Colecciones de Datos

Trabajando con colecciones de datos

Las colecciones de objetos son comunes en la programación moderna, y en especial bajo el paradigma de la Programación Orientada a Objetos. Tal como su nombre lo indica, las colecciones de objetos permiten colectar elementos de un mismo tipo y, si existen, de sus subtipos. Esto quiere decir, desde el punto de vista de la POO, que dentro de una colección podemos guardar en memoria objetos de una misma clase, pero también de sus subclases.

En Java, la forma más simple de trabajar con colecciones de objetos es utilizando la clase ***ArrayList*** que permite guardar en forma dinámica y secuencial un elemento detrás del otro, acceder a los mismos y eliminarlos cuando lo necesitemos.

Aquí veremos cómo crear colecciones de objetos utilizando ArrayList. también vermos cómo manipular su contenido a través de los siguientes métodos (hay más, pero por ahora no los trataremos):

add(elemento**)**	Permite agregar un elemento al final de la colección.
add(n, elemento)	Permite agregar un elemento en la posición señala por *n*. Si esta posición está fuera del rango aceptable (determinado por la cantidad de elementos de la lista) se producirá un error de ejecución.
get(n**)**	Permite obtener el elemento *n* de la colección. Si esta posición está fuera del rango aceptable (determinado por la cantidad de elementos de la lista) se producirá un error de ejecución.
remove(n**)**	Permite extraer el elemento *n* de la colección. Si esta posición está fuera del rango aceptable (determinado por la cantidad de elementos de la lista) se producirá un error de ejecución.

En Java los ArrayList son una implementación de la *interfaz **List***. También se los conoce como *listas*, aunque este concepto es más vasto de lo que trataremos aquí, donde sólo los utilizaremos para crear colecciones dinámicas de datos.

Especificando la clase de contenido de un ArrayList

Esta clase tiene una particularidad: para instanciar un ArrayList debemos indicar la clase de elementos que contendrá. Crear el ArrayList sin especificar la clase de elementos que contendrá está permitido, pero indicará una alerta y asumirá que sus elementos son de la clase ***Object***. Los ArrayLists soportan que en su definición se indique cuál será la clase que tomaremos como clase base para añadir elementos a la estructura.

Si necesitamos crear una colección de objetos que pueda guardar cualquier clase de elemento, debemos asociarla a la clase Object, pero la clase Object no conocerá los métodos y atributos específicos de cada elemento que pongamos en la lista, y esto

impedirá acceder a sus atributos y métodos particulares en forma automática. Más adelante veremos cómo reconocer y acceder a los atributos y miembros específicos de un elemento de una clase derivada.

Creación de un ArrayList

Como primera medida importaremos el paquete correspondiente a partir de la adición de su *namespace*:

```
import java.utils.ArrayList;
```

Una vez agregado el paquete nuestro programa permitirá crear colecciones de la siguiente manera (podemos hacerlo en dos partes, primero declarando la variable y luego creando el objeto; aquí está la "versión corta"):

```
ArrayList<ClaseBase> coleccion = new ArrayList<>();
```

Así, para crear una colección de objetos de la clase Persona, haremos:

```
ArrayList<Persona> personas = new ArrayList<>();
```

Tengamos en cuenta que al hacer esto estamos creando únicamente la colección vacía, sin ningún elemento. La clase ArrayList posee el método ***size()*** que nos indica cuál es la cantidad de elementos que se han guardado en el momento en la colección. Si ejecutásemos la siguiente instrucción inmediatamente después de crearla, obtendríamos como respuesta un 0:

```
System.out.println(coleccion.size());
```

En caso de necesitar crear una colección de elementos de los tipos nativos del lenguaje (integer, double, etc.) debemos recurrir a sus contrapartes como clase:

```
ArrayList<Integer> numeros = new ArrayList<>();
```

Tampoco deberemos crear una nueva clase para generar colecciones de Strings, porque de por sí ***String*** ya es una clase:

```
ArrayList<String> palabras = new ArrayList<>();
```

Agregar un elemento a un ArrayList

Para agregar un elemento a un ArrayList utilizaremos el método ***add(elemento)***, el cual permite agregar un nuevo elemento ***al final*** de la estructura (es decir, como último elemento). Hay forma de indicar específicamente dónde insertar el nuevo elemento, pero eso excede lo que hoy queremos aprender.

Por ejemplo, para agregar un elemento, podemos hacer:

```
// creamos el objeto
Objeto elemento = new Objeto();
...
...
...
// lo agregamos a la colección
coleccion.add(elemento);
```

Solamente en el caso de las clases relacionadas a los tipos nativos (integer, double, etc.) podremos agregar elementos sin necesidad de crear explícitamente el objeto (Java lo hará por nosotros):

```
numeros.add(19); // insertará el número 19 en la colección
```

Obtener el elemento n de un ArrayList

Para obtener el elemento *n* del ArrayList utilizaremos el método ***get(n)***, donde *n* es la posición en la cual está el elemento. El valor de *n* debe estar entre 0 y ***size()-1***, caso contrario se producirá un error de ejecución.

Para obtener el primer elemento de la colección haremos:

```
Objeto elemento = coleccion.get(0);
```

Esto guardará la *referencia* al elemento sin quitarlo de la lista (no hace una copia). Entonces, si modificamos el elemento que tenemos en el auxiliar, obviamente estaremos modificando el elemento que está en la lista, pues es el mismo elemento referenciado desde dos lugares distintos. Veremos esto con más detalle en la próximas materias.

Extraer el elemento n de un ArrayList

Para extraer el elemento *n* del ArrayList utilizaremos el método ***remove(n)***, donde *n* es la posición en la cual está el elemento. El tratamiento es similar al caso del ***get()***: el valor de *n* debe estar entre 0 y *size()-1*, caso contrario se producirá un error de ejecución. A diferencia del ***get()***, ***remove()*** extrae el elemento de la posición, y si éste no era el último, su lugar será ocupado por el siguiente elemento (y así hasta el final), acortando el tamaño de la estructura en 1.

Por ejemplo, para extraer el primer elemento de la colección haremos:

```
Objeto elemento = coleccion.remove(0);
```

o, directamente:

```
coleccion.remove(0);
```

La diferencia entre estos dos casos es que mientras el primero guarda el elemento removido en una variable auxiliar para su uso posterior, el segundo lo descarta inmediatamente y ya no tendremos acceso al mismo.

Pero sacar un elemento de la colección, como vimos más arriba, tiene algunas consecuencias impensadas, algunas de las cuales trataremos más adelante en esta misma guía.

Recorrer el ArrayList completo con for-each

El ciclo conocido como *for-each* es una variante del *for* tradicional que, en vez de utilizar un valor índice para iterar una determinada cantidad de veces, utiliza un *iterador* implícito para visitar en cada ciclo cada los elementos cargados en la lista, de a uno por vez, del primero al último elemento. Para eso necesita una variable local en la cual guardará temporalmente la referencia al elemento visitado.

La estructura del *for-each* es la que sigue:

```
for (Clase elementoAuxiliar : coleccion) {
   proceso(elementoAuxiliar);
}
```

En el ejemplo, proceso() deberá reemplazarse por lo que querramos hacer con el elemento visitado. Por ejemplo, para mostrar el nombre y el número de DNI de cada persona guardada en la colección, haríamos:

```
for (Persona persona : personas){
   System.out.println(persona.getNombre());
   System.out.println(persona.getDocumento());
}
```

Este ejemplo podría leerse como *"para cada persona que esté en la colección de personas, mostrar el nombre y el número de documento"*. Esto hará que en cada iteración (en cada vuelta del ciclo) la variable *persona* guarde temporalmente, una a una, cada persona que esté en la estructura, comenzando con la primera de todas y hasta que no haya más. La variable persona tendrá asignada en cada ciclo, entonces, el objeto de la colección que se está visitando en ese momento.

Buscar un elemento en la colección

Utilizar el ciclo *for-each* es muy fácil y cómodo, pero no tiene una manera natural de abandonar el recorrido por la colección sin recorrerlo por completo. El inconveniente de esto es que si estamos buscando un elemento específico y ya lo encontramos, o si ya

sabemos que no lo encontraremos, seguiríamos recorriendo la estructura hasta el final innecesariamente.

Hay formas de recorrer la estructura parcialmente. Una de ellas es utilizando explícitamente un *iterador*, la cual probablemente sea la forma más conocida de hacerlo. Pero trabajar con iteradores requiere de cierto cuidado y tener en cuenta algunos conceptos que dejaremos para más adelante.

Entonces nos queda una forma de recorrer la estructura parcialmente sin necesidad de hacerlo por completo: aprovechando la capacidad de acceder a un elemento determinado con ***get()***. Por ejemplo, si necesitamos ubicar una persona determinada de la lista, podríamos hacer lo siguiente:

```
// Búsqueda de la persona por nro de documento.
// creamos una variable auxiliar donde dejaremos
// el elemento buscado, si lo encontramos.
Persona personaBuscada = null;
int indice = 0;
// buscamos mientras no hayamos llegado al final
// de la estructura y no hayamos encontrado el
// elemento que estamos buscando.
while (indice < personas.size() && personaBuscada == null) {
   if (personas.get(indice).getDocumento() == docBuscado) {
         // si la encontramos, la asignamos al auxiliar
         personaBuscada = personas.get(indice);
   } else {
         // incrementamos el índice para
         // pasar al siguiente elemento
         indice++;
   }
}
```

El ejemplo anterior utilizará el ciclo *while* para recorrer la estructura mientras haga falta. El *if* interno, como está expresado en los comentarios, cumple dos tareas. La primera es asignar el elemento al auxiliar cuando es el que buscamos. La segunda, es darnos la posibilidad de intentar con el siguiente, a ver si es el que queremos. El incremento de índice puede hacerse también fuera del if (no en el else), aunque en ese caso perdería la posición donde encontró el elemento. De esta forma, guarda también la posición y no incremente el índice sin necesidad.

Remover algunos de los elementos del ArrayList

Un tratamiento similar debemos aplicar cuando queremos extraer algunos elementos de la colección, pero no todos. El tema, en este caso, pasa porque al remover un elemento determinado, los posteriores se adelantan en una posición. Esto hace que al incrementar el valor del índice inmediatamente después de extraer el elemento, como todos los elementos posteriores se movieron para adelante una posición, en la posición que acabamos de abandonar ha quedado, si existe, un elemento que no hemos visitado. Para que esto no suceda, sólo debemos avanzar cuando el elemento recién visitado no haya sido extraído. Si lo hemos extraído, nos quedaremos en esa posición esperando a que los

elementos se reacomoden (que todos los posteriores ocupen su nuevo lugar) y, así, no perder el nuevo elemento que cayó en la posición donde estamos parados.

Veamos el siguiente ejemplo: supongamos que tenemos una colección de números con los valores 1, 7, 4, 9, 17, 22. No es necesario que estén ordenados. Lo que intentaremos es dejar sólo los números impares, eliminando los pares:

```
// Eliminando todos los números pares de la colección.
int indice = 0;
// buscamos mientras no hayamos llegado al final
// de la estructura.
while (indice < numeros.size()) {
   if (numeros.get(indice) % 2 == 0) {
         // si es par lo quitamos
         numeros.remove(indice);
   } else {
         // incrementamos el índice para
         // pasar al siguiente elemento
         indice++;
   }
}
```

Así, recorreremos la colección de números hasta llegar al 4 (índice=2, recordemos que el primero tiene índice=0). Entonces, lo extraeremos, y todos los elementos posteriores se adelantarán un lugar. La colección, entonces, queda así:

```
1, 7, 9, 17, 22
```

Nosotros seguiremos *parados* en la posición 2, donde ahora, en vez del 4 que extrajimos, está el 9. Gracias a que no avanzamos, en la nueva iteración del ciclo (donde podría haber encontrado un nuevo número par) trataremos el 9, luego el 17 y así hasta llegar al 22. Aquí también nos detendremos a extraer el número y no incrementaremos, pero la lista habrá perdido un nuevo elemento (el 22), quedando entonces sólo con 4 elementos (1, 7, 9, 17). Como el ciclo sólo itera mientras el índice sea menor a la cantidad de elementos de la colección, saldremos y la lista quedará solamente con los números impares.

Trabajando con colecciones de objetos dentro de otros objetos

Uno de los usos más comunes de las colecciones es guardar listas de valores dentro de objetos contenedores: la lista de socios de un club, los goles hechos en un partido de fútbol, los ítems de una factura de compra, etc.

En estos casos, exponer directamente la colección suena a peligro, ya que exponer públicamente un atributo hace que no sepamos qué es lo que se hace con él fuera de la clase. Por lo general, necesitamos *ofuscar* u ocultar estas colecciones como atributos privados de la clase que los contiene, y proveer de métodos indirectos de acceso a la colección.

Veamos entonces algunas formas de proveer este acceso.

Creando colecciones como atributo de una clase

Supongamos que tenemos la clase *Club*, y que el club mantiene una lista de sus socios. Para esto, crearemos el atributo privado socios, que será un ArrayList de elemento de clase *Socio*. Si bien podemos instanciar aquí la colección, sólo la declararemos y le asignaremos el valor ***null*** para que la colección sea creada explícitamente en el constructor de la clase Club, pues consideramos que es el mejor lugar para hacerlo:

```
public class Club {
   private ArrayList<Socio> socios = null;

   public Club() {
         this.socios = new ArrayList<Socio>();
   }

   public void agregarSocio(Socio socio) {
         socios.add(socio);
   }

   public void agregarSocio(String nombre) {
         socios.add(new Socio(nombre));
   }
}
```

También podemos ver dos métodos distintos para agregar socios, el primero pasándole directamente un objeto de la clase Socio, y el segundo que sólo recibe el nombre del nuevo socio, que será instanciado y agregado en el momento.

¿Y qué deberíamos hacer para dar de baja un socio? ¿Cómo hacemos para devolver la instancia de socio que queremos eliminar sin exponer la colección completa? Tal vez podríamos hacer algo como lo que sigue:

```
public Socio darDeBaja(int numero) {
   Socio socio = null;
   int indice = 0;
   while (indice < socios.size() && socio == null) {
         if (socios.get(indice).getNumero() == numero) {
               socio = socios.get(indice);
         } else {
               indice++;
         }
   }
   return socio;
}
```

Así devolvemos null o el socio encontrado, que ya ha sido removido de la colección, para que desde donde han llamado a la rutina se procese la instancia de la manera que se desee (por ejemplo, para mostrar los datos del socio dado de baja).

Devolver una colección secundaria

Siguiendo con el ejemplo anterior, supongamos que queremos devolver una colección con todos los socios vitalicios del club. Para eso podríamos hacer algo como lo que sigue:

```
public ArrayList<Socio> obtenerVitalicios() {
   ArrayList<Socio> vitalicios = new ArrayList<Socio>();
   for (Socio s: socios) {
         if (s.esVitalicio()) {
               vitalicios.add(s);
         }
   }
   return vitalicios;
}
```

En este caso agregará a la colección *vitalicios* (creada como variable local) todos aquellos socios que cumplan con la condición de ser vitalicios.

Pero, ¿qué pasará, entonces, con la lista de socios principal, donde estaban los estos socios, ahora en la lista de socios vitalicios?

No pasará nada: todos los socios estarán en la lista principal, y aquellos que cumplan con lo condición deseada, además, estarán en esta lista secundaria. Cada socio vitalicio estará tanto en la lista secundaria como en la principal, porque en ambas se guarda la referencia al objeto de clase Socio en cuestión. En caso de eliminar el elemento de una de las listas, seguirá *vivo* en la otra. Esto puede no ser algo deseable, pero es algo que se tratará en otra oportunidad.

Conclusión

Hemos dado un recorrido al tratamiento de las colecciones de datos en Java, pero tengamos en cuenta que esto es sólo una parte de las cosas que podemos hacer. ya veremos en el futuro (en próximas materias) otras posibilidades.