Collections 1° parte: Listas y Conjuntos

CESSI #ArgentinaPrograma #YoPrograma

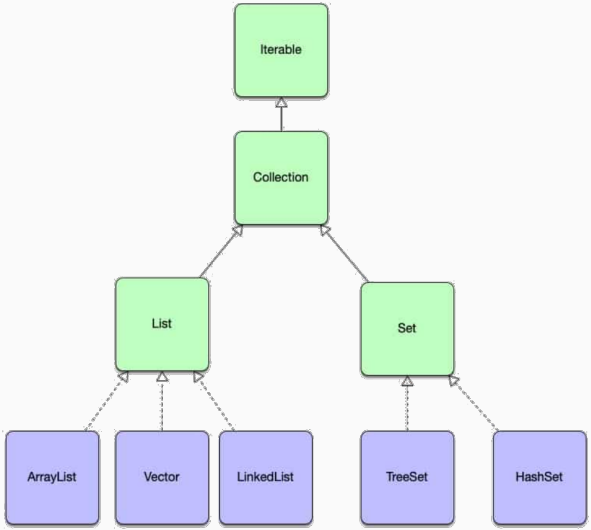
Leonardo Blautzik - Federico Gasior - Lucas Videla

Agosto -Diciembre de 2021

Estructuras de Datos

- Estructura de Datos: Es una representación de una colección de datos junto con las operaciones que se pueden hacer sobre ellos.
- Son reusables: se pueden usar para resolver distintos problemas.
- Para cada problema, distintas estructuras pueden tener diferentes tiempos de respuesta (eficiencia temporal).
- Comenzaremos usando estructuras ya provistas por Java para ver cómo se usan. No necesitamos saber cómo están implementadas para usarlas, aunque sí debemos saber cuál es la eficiencia de cada operación.

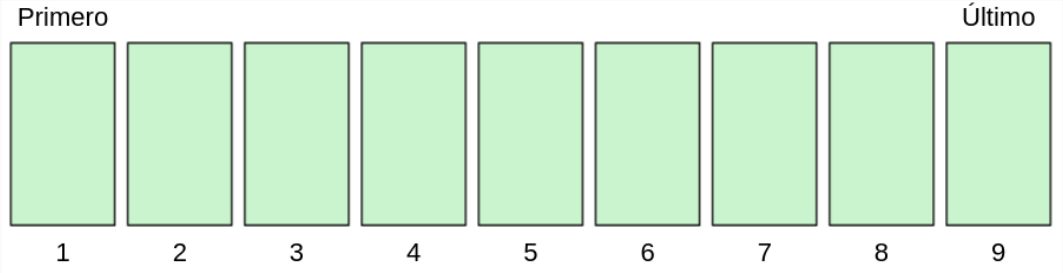
El framework de colecciones define la siguiente Jerarquía de clases



Listas

Datos organizados secuencialmente.

Por ejemplo: nombre y legajo de los alumnos presentes, ordenados según cómo están sentados, de izquierda a derecha (borrados si se van y actualizando las posiciones si se cambian de lugar).



Listas

Tipo `List<E>` en Java.

Para crear una lista (existen varias implementaciones):

```
List<String> alumnos = new LinkedList<String>();
List<String> alumnos = new ArrayList<String>();
```

La clase Lista en la API de Java presenta distintas implementaciones:

`LinkedList (Lista doblemente enlazada).`
`ArrayList (Lista sobre arreglos).`

List: Implementaciones

- **ArrayList:** Implementada sobre un array de tamaño variable. Presenta problemas de eficiencia para agregar o borrar elementos en posiciones intermedias (ya veremos por qué). ArrayList es muy eficiente cuando se pide el valor almacenado en una posición arbitraria.
- **LinkedList:** Implementada sobre una lista enlazada. Resuelve algunos de los problemas de eficiencia para agregar o borrar de ArrayList, pero es ineficiente al momento de pedir el valor almacenado en una posición arbitraria.

Algunas Operaciones Sobre Listas

- Agregar un elemento:

```
boolean add(int index, E element);
boolean add(E element);
```

- Borrar un elemento:

```
boolean remove(int index);
boolean remove(E element);
```

- Obtener un elemento:

```
E get(int index);
```

- ¿Contiene un elemento dado?

```
boolean contains(E element);
```

Iteradores

Se puede recorrer una lista usando un iterador. La interfaz básica `Iterator` permite recorrer hacia adelante una colección. El orden de una iteración sobre una lista es hacia adelante a través de los elementos. Un objeto `List` también soporta un `ListIterator`, el cual permite recorrer la lista hacia atrás, así como insertar o modificar elementos de la lista.

Uso de Iteradores:

```
List<Student> list = new ArrayList<Student>();
// se agregan varios elementos...

Iterator<Student> itr  = list.iterator()
while(itr.hasNext()){
    System.out.println(itr.next();
}
ListIterator<Integer> itr1 = lista.listIterator(0);
ListIterator<Integer> itr2 = lista.listIterator(lista.size());
```

La Herencia entre Iterator Interface e ListIterator Interface

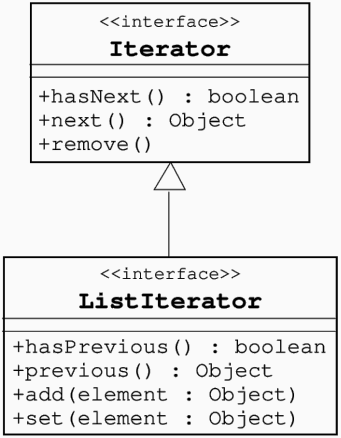


Figure 1: The Iterator Interface Hierarchy

Iterando con forEach

```
List<String> lista= new ArrayList<String>();
   lista.add("hola");
   lista.add("que");
   lista.add("tal");
   lista.add("estas");
   //con un for tradicional
   for (int i=0;i<lista.size();i++) {
     System.out.println(lista.get(i));
   }
   //con forEach
   for(String s :lista) {
     System.out.println(s);
   }
```

Set

La interfaz Set define una colección que **no puede contener elementos duplicados**. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos.

Para comprobar si los elementos son duplicados, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos equals y hashCode.

Set: Implementaciones

Dentro de la interfaz **Set** existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **HashSet** : este implementación almacena los elementos en una **Tabla Hash**. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.
- **TreeSet**: esta implementación almacena los elementos en un **Árbol Balanceado**. Los elementos almacenados deben implementar la interfaz Comparable. Esta implementación garantiza un rendimiento de log(N) en las operaciones básicas.
  Nota: compareTo() debe ser coherente con equals(), es decir: si `a.equals(b)` devuelve `true`, entonces `a.compareTo(b)` debe devolver 0.
- **LinkedHashSet**: esta implementación almacena los elementos en una **Tabla Hash+Lista Enlazada**.

Un ejemplo de Set

```java
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("one");
        set.add("second");
        set.add("3rd");
        set.add("4");
        set.add("5.0F");
        set.add("second");// duplicate, not added
        set.add("4"); // duplicate, not added "
        System.out.println(set);
    }
}
```

Consigna

1. Implementar un método estático getPersonas que reciba el nombre de un archivo y devuelva un objeto LinkedList con personas que fueron leídas del archivo de texto con formato “dni apellido edad”.
2. Implementar un método estático getPersonasMayoresAEdad que reciba un objeto LinkedList y una edad y devuelva otro objeto LinkedList con las personas cuyas edades son mayores a esa edad.
3. Sobreescribir los métodos: **equals** de Object para determinar que dos objetos personas son iguales si sus dni´s son iguales, **toString** para aplanar el objeto a una cadena que contiene los colaboradores internos del objeto separado por “,”.
4. Rehacer el punto 1. pero evitando cargar del archivo personas repetidas.
5. Generar el archivo personasMayoresdeX.csv con el resultado del punto 2. en dos versiones: ordenado por Apellido y ordenado por DNI.