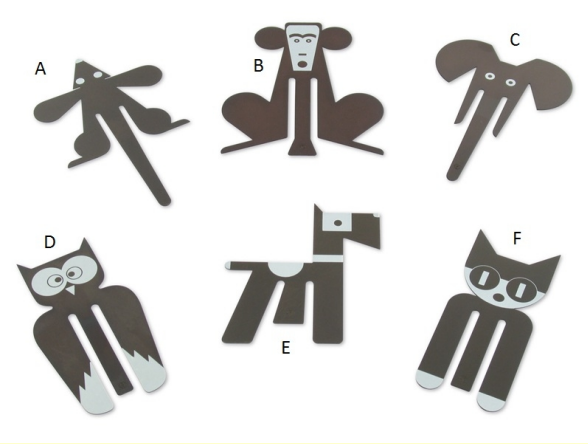
Colecciones en JAVA



<devAcademy>

Colecciones

- Java, ofrece dentro del paquete java.util, una serie de clases e interfaces que mejoran la eficacia y la eficiencia de las agrupaciones de datos respecto de los arrays primitivos
- Por ejemplo, no tenderemos que preocuparnos por el tamaño de nuestra colección de datos, ya que él sólo se va a adaptar dinámicamente, según las necesidades en tiempo de ejecución

Colecciones

- Las interfaces más importantes que debemos conocer son:
  - List: Colección de objetos con una secuencia determinada
  - Set: Colección de objetos que no admite duplicados
  - Map: Colección de objetos que mantiene asociados por un par clave-valor

Interface List

- Las clases más representativas que implementan la interfaz List, son:
  - Vector
  - ArrayList
  - LinkedList

Clases List

- Las principales diferencias entre un ArrayList, un Vector y una LinkedList, atienden a tres criterios
  - Sincronización
  - Tamaño incremental
  - Operatividad

Vector vs ArrayList

- Ambos son un array que crece dinámicamente. Por defecto, ambas reservan un espacio para 10 objetos
- Podemos indicar al instanciar objetos Vector y ArrayList, el tamaño inicial deseado, ya que tienen constructores que nos permiten determinarlo

  - **Vector**(int initialCapacity)
  - **ArrayList**(int initialCapacity)

Vector vs ArrayList

- Cuando la estructura está llena, el Vector, crece al doble de su capacidad y el ArrayList al triple.

- Para el Vector, además, podemos determinar el número de incremento que deseamos, para optimizar el uso de la memoria, mediante el constructor
  - Vector(int initialCapacity, int capacityIncrement)

Vector vs ArrayList

- Es importante, para ganar en rendimiento, que controlemos el modo de crecimiento. Si usamos un Vector, con el constructor, podemos definir el nº incremental
- Si es con ArrayList, mejor que otorguemos en inicio un número lo suficientemente grande, para no ocupar memoria innecesariamente (al crecer x3)

Vector vs ArrayList

- La principal diferencia entre ambos es que un objeto Vector, es sincronizado, y el ArrayList no lo es. Si un objeto es sincronizado, Java se va a encargar de que sólo un hilo va a poder acceder al mismo tiempo al objeto

- Por tanto, si necesitamos trabajar con una estructura que puede ser atacada por varios hilos y nos interesa controlarlo, usaremos un Vector; si no, un ArrayList, porque mantener la sincronización, “sale caro”

ArrayList vs LinkedList

- La prinicipal diferencia entre estos objetos, es el rendimiento entre los operadores add y get
  - Get // para obtener un elemento
  - Add // para insertar un elemento

ArrayList vs LinkedList

- Get (index) A la hora de acceder a un elemento, la complejidad en una LinkedList es O(n) , mientras que en un ArrayList, es O(1). O sea, que será mucho más rápido el acceso en un ArrayList que en una LinkedList

ArrayList vs LinkedList

- Add (element) A la hora de añadir un elemento, en un ArrayList, en el peor de los casos, tendremos una complejidad O(n) (cuando la estructura está llena y toca redimensionarla), mientras que en una LinkedList, siempre será O(1).

Resumen List

- Si tenemos la necesidad de buscar y acceder a los elementos, lo mejor es usar un ArrayList
- Si crear y borrar va a ser lo que predomine, lo mejor será usar una LinkedList
- Si necesitamos acceso sincronizado, Vector es la mejor opción

Ejemplo de uso ArrayList

```
ArrayList<Integer> al =newArrayList<Integer>();
al.add(3);
al.add(2);
for (Integer miembro : al) {
System.out.println(miembro);//for each, válido para recorrer Colecciones
}
```

Práctica 1

- Redefinir la clase Main de peronas, e ignorar la clase ListaPersonas, para usar la ArrayList<Personas>

  Jugar con los métodos de add(), remove(), Usar la estrucutra for each para recorrer los miembros de las listas

Interface Iterator

- Todas las clases vistas, implementan la interfaz Iterable (ya que su clase padre, Collection, implementa dicha interfaz. Decimos, entonces, que la colección es “recorrible” y ello implica, que tenga su propio método iterator().
- Este método, nos devuelve un objeto que implementa la interfaz Iterator

Interface Iterator

- Iterator, sirve para recorrer una estructura y define los siguientes métodos
  - boolean hasNext() //¿quedan elementos?
  - next() //dame el siguiente
  - remove() //borra el actual

Interface Iterator

- De este modo, encapsulamos la estructura, del modo de recorrerla, tarea que se reduce a algo sencillo:

```
Iterator<Personas> i_personas =
    lista_personas.iterator();
while (i_personas.hasNext()) {
 Persona p = i_.next(); . . . }
```

Práctica 2

- Hacer uso del método iterator() que nos ofrecen las clases ArrayList, LinkedList y Vector, para recorrer las colecciones de objetos de forma alternativa a la hecha con el for-each

Práctica avanzada

- Hacer que ListaPersonas implemente Iterable y desarrollar una clase que implemente un Iterator sobre personas

Interface Map

- Las clases más importantes que implementan está interfaz, son:

  - HashMap <Clave, Valor>
  - TreeMap<Clave, Valor>
  - LinkedHashMap<Clave, Valor>

Interface Map

- Ninguna de estas clases, permiten la inserción Clave duplicada. Si el nuevo objeto presenta una clave existente, el nuevo objeto reemplaza al viejo

- Y el acceso es no sincronizado (varios hilos pueden acceder a él simultáneamente)

Interface Map

- Los métodos más empleados son
  - Valor put (Clave, Valor)
  - Valor get (Clave)
  - Valor remove(Clave)
  - void clear()
  - boolean containsValue (Value)

Clase HashMap

- Los elementos no tienen un orden específico

Clase TreeMap

- La característica del TreeMap, es que ordena por si solo la colección, por la clave dada.

- Al mantener la colección ordenada, el el borrado y la inserción son algo más costosos

Clase LinkedHashMap

- Igual que HashMap ,pero a la hora de recorrer -iterar- sobre un LinkedHashMap, el orden de inserción es respetado

Práctica 3

- Crear Maps un de personas, dando como clave el orden de inserción
- Crear un TreeMap de alumnos, usando como clave la nota de cada alumno

Interfaz Set

- Basado en la definición matemática de conjunto (SET), no se permiten elementos (objetcs) duplicados

- No confundir con Map, donde lo que no se permiten son claves (keys) duplicadas

HashSet

- Al insertar, se comprueba que el nuevo elemento no existe ya en la colección
- El orden de inserción, no se corresponde con el de iteración

HashSet

- En realidad, HashSet<Elemento> es sólo un Wrap sobre HashMap<Elemento, Elemento>
- El propio objeto insertado en un Set, es usado como clave

TreeSet

- Los elementos insertados son ordenados.
- Además, por ser un Set, no te permite que añadas un Elemento, que cuya comparación equals con cualquier elemento, de verdadero

LinkedHashSet

- Igual que HashSet, pero el orden de inserción, es respetado (el primero insertado, será el primero de la lista, el segundo, el segundo, etc.)

Clase Collections

- Es una clase que ofrece métodos estáticos y que recibe, opera y devuelve colecciones (clases que implementaban la interfaz java.util.Collection)

Clase Collections

- Cuenta con métodos para determinar el orden de los elementos de una colección
  - Sort (Colection) //nos devuelve una colección ordenada por “su orden natural”
  - Shuffle(Colection) //nos devuelve la colección ordenada aleatoriamente

Interfaz Comparable

- Cuando nos referimos al orden natural, Java ordena los tipos primitivos directamente: 1>2, a>b, etc..
- Pero si nosotros queremos determinar el “orden natural” de una clase, debemos hacer que la clase a ordenar implemente la interfaz Comparable

Interfaz Comparable

- Por implementar Comparable, tendrá que sobreescribir el método:

  - int compareTo(Objeto), pudiendo devolver

    0 si son iguales

    -1 si el objeto pasado es mayor

    1 si el objeto pasado es menor

Interfaz Comparable

- Impementada Comparable, cuando un invoquemos el método de Collections.sort(listaPersonas), nos devolverá la lista ordenada según el criterio implementado en el método compareTo

Interfaz Comparator

- Es una interfaz muy parecida a la anterior, salvo que la cuestión del orden, queda definido fuera de la clase que queremos ordenar.
- Se habla de “orden total” y no de orden natural

Interfaz Comparator

- Para trabajar con esta forma de ordenada, debemos hacer una clase que implemente Comparator y en ella el método
  - int compare (Objeto 1, Objeto 2)

Interfaz Comparator

- Una vez definida nuestra clase que implementa Comparator, si queremos utilizarla para ordenar, debemos usar el método sort, sobrecardo de esta manera:

  - sort (Collection, Comparator)

Práctica 4

Se pide, definir un orden natural de personas por el orden alfabético de su nombre, usando para ello Comparable

Definir un orden total por edad, usando para ello Comparator

Usar Collections.sort para probar las dos soluciones y mostrar además también, una versión de orden aleatorio, haciendo uso del método shuffle.

APÉNDICE

Java no permite insertar el elemento en un conjunto, si el elemento ya está presente. Y en un Mapa, el objeto nuevo, reemplazará al antiguo si la clave del nuevo, estaba ya presente.

¿Pero cómo internamente interpreta Java si una clave está ya presente?

APÉNDICE

Para comprobar si una clave o elemento (definitivamente, un objeto), existe en una colección, Java hace lo siguiente:

Para cada objeto, entra en su hashcode(), si el hashcode() devuelve lo mismo, llama a equals() para desempatar. Si equals también devuelve true, es que existe. Monográficamente:

Existe si (hashcode() && equals()) para algún objeto

APÉNDICE

Atendiendo a esto, es posible que nos interese sobreescribir o no los métodos hashcode() o equals() del objeto empleado como clave

APÉNDICE

El método hascode() es empleado para acceder a un objeto albergado en un Set o en un Map. Por defecto, el hashcode de un objeto, es su dirección de memoria (basado en identidad : implementación por defecto de Object)

APÉNDICE

Si implementamos el hashcode() dependiendo del estado de un objeto, podríamos perder la referencia al objeto, al modificar su valor y por tanto, su hashcode

APÉNDICE

Los consejos para sobreescribir hashcode y equals consistentes están en la propia API