APRENDERAPROGRAMAR.COM

INTERFACE COLLECTION DE JAVA.UTIL DEL API JAVA. EJERCICIOS Y EJEMPLOS RESUELTOS. MÉTODOS ADD, REMOVE, SIZE. STREAMS (CU00917C)

Sección: Cursos
Categoría: Lenguaje de programación Java nivel avanzado I

Fecha revisión: 2039

Resumen: Entrega nº17 curso "Lenguaje de programación Java Nivel Avanzado I".

Autor: Manuel Sierra y José Luis Cuenca

INTERFACE COLLECTION

A continuación vamos a proceder a describir de manera detallada todas las interfaces y clases que consideramos más relevantes del paquete java.util y que hemos introducido previamente. Para ello comenzaremos por la raíz de todas ellas que es la interface Collection.

COLLECTION

Esta interfaz es "la raíz" de todas las interfaces y clases relacionadas con colecciones de elementos. Algunas colecciones pueden admitir duplicados de elementos dentro de ellas, mientras que otras no admiten duplicados. Otras colecciones pueden tener los elementos ordenados, mientras que en otras no existe orden definido entre sus elementos. Java no define ninguna implementación de esta interface y son respectivamente sus subinterfaces las que implementarán sus métodos como son por ejemplo las interfaces Set o List (que son subinterfaces de Collection).

Una colección es de manera genérica un grupo de objetos llamados elementos. Esta interfaz por tanto será usada para pasar colecciones de elementos o manipularlos de la manera más general deseada.

En resumen la idea es la siguiente: cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos.

En Java, se emplea la interface genérica Collection para este propósito. Gracias a esta interface, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección… ya que se trata de métodos definidos por la interface que obligatoriamente han de implementar las subinterfaces o clases que hereden de ella.

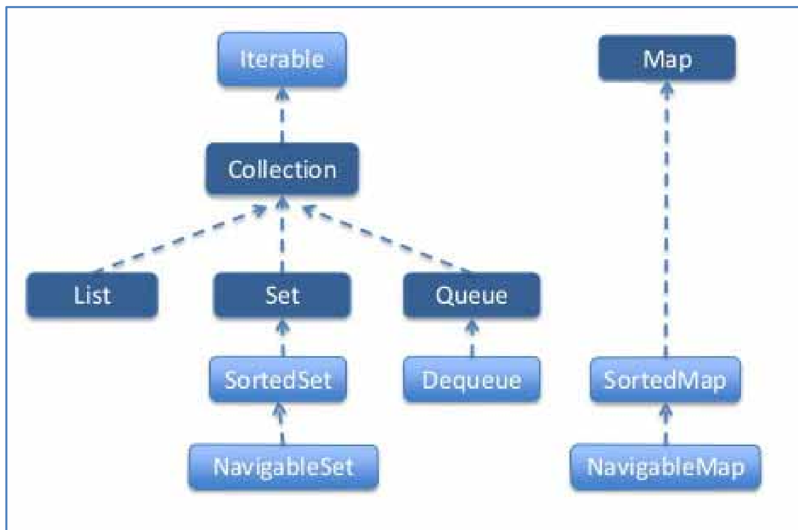
Collection se ramifica en subinterfaces como Set, List y otras que verémos en profundidad a lo largo del curso.

TIPOS DE COLECCIONES

Como estudiaremos a lo largo del curso, Java proporciona una serie de estructuras (intefaces, clases…) muy variadas para almacenar datos.

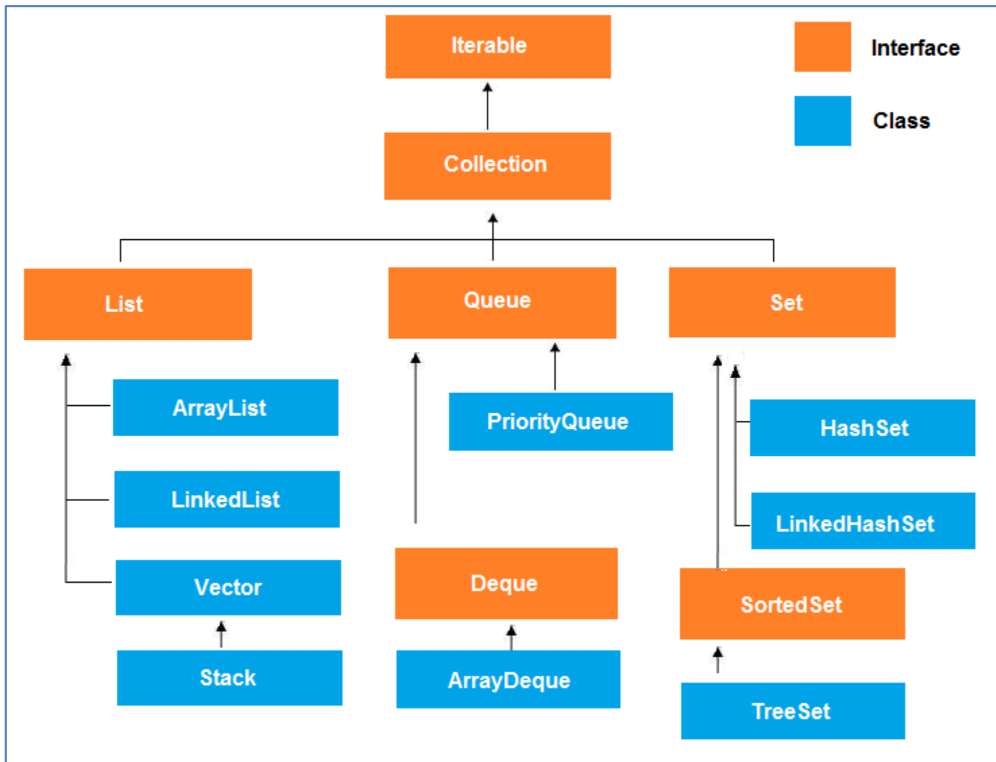
El siguiente esquema resume las interfaces y clases más usadas para colecciones de datos:

© aprenderaprogramar.com, 2006-2039



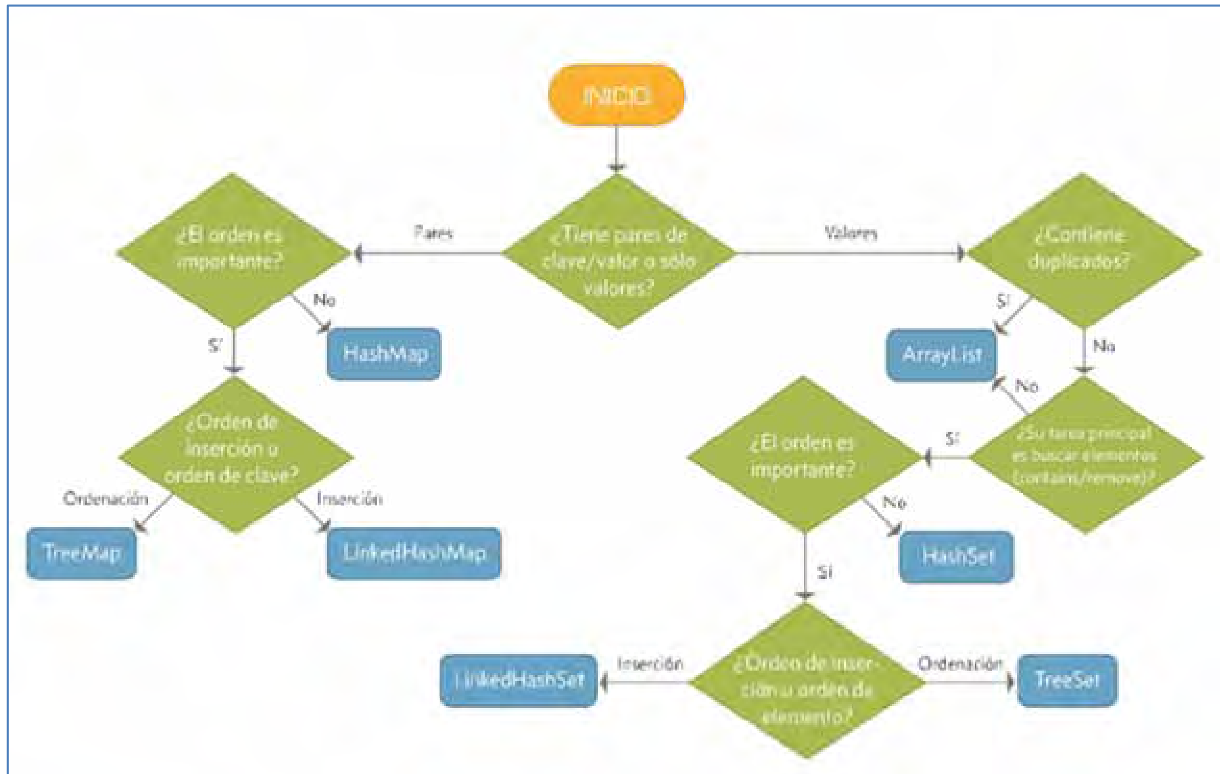
Un tipo de colecciones de objetos en Java son los Maps. A pesar de que estas estructuras denominadas "mapas" o "mapeos" son colecciones de datos, en el api de Java no derivan de la interface Collection. No obstante, muchas veces se estudian conjuntamente junto a las clases e interfaces derivadas de Collection.

El siguiente esquema resume cómo se organiza la interface Collection y sus derivaciones:



Estas estructuras ofrecen diversas funcionalidades: ordenación de elementos o no, mejor rendimiento para la ordenación para la inserción, tipos de operaciones disponibles, etc. Es importante conocer cada una de ellas para saber cuál es la estructura más adecuada en función de la situación en que nos

encontremos. Por ejemplo si necesitamos realizar miles de búsquedas al día es importante elegir una estructura que tenga buenos rendimientos para búsquedas. Si necesitamos realizar miles de inserciones manteniendo un orden es importante usar una estructura que tenga buenos rendimientos para inserciones. Un buen uso de estas estructuras mejorará el rendimiento de nuestra aplicación. Para conocer qué tipo de colección usar, podemos emplear diagramas de decisión similares al siguiente:



MÉTODOS DEFINIDOS EN LA INTERFAZ COLLECTION

Como cualquier interface en Java, Collection nos obliga a implementar varios métodos de los que queremos destacar los siguientes:

- Boolean add (E e)
- Boolean remove (Object o)
- Int size()

El primer método añade el elemento de la clase E a la colección, el segundo eliminaría el objeto o de la colección y el tercero nos devolvería el tamaño de la colección. Queda claro por tanto que la interface Collection sirve para trabajar con colecciones de objetos, no de tipos primitivos.

Hay más métodos que podemos observar consultando la documentación pero por simplicidad hemos querido destacar tan solo estos.

© aprenderaprogramar.com, 2006-2039

STREAMS

En primer lugar indiquemos que en Java existen clases del paquete Java I/O denominadas InputStream, OutputStream y otras con nombres similares derivadas de estas. Sin embargo, el término Stream en las últimas versiones de Java tiene otro significado adicional, que no tiene nada que ver con estas clases. Los Streams en las últimas versiones Java son un nuevo soporte para las operaciones con datos.

No vamos a ver ahora los Streams en profundidad, pero sí vamos a hacer una pequeña incursión porque los Streams están muy relacionados con las colecciones. Con la aparición de la API Stream las colecciones han aumentado su funcionalidad. Los streams nos dan la posibilidad de realizar operaciones funcionales sobre los elementos de las colecciones, siguiendo la filosofía de la denominada "programación funcional".

No queremos entrar en detalle ahora con esto. Vamos a ver un ejemplo de la API Stream sólo para hacernos una idea. Supongamos que queremos iterar sobre una lista de números enteros y averiguar la suma de todos los números enteros mayores de 10.

A continuación presentamos cómo puede resolverse esta tarea de la forma tradicional (usando un iterador, código en el lado izquierdo) o mediante el uso de Streams (código en el lado derecho).

```
import java.util.*; /* Ejemplo Streams aprenderaprogramar.com */
public class  TestDeEnteros  {
  public static void main(String[] args) {
      ArrayList<Integer> miListadoAL;
      miListadoAL = new ArrayList<Integer>();
      miListadoAL.add(44); miListadoAL.add(7); miListadoAL.add(16);
      miListadoAL.add(29); miListadoAL.add(8);  miListadoAL.add(50);
      System.out.print("La suma de elementos mayores de 10 es: ");
```

```
Iterator<Integer> it = miListadoAL.iterator();
      int sum = 0;
      while (it.hasNext()) {
         int num = it.next();
         if (num > 10) {
            sum += num;
         }
      }
```

```
      System.out.print(
      miListadoAL
      .stream()
      .filter(i -> i > 10)
      .mapToInt(i -> i).sum()
      );
   }
}
```

Ten en cuenta que el código que usa Streams puede no funcionar en versiones antiguas de Java.

El uso de Streams puede tener ventajas. Por ejemplo en muchos casos reduce considerablemente la cantidad de codigo; genera resultados a través de una serie de operaciones sobre datos sin modificar estos datos; y permite optimizar procesos para reducir el consumo de recursos (como memoria).

No os preocupeis si ahora no lo veis del todo claro. Esto es solo una primera toma de contacto con los streams. Los Streams merecen ser estudiados con mayor profundidad. De momento, lo único que debemos saber es que hay una forma adicional de operar sobre colecciones distinta a la forma tradicional: el uso de Streams.

© aprenderaprogramar.com, 2006-2039

EJEMPLO Y EJERCICIO RESUELTO

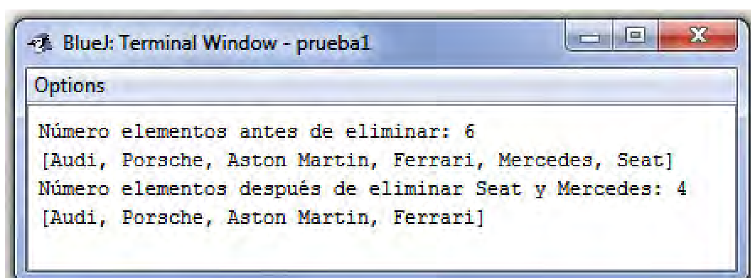
Pasamos ahora a la acción y vamos a escribir nuestro primer programa Java utilizando colecciones de elementos de forma tradicional. Para ello antes de ver otras colecciones más avanzadas vamos a utilizar la clase ArrayList que ya se ha visto en varias ocasiones anteriormente y debemos por tanto conocer. Como Collection es la raíz de todas las interfaces de colecciones de elementos, entonces ArrayList tiene que tener implementados los métodos de esta interfaz obligatoriamente y utilizaremos los métodos add, remove y size que hemos destacado previamente.

Para ello vamos a crear un objeto ArrayList en el cual insertaremos objetos de la clase String. Luego eliminaremos algunos y finalmente llamaremos al método size para que nos devuelva el tamaño de la colección de elementos de la clase String.

```
/* Ejemplo Interfaz Collection aprenderaprogramar.com */
import java.util.ArrayList;
import java.util.Collection;

public class Programa {
   public static void main(String arg[]) {
      Collection listaMarcasCoches = new ArrayList<String>(); // El tipo de listaMarcasCoches es Collection
      listaMarcasCoches.add("Audi");
      listaMarcasCoches.add("Porsche");
      listaMarcasCoches.add("Aston Martin");
      listaMarcasCoches.add("Ferrari");
      listaMarcasCoches.add("Mercedes");
      listaMarcasCoches.add("Seat");
      System.out.println("Número elementos antes de eliminar: " + listaMarcasCoches.size() ) ;
      System.out.println (listaMarcasCoches.toString() ) ;
      listaMarcasCoches.remove ("Seat");
      listaMarcasCoches.remove ("Mercedes");
      System.out.println("Número elementos después de eliminar Seat y Mercedes: 
" + listaMarcasCoches.size() ) ;
      System.out.println(listaMarcasCoches.toString () );
      }
}
```

Tras ejecutar el programa en BlueJ, tenemos la siguiente salida:



© aprenderaprogramar.com, 2006-2039

CONCLUSIONES

Podemos observar el uso de la interfaz Collection claramente en el anterior ejemplo. Al ser la raíz de casi todas las interfaces de colecciones es la más sencilla y por tanto la que menos métodos tiene. Utilizamos como comentamos anteriormente la implementación de ArrayList por ser también sencilla y por ir avanzando poco a poco sentando las bases para seguir avanzando posteriormente.

En el ejemplo introducimos 6 elementos en nuestra colección que fueron las cadenas de texto "Audi", "Porsche", "Aston Martín", "Ferrari", "Mercedes" y "Seat" para posteriormente eliminar 2 (Seat y Mercedes) quedándonos por tanto en la colección con 4 elementos.

¿Qué diferencia existe entre declarar el objeto listaMarcasCoches como Collection, List ó ArrayList? Fíjate que estas tres posibilidades existen, ya que la sentencia new **ArrayList**<String>(); crea un objeto de tipo ArrayList que es al mismo tiempo de tipo ArrayList, List y Collection debido al polimorfismo de Java. La diferencia estriba en que declarando el objeto listaMarcasCoches como Collection, en dicho objeto se puede almacenar cualquier objeto que cumpla con la interface. Por tanto ese objeto podría pertenecer a clases como ArrayList, HashSet, LinkedList, Stack, TreeSet, Vector, etc. En cambio, cuanto más restringida sea la declaración, menos posibilidades de tipos admite el objeto. Por ejemplo si se declara como tipo List admite menos tipos que si se declara como tipo Collection. Y finalmente tendríamos el caso de máxima restricción, que sería declararlo como tipo ArrayList, en cuyo caso el único tipo admitido sería el propio ArrayList.

El uso del polimorfismo nos permite desarrollar programas donde un mismo objeto admite distintos tipos, y esto puede resultar ventajoso para elegir el tipo más adecuado en función de las circunstancias o no tener que realizar cambios que afecten a todo el programa si queremos cambiar la clase específica en que está instanciado un objeto.

Las últimas versiones de Java incorporan una nueva forma de operar sobre colecciones que en algunos casos puede resultarnos de interés: el uso de Streams.

Próxima entrega: CU00918C

Acceso al curso completo en aprenderaprogramar.com -- > Cursos, o en la dirección siguiente:
http://aprenderaprogramar.com/index.php?option=com_content&view=category&id=58&Itemid=180

© aprenderaprogramar.com, 2006-2039