Revisando conceptos: colecciones

El Java Collections Framework

En este punto, estamos familiarizados con las colecciones. La idea detrás del Java Collections Framework es tener una serie de interfaces que cualquiera pueda implementar. Es decir, el Framework nos da todas las operaciones posibles que podemos hacer con las List y los Set, agrupando todo a su vez, en una super-interfaz llamada **Collection**. Es decir, a partir de collection, podríamos nosotros implementar nuestra propia colección.

Supongamos que queremos "inventar" una nueva colección, podríamos llamarle "Circulo". En esta colección, podemos tener cero, uno o más elementos. Cuando hay más de un elemento, el último hace referencia al primero, de forma tal que, si lo recorremos, nunca terminaríamos… Parece ser medio intrincado y poco útil, pero es solo un ejemplo. Entonces, si respetamos la interfaz Collection:

```
public class Circulo implements Collection {
  // obligación de darle cuerpo a todos los métodos de Collection
}
```

¿Cómo es posible eso? Porque de eso se trata Collection y sus sub-interfaces. De poder tener **estructuras de datos polimórficas**. Tanto ArrayList como HashSet, como nuestro Círculo, al cumplir con la interfaz Collection pasan la prueba "es-un" (más bien, podríamos decir "se-comporta-como-un") Collection. De esta forma podríamos tener:

```
public class Test {
  public static void main (String [] args) {
    Collection c;
    System.out.println("Ingrese opcion");
    String opcion = new Scanner(System.in).nextLine();
    if(opción.equals("arraylist") {
      c = new ArrayList();
    } else If(opción.equals("hashset") {
```

```
      c = new HashSet();
    } If(opción.equals("circulo") {
      c = new Circulo();
    }
    c.add(new Rottweiler());
    c.add(new Husky());
    c.add(new Yorkie());
    System.out.println("Hay " + c.size() + " perros");
    System.out.println(c.clear(););
  }
}
```

Es decir, no importa qué elija el usuario aquí, **no importa qué implementación de Collection se use**, sino que lo que importa es que se usa una Collection. Por tanto, sea un HashSet, ArrayList o nuestro invento "Círculo", las operaciones add(), size() y clear() siempre van a estar allí, el código siempre va a compilar. Y no podría ser de otro modo: ArrayList y HashSet **respetan el contrato** que impone Collection; si queremos que nuestro invento "Circulo" funcione aquí, también debería.

La filosofía del Java Collections Framework es bastante sencilla: el framework provee interfaces que indican qué debe hacer una clase para considerarla Collection. Luego, especifica un poco esta interfaz para indicar qué debe hacer una clase para considerarse una List. A su vez, hay una serie de implementaciones de referencia, como ArrayList o HashSet que dicen cómo deben hacer lo que deben hacer. HashSet es una implementación de Set quien hace cumplir las restricciones que conocemos de los conjuntos (Set, como interfaz, no puede imponer estas restricciones, ¡sus métodos no tienen código!). Siguiendo el mismo tren de pensamiento, cuando programemos Círculo diremos cómo llevará a cabo los comportamientos que impone Collection, y allí deberíamos establecer en el código de sus métodos esta característica de que el último referencie al primero.

El hecho de usar interfaces hace que el Java Collections Framework posea un bajo grado de acoplamiento. Y eso es bueno. Si usamos Collection, List o Set, no debemos preocuparnos por cambios en el futuro, pues podríamos "intercambiar" implementaciones. Por ejemplo, digamos que el día de mañana se encuentra un bug de seguridad en ArrayList y debemos cambiar rápidamente a LinkedList. Todo nuestro código seguirá compilando y funcionando

como antes. Por eso, siempre, en colecciones y cualquier otra ocasión que podamos, debemos poner del lado izquierdo del igual a la interfaz:

```
Collection c = new ArrayList();
```

O en su defecto:

```
List l = new ArrayList();
```

De esta manera, dependeremos solo de los métodos que provean las interfaces y no de algún método muy específico que tenga una implementación determinada. Nunca es conveniente hacer:

```
ArrayList al = new ArrayList();
```

Porque si en algún punto nos resultó útil y utilizamos un método específico de la clase ArrayList, entonces no podremos intercambiar tan fácilmente de implementación. Puede que LinkedList no tenga ese método y nuestro código dejará de compilar.

Colecciones y Generics

Al momento de ver las colecciones vimos que, entre sus operaciones básicas, tenían métodos para agregar, recorrer y, en algunos casos, obtener elementos por posición, vaciar la colección, etc. Si prestamos atención a los tipos utilizados en estas operaciones, veremos que siempre se utiliza "Object".

- add(Object o): void
- get(int i): Object
- iterator(): iterator
- hasNext(): boolean
- next(): Object

Esto se debe a que, en las colecciones, a diferencia de los arreglos, no podemos definir el tipo de los elementos que contienen. En Java, todas las clases heredan, en último término, de la clase Object. Es decir, cualquier clase que utilicemos es-un "Object". Esto nos permitiría mezclar elementos en una colección. Auto es-un Object, Perro es-un Object. De hecho, hasta podríamos hacer un Object[] aunque hasta ahora vimos cómo hacer arreglos de un tipo bien específico, dado que facilitaba la tarea de la recorrida de los elementos y el acceso a sus datos.

Veamos, entonces, qué sucede con un arreglo:

```
Auto[] a = new Auto[4];
// poner un Auto en la posición 0 del arreglo de Autos
a[0] = new Auto();
//error de compilación, el arreglo es de Autos, no de Huskies
a[1] = new Husky();
```

Veamos lo que sucede en una colección, más precisamente una List. Supongamos una jerarquía de electrodomésticos. Cada electrodoméstico seguro tendrá operaciones "encender" y "apagar". Algunas de sus subclases pueden ser: Televisor, Refrigerador, Batidora, Licuadora, Microondas, etc.

Un Refrigerador, además de encender() y apagar(), puede tener una operación llenar() y vaciar(). Asumiendo que existen estas clases y sus operaciones correspondientes podríamos hacer una lista con los electrodomésticos:

```
List electros = new ArrayList();
// en la lista agrego un objeto Batidora
electros.add(new Batidora());
// al mismo tiempo agrego un objeto Licuadora
electros.add(new Licuadora());
// agreguemos a la lista un Refrigerador
electros.add(new Refrigerador ());
//Si en algún momento quisiera obtener algún bien de la lista:
Object o = electros.get(0);
Batidora a = (Batidora)o; // debo castear
Object o2 = electros.get(1);
Licuadora c = (Licuadora)o2; // debo castear
//agreguemos una tele
l.add(new Televisor());
```

Entonces, si quisiera recorrer la lista electrodomésticos y "encenderlos" a todos podríamos hacer esto:

```
for(int i=0; i<bienesDeLaEmpresa.size(); i++) {
     ((Electrodomestico)electros.get(i)).cargar();// hay que castear
}
```

¿Pero qué pasaría si queremos llenar el refrigerador?

```
for(int i=0; i < electros.size(); i++) {
    Electrodomestico e = (Electrodomestico) electros.get(i);
    if(e instanceof Refrigerador) { // preguntar antes de castear
        // hay que castear a Refrigerador
        ((Refrigerador)e).llenar();
    }
    e.encender();
}
```

Entonces, pareciera que es necesario castear siempre para encender y, además, solo en las posiciones en las que aparezca un Refrigerador debo hacer un casteo más específico para poder llenarlo. De otra forma, tendríamos un error de compilación (o peor, ¡de ejecución!).

Ahora bien, para evitar errores a futuro, puede que sea interesante no tener la posibilidad de "mezclar" electrodomésticos y poder tener colecciones de cada electrodoméstico que se puedan llenar/vaciar y colecciones de electrodomésticos que no. ¿De qué forma podríamos limitar el tipo de objetos que podemos agregar a las colecciones? Gracias a la programación paramétrica, en las colecciones se puede establecer de antemano un tipo "genérico" que se almacenará. Uno de los beneficios inmediatos es poder tener un chequeo de tipos en tiempo de compilación. Entonces, si definimos una colección de Refrigeradores, no podremos agregar una Batidora y viceversa. Esta característica se expresa mediante la sintaxis "<TipoGenerico>" y al momento de crear la colección sería:

```
List<Refrigerador> listaDeRefris = new ArrayList<Refrigerador>();
// alternativamente, puede omitirse el tipo. El compilador lo infiere
List<Batidora> abtidoras = new ArrayList<>();
```

Aquí, como el nombre de la variable lo indica, lo que construimos es una "lista de objetos tipo Refrigerador" o, dicho de otra forma, "una lista de refrigeradores". De esta manera, tenemos algo mucho más similar a lo que veníamos haciendo con los arreglos pero con las ventajas que nos dan las colecciones.

Si usamos esta construcción, tendremos un control en tiempo de compilación sobre los tipos de los objetos que agregamos a la colección, de forma tal que, al momento de correr el código, no hace falta chequear por el instanceof dado que no deberemos castear, ya que nunca podríamos "mezclar" tipos de objetos. Entonces, hacer una cosa así:

```
List<Refrigerador> refris = new ArrayList<>();
```

Nos da como resultado tener las siguientes operaciones:

- add(Refrigerador o): void
- get(int i): Refrigerador
- iterator(): Iterator<Refrigerador> y del iterator se desprenden
    - hasNext(): boolean
    - next(): Refrigerador

Por lo tanto, el código anterior quedaría:

```
List<Refrigerador> refris = new ArrayList<>();
//refris.add(new Televisor()); // << no compila!
for(int i=0; i < refris.size(); i++) {
    //el get siempre da un Refrigerador, entonces lo puedo llenar()
    bienesDeLaEmpresa.get(i).cargar();
    // y también encender!
    bienesDeLaEmpresa.get(i).encender();
}
```

Lo mismo ocurriría si utilizáramos cualquier otra Collection (Collection, List, Set) mediante un Iterator.

```
for(Iterator<Refrigerador> it = refris.iterator(); it.hasNext();) {
    //el next siempre da un Refrigerador, dado que el
    //Iterator "es de refrigeradores", entonces puedo llenar() cada uno
    it.next().llenar();
}
```

Entonces, la ventaja aquí se da en tiempo de compilación, por los chequeos de tipo que hace el compilador, y en tiempo de ejecución porque se reduce la posibilidad de obtener errores de casteo (ClassCastException).

Instrucción for…each

Hemos visto la construcción for…each en Java. La introducción de los tipos genéricos en Java y, gracias al uso de los Iterators, como vimos en la sección anterior, dieron lugar a una nueva forma de recorrer colecciones (y arreglos) desde Java 1.5 en adelante: el for…each. Es mucho más sencillo y corto de escribir que la iteración por índice o con iteradores y en código se ve así:

```
// Se lee "por cada Refrigerador en la lista de refrigeradores…"
for(Refrigerador r: refris) {
    r.llenar();
    r.encender();
}
```

Te invitamos a jugar con las colecciones. Intenta las diferentes implementaciones, podrás experimentar el poder de las interfaces intercambiando implementaciones. ¿Qué tal si haces alguna comparación de tiempos entre un ArrayList y un LinkedList? Intenta preguntar al usuario qué implementación utilizar. Luego, llena la lista con la misma cantidad de objetos y compara una y otra implementación. Puedes registrar el momento actual con System.currentTimeMillis().