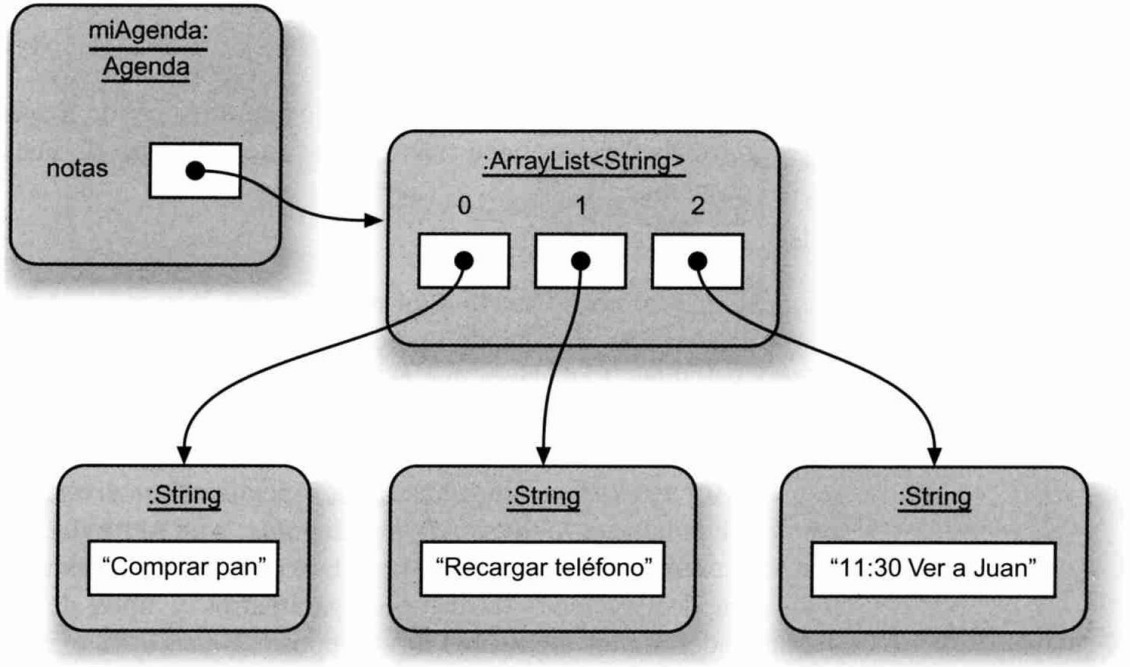


4.6 Clases genéricas



4.7 Numeración dentro de las colecciones

comienza a partir de cero. La posición que ocupa un objeto en una colección es conocida más comúnmente como su *índice*. El primer elemento que se agrega a una colección tiene por índice al número 0, el segundo tiene al número 1, y así sucesivamente.



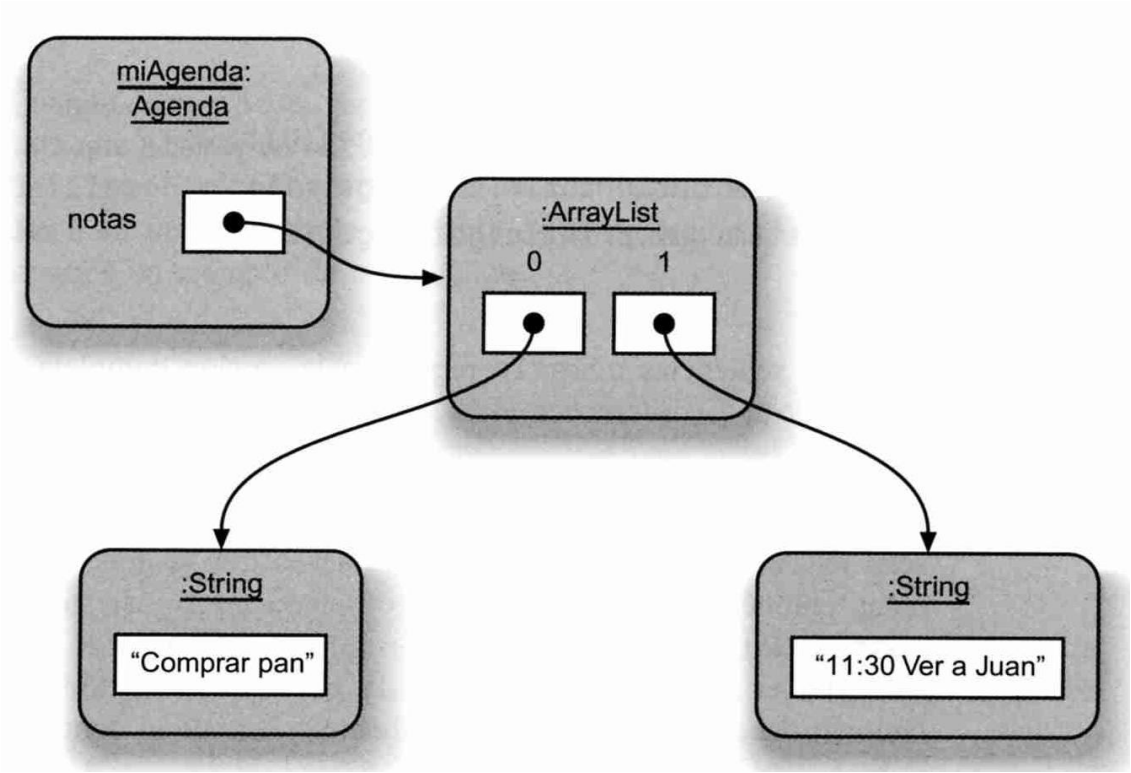
Cuidado: si usted no es cuidadoso, podría intentar acceder a un elemento de una colección que está fuera de los índices válidos del `ArrayList`. Cuando lo haga, obtendrá un mensaje del error denominado *desbordamiento*. En Java, verá un mensaje que dice *IndexOutBoundsException*.

Eliminar un elemento de la “colección”

```
public void eliminarNota(int numeroDeNota)
      {
            if(numeroDeNota < 0) {
                  // No es un número de nota válido, no
se hace nada.
            }
            else if(numeroDeNota < numeroDeNotas()) {
                  // Número de nota válido, se la puede
                     borrar.
                  notas.remove(numeroDeNota);
            }
            else {
                  // No es un número válido de nota,
                     entonces no se hace nada.
            }
      }
```

Modificación de los índices

Una complicación del proceso de eliminación es que se modifican los valores de los índices de las restantes notas que están almacenadas en la colección. Si se elimina una nota que tiene por índice un número muy bajo, la colección desplaza todos los siguientes elementos una posición a la izquierda para llenar el hueco; en consecuencia, sus índices disminuyen en 1.



4.9.1 El ciclo “for-each”

- Realiza el ciclo una vez por cada elemento de la colección

Un **ciclo** puede usarse para ejecutar repetidamente un bloque de sentencias sin tener que escribirlas varias veces.

```
for (TipoDelElemento elemento : colección) {
     cuerpo del ciclo
}
```

```
Para cada elemento en la colección hacer:   {
     cuerpo del ciclo
}
```

Define la variable de ciclo.
El tipo debe ser el mismo que el declarado en la colección

```
/**
* Imprime todas las notas de la agenda
*/
public void imprimirNotas()
{
      for(String nota : notas) {
            System.out.println(nota);
      }
}
```

```java
/**
 * Show a list of all the files in the collection.
 */
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

Proceso selectivo

```java
/**
 * List the names of files matching the given search string.
 * @param searchString The string to match.
 */
public void listMatching(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            // A match.
            System.out.println(filename);
        }
    }
}
```

4.10.1 El bucle“while”

```
while (condición del ciclo) {
    cuerpo del ciclo
}
```

Comparación con “for-each”

```
int indice = 0;
while(indice < notas.size()) {
   System.out.println(notas.get(indice));
   indice ++;
}
```

```
public void imprimirNotas()
{
      for(String nota : notas) {
            System.out.println(nota);
      }
}
```

Este *ciclo while* es equivalente al *ciclo for-each* que hemos discutido en la sección anterior. Son relevantes algunas observaciones:

- En este ejemplo, el *ciclo while* resulta un poco más complicado. Tenemos que declarar fuera del ciclo una variable para el índice e iniciarlo por nuestros propios medios en 0 para acceder al primer elemento de la lista.
- Los elementos de la lista no son extraídos automáticamente de la colección y asignados a una variable. En cambio, tenemos que hacer esto nosotros mismos usando el método `get` del `ArrayList`. También tenemos que llevar nuestra propia cuenta (índice) para recordar la posición en que estábamos.
- Debemos recordar incrementar la variable contadora (índice) por nuestros propios medios.

4.10.3 Búsqueda en una colección

Ejemplos

```
int numero = 0;
while (numero <= 30) {
      System.out.println(numero);
      numero = numero + 2;
}
```

```
int indice = 0;
boolean encontrado = false;
while (indice < notas.size() && !encontrado) {
      String nota = notas.get(indice);
      if (nota.contains(cadABuscar)) {
          encontrado = true;
      }
      else {
         indice++;
      }
}
```

```
System.out.println (n1 == n2);        //false
System.out.println (n1.equals(n2));   //true
```

4.12 El tipo “Iterator”

Un **iterador** es un objeto que proporciona funcionalidad para recorrer todos los elementos de una colección.

- Es una clase de tipo genérico, no define un tipo único
- Hay que indicarle el tipo
- Está definida en el paquete *java.util*; hay que importarla

```
import java.util.ArrayList;
import java.util.Iterator;
```

Un `Iterator` provee dos métodos para recorrer una colección: `hasNext` y `next`. A continuación describimos en pseudocódigo la manera en que usamos generalmente un `Iterator`:

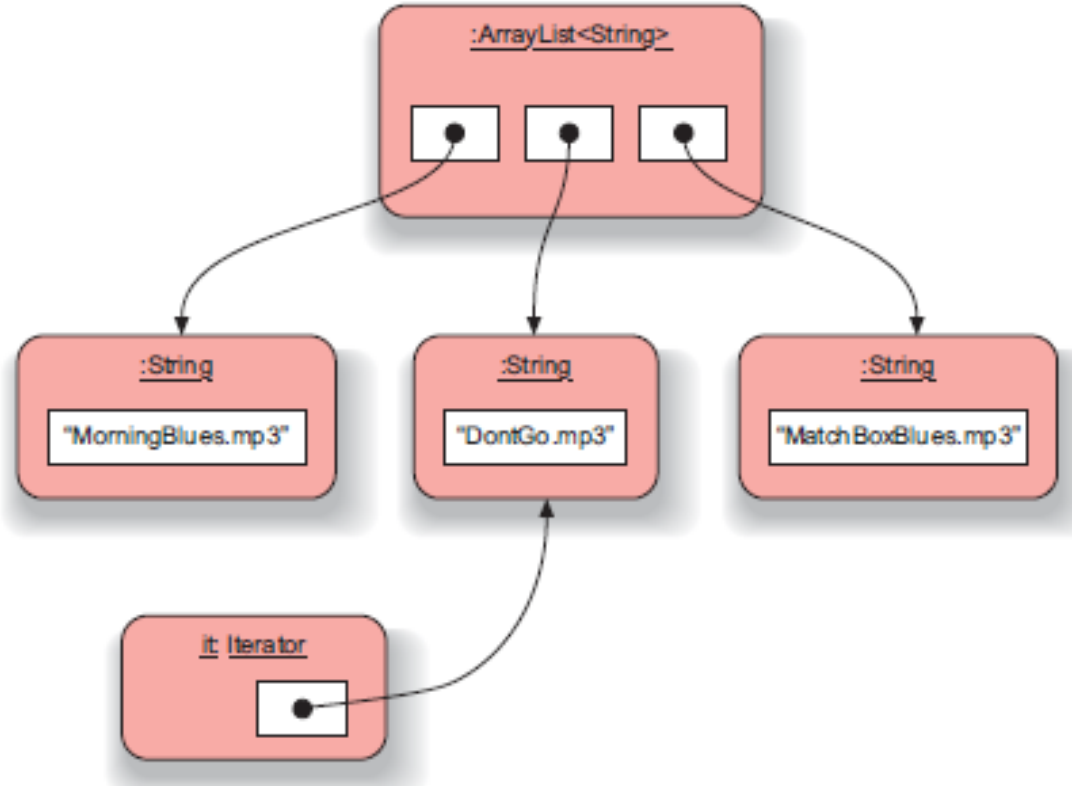
```
Iterator<TipoDelElemento> it = miColeccion.iterator();
while (it.hasNext()) {
  Invocar it.next() para obtener el siguiente elemento
  Hacer algo con dicho elemento
}
```

It.hasNext():
comprueba si hay mas elementos
It.next()
Obtiene el siguiente elemento

```
/**
  * Listar todas las notas de la agenda.
  */
public void listarTodasLasNotas()
{
    Iterator<String> it = notas.iterator();
    while (it.hasNext()) {
      System.out.println(it.next());
    }
}
```

```
int indice = 0;
while(indice < notas.size()) {
  System.out.println(notas.get(indice));
  indice ++;
}
```

```java
/**
 * List all the tracks.
 */
public void listAllTracks()
{
    Iterator<Track> it = tracks.iterator();
    while(it.hasNext()) {
        Track t = it.next();
        System.out.println(t.getDetails());
    }
}
```

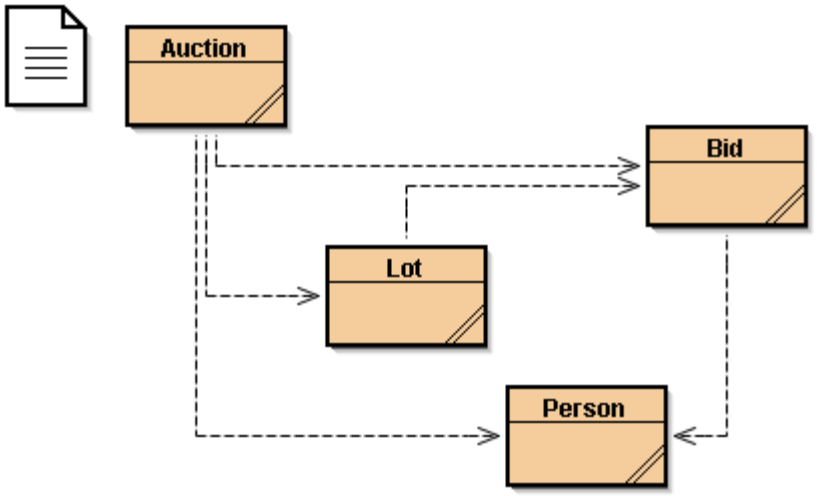


4.12 Eliminación de elementos

- No podemos eliminar un elemento de la colección en un bucle “for-each”.
- Tenemos que utilizar “Iterator”

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it.remove();
    }
}
```

Ejemplo: subasta



4.14.2 La palabra reservada “null”

Se usa la palabra reservada `null` de Java para significar que «no hay objeto» cuando una variable objeto no está haciendo referencia realmente a un objeto en particular. Un campo que no haya sido inicializado explícitamente contendrá el valor por defecto `null`.

```
/**
   * Attempt to bid for this lot. A successful bid
   * must have a value higher than any existing bid.
   * @param bid A new bid.
   * @return true if successful, false otherwise
   */
  public boolean bidFor(Bid bid)
  {
     if((highestBid == null) ||
           (bid.getValue() > highestBid.getValue())) {
        // This bid is the best so far.
        highestBid = bid;
        return true;
     }
     else {
        return false;
     }
  }
```

4.14.5 Objetos “anónimos”

```
/** clase auction (subasta)
    * Enter a new lot into the auction.
    * @param description A description of the lot.
    */
   public void enterLot(String description)
   {
       lots.add(new Lot(nextLotNumber, description));
       nextLotNumber++;
}
```

Aquí estamos haciendo dos cosas:

- Creamos un nuevo objeto `Lote` y
- Pasamos este nuevo objeto al método `add` de `ArrayList`.

Podríamos haber escrito lo mismo en dos líneas de código para producir el mismo efecto pero en pasos separados y más explícitos:

```
Lote nuevoLote = new Lote(numeroDeLoteSiguiente, descripcion);
lotes.add(nuevoLote);
```

Ambas versiones son equivalentes, pero si la variable `nuevoLote` no se usa más dentro del método, la primera versión evita declarar una variable que tenga un uso tan limitado. En efecto, creamos un objeto anónimo, un objeto sin nombre, pasándolo directamente al método que lo utiliza.

Colecciones de tamaño fijo: “arreglos” o “arrays”

- El acceso a los elementos de un arreglo es generalmente más eficiente que el acceso a los elementos de una colección de tamaño flexible.
- Los arreglos son capaces de almacenar objetos o valores de tipos primitivos. Las colecciones de tamaño flexible sólo pueden almacenar objetos[3].

Declaración de variables arreglos

```
private int[ ] contadoresPorHora;
```

La característica distintiva de la declaración de una variable de tipo arreglo es un par de corchetes que forman parte del nombre del tipo: `int[ ]`. Este detalle indica que la variable `contadoresPorHora` es de tipo *arreglo de enteros*. Decimos que `int` es el tipo base de este arreglo en particular. Es importante distinguir entre una declaración de una variable arreglo y una declaración simple ya que son bastante similares:

```
int hora;
int[ ] contadoresPorHora;
```

4.14.6 Encadenamiento de llamadas a métodos

- Modos alternativos

```
Bid bid = lot.getHighestBid();
Person bidder = bid.getBidder();
String name = bidder.getName();
System.out.println(name);

System.out.println(lot.getHighestBid().getBidder().getName());
```

```
lot.getHighestBid().getBidder().getName()
```

4.16 Colecciones de tamaño fijo

Concepto

Un arreglo es un tipo especial de colección que puede almacenar un número fijo de elementos.

- El acceso a los elementos de un arreglo es generalmente más eficiente que el acceso a los elementos de una colección de tamaño flexible.
- Los arreglos son capaces de almacenar objetos o valores de tipos primitivos. Las colecciones de tamaño flexible sólo pueden almacenar objetos[3].

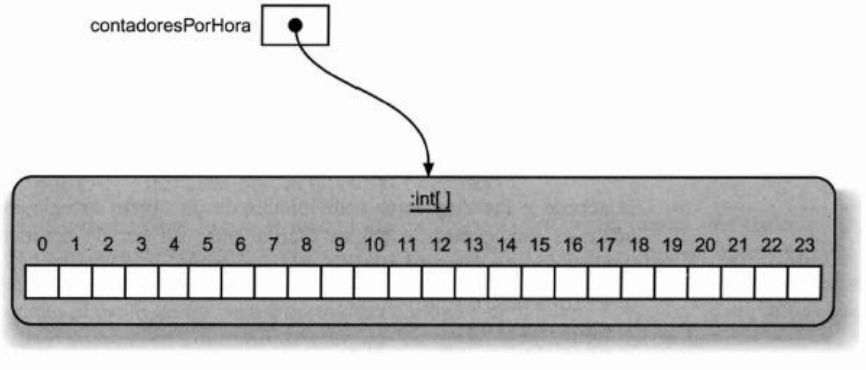
4.16.3 Creación de objetos “matriz”

La forma general de la construcción de un objeto arreglo es:

```
new tipo[expresión-entera]
```

La elección del tipo especifica de qué tipo serán todos los elementos que se almacenarán en el arreglo. La expresión entera especifica el tamaño del arreglo, es decir, un número fijo de elementos a almacenar.

```
int[ ] contadoresPorHora;
contadoresPorHora = new int[24];
```



En un solo paso

```
String [ ] nombres = new String[10];
```

4.16.4 Usar objetos “matriz”

Se accede a los elementos individuales de un objeto arreglo mediante un *índice*. Un índice es una expresión entera escrita entre un par de corchetes a continuación del nombre de una variable arreglo. Por ejemplo:

```
etiquetas[6];
maquinas[0];
gente[x + 10 - y];
```

Los valores válidos para una expresión que funciona como índice depende de la longitud del arreglo en el que se usarán. Los índices de los arreglos siempre comienzan por cero y van hasta el valor uno menos que la longitud del arreglo. Por lo que los índices válidos para el arreglo `contadoresPorHora` son desde 0 hasta 23, inclusive.

```
etiqueta[5] = "Salir";
double mitad = lecturas[0] / 2;
System.out.println(gente[3].getNombre());
maquinas[0] = new MaquinaDeBoletos(500);
```

4.16.6 El ciclo “for”

Un *ciclo for* tiene la siguiente forma general:

```
for (inicialización; condición; acción modificadora) {
   setencias a repetir
}
```

```
for(int hora = 0; hora < contadoresPorHora.length; hora++)
{
                    System.out.println(hora + ": " +
contadoresPorHora[hora]);
  }
```

Comparación con “while” y “for-each

```
int hora = 0;
while (hora < contadoresPorHora.length) {
 System.out.println(hora + ": " + contadoresPorHora[hora]);
 hora++
}
```

```
for(int valor : contadoresPorHora) {
  System.out.println(":   " + valor(;
}
```

4.16.8 El bucle for y los iteradores

```
for(Iterator<Track> it = tracks.iterator(); it.hasNext(); ) {
    Track t = it.next();
    if(t.getArtist().equals(artist)) {
        it.remove();
    }
}
```

¿Qué ciclo debo usar? Hemos hablado sobre tres ciclos diferentes: el *ciclo for*, el *ciclo for-each* y el *ciclo while*. Como habrá visto, en muchas situaciones el programador debe seleccionar el uso de alguno de estos ciclos para resolver una tarea. Generalmente, un ciclo puede ser rescrito mediante otro ciclo. De modo que, ¿cómo puede hacer para decidir qué ciclo usar en una situación? Ofrecemos algunas líneas guías:

- Si necesita recorrer todos los elementos de una colección, el *ciclo for-each* es, casi siempre, el ciclo más elegante para usar. Es claro y conciso (pero no provee una variable contadora de ciclo).
- Si tiene un ciclo que no está relacionado con una colección (pero lleva a cabo un conjunto de acciones repetidamente), el *ciclo for-each* no resulta útil. En este caso, puede elegir entre el *ciclo for* y el *ciclo while*. El *ciclo for-each* es sólo para colecciones.
- El *ciclo for* es bueno si conoce anticipadamente la cantidad de repeticiones necesarias (es decir, cuántas vueltas tiene que dar el ciclo). Esta información puede estar dada por una variable, pero no puede modificarse durante la ejecución del ciclo. Este ciclo también resulta muy bueno cuando necesita usar explícitamente una variable contadora.
- El *ciclo while* será el preferido si, al comienzo del ciclo, no se conoce la cantidad de iteraciones que se deben realizar. El fin del ciclo puede determinarse previamente mediante alguna condición (por ejemplo, lee una línea de un archivo (repetidamente) hasta que alcanza el fin del archivo).

Términos introducidos en este capítulo

colección, arreglo, iterador, *ciclo for-each*, *ciclo while*, *ciclo for*, índice, sentencia import, biblioteca, paquete, objeto anónimo

Resumen de conceptos

- **colecciones** Las colecciones de objetos son objetos que pueden almacenar un número arbitrario de otros objetos.
- **ciclo** Un ciclo se usa para ejecutar un bloque de sentencias repetidamente sin tener que escribirlas varias veces.
- **iterador** Un iterador es un objeto que proporciona funcionalidad para recorrer todos los elementos de una colección.
- **null** Se usa la palabra reservada de Java `null` para indicar que «no hay objeto» cuando una variable objeto no está haciendo referencia a un objeto en particular. Un campo que no ha sido inicializado explícitamente contendrá por defecto el valor `null`.
- **arreglo** Un arreglo es un tipo especial de colección que puede almacenar un número fijo de elementos.