4 | Colección abstracta

Meta

Que el alumno aplique la reutilización de código mediante el mecanismo de herencia e interfaces propuesto por el paradigma orientado a objetos en `Java`.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Escribir código general en una clase padre, sin conocer detalles sobre la implementación de las clases descendientes.
- Utilizar la definición, mediante una interfaz, de un tipo de dato abstracto, para programar funciones generales, sin conocer detalles sobre la implementación de los objetos que utiliza.

Antecedentes

Para reducir la cantidad de trabajo en las prácticas siguientes, se utilizarán las ventajas del paradigma orientado a objetos. Concretamente, se programará una biblioteca con varias estructuras de datos y las operaciones comunes a todas ellas serán implementadas en una clase padre. En esta práctica se trata de completar tantos métodos como sea posible programar eficientemente, aún sin haber programado ninguna de esas estructuras.

Para adquirir algo de habilidad creando código profesional, este paquete trabajará cumpliendo un conjunto de especificaciones dictadas por la API[1] de `Java`.

[1] Interfaz de programación de aplicaciones.

Actividad 4.1

Para familiarizarte con el ambiente de trabajo, revisa la documentación de las clases `Collection<E>` e `Iterator<E>` de `Java`.

Todas las estructuras que programaremos implementarán `Collection<E>`. No te preocupes, no duplicaremos la labor de las estructuras que ya vienen programadas en `Java`. La mayoría de nuestras estructuras ofrecerán características distintas a las versiones de la distribución oficial. Más aún, por motivos didácticos y ligeramente nacionalistas, nuestras clases tendrán nombres en español[2].

Actividad 4.2

Revisa la documentación del paquete `java.util`. ¿Qué estructuras de datos encuentras incluidas?

La primer clase a programar de nuestro paquete se llama `ColeccionAbstracta<E>` e implementa la intefaz `Collection<E>`. Todas nuestras estructuras heredarán de ella, por lo que el trabajo de esta práctica nos ahorrará mucho código en las venideras. Como `ColeccionAbstracta<E>` no sabe aún cómo serán guardados los datos, no podrá implementar todos los métodos de `Collection<E>`; de ahí que será de tipo `abstracto`. Para poder trabajar, hará uso del único conocimiento que tiene de estructuras tipo `Collection<E>`: que todas ellas implementan el método `iterator()`, que devuelve un método de tipo `Iterator<E>`.

Dado que el iterador recorre la estructura (sea cual sea ésta), otorgando acceso a cada uno de sus elementos una única vez, es posible implementar varios de los métodos de `Collection<E>` haciendo uso de este objeto.

Desarrollo

1. Crea una clase llamada `ColeccionAbstracta<E>` que implemente la interfaz `Collection<E>`, dentro del paquete `estructuras`.

2. Agrega un atributo con acceso `protected` para almacenar en él el número de elementos que tiene la estructura. Su valor inicial por defecto es cero, no lo modificarás en esta clase, pero lo modificarán más adelante sus clases herederas. En la clase `Conjunto<E>` se asume que esta variable se llama `tam`. Eres libre de usar otro nombre, pero si lo haces debes modificar la clase `Conjunto<E>` reemplazando todas las ocurrencias de la variable `tam` por el nombre que elegiste.

[2]Aunque los métodos seguirán teniendo nombres en inglés, pues así lo requieren las interfaces.

3. Implementa únicamente los métodos listados a continuación. Una sugerencia es que añadas todas las firmas de los métodos y hagas que devuelvan `0` o `null` para verificar que tu clase compile. Observa que la clase `Conjunto<E>` fue provista como ejemplo de clase hija. Una vez que agregues los métodos, `Conjunto<E>` deberá compilar sin problemas, esto será necesario para que las pruebas unitarias funcionen.
   - `public boolean isEmpty()`
   - `public boolean contains(Object o)`
   - `public Object[] toArray()`
   - `public <T> T[] toArray(T[] a)`
   - `public boolean containsAll(Collection<?> c)`
   - `public boolean addAll(Collection<? extends E> c)`
   - `public boolean remove(Object o)`
   - `public boolean removeAll(Collection<?> c)`
   - `public boolean retainAll(Collection<?> c)`
   - `public void clear()`
   - `public boolean equals(Object o)`
   - `public int hashCode()`
   - `public int size()`
4. Adicionalmente, sobreescribe el método `toString()` de la superclase `Object` para que la colección devuelva una cadena con todos los elementos almacenados en ella. Te será muy útil en el futuro para depurar tus colecciones mientras las programas.

Preguntas

1. ¿Qué estructuras de datos incluye la API de `Java` dentro del paquete que importas, `java.util`?
2. ¿Cuál crees que es el objetivo de la interfaz `Collection`? ¿Por qué no hacer que cada estructura defina sus propios métodos?
3. ¿Qué métodos permite la interfaz `Collection` que su funcionalidad sea opcional? ¿Qué deben hacer estos métodos opcionales si no se implementa su funcionalidad? ¿Por qué crees que son opcionales?