# Colecciones Genéricas

1.- Introducción

...

2.- Panorama de las Colecciones

Una *colección* es una estructura de datos, en realidad, un objeto, que puede contener referencias a otros objetos. Por lo general, las colecciones contienen referencias a objetos de cualquier tipo que tienen la relación *es-un* con el tipo almacenado en la colección.

Las interfaces del *marco de colecciones* declaran que las operaciones se realizan de forma genérica en varios tipos de colecciones. La figura 16.1 muestra algunas de las interfaces del marco de colecciones. Varias implementaciones de estas interfaces se proporcionan dentro del marco. También puede proporcionar sus propias implementaciones.

Colecciones Basadas en **Objeto**

Las clases e interfaces del marco de colecciones son miembros del paquete **java.util**. En las primeras versiones de Java, las clases del marcos de colecciones almacenaban y manipulaban solo referencias **Object**, lo que nos permitía almacenar cualquier objeto en una colección.

Los programas normalmente necesitan procesar tipos específicos de objetos. Como resultado, las referencias **Object** obtenidas de una colección deben *convertirse descendentemente* (*downcasting*) a un tipo apropiado para permitir que el programa procese los objetos correctamente. Como discutimos antes, por lo general debe evitarse la conversión descendente.

Colecciones Genéricas

Para eliminar este problema, el marco de colecciones se mejoró con las capacidades de *genéricos* que introdujimos con **ArrayLists**. Los genéricos nos permiten especificar el tipo exacto que se almacenará en una colección y nos brindan los beneficios de la verificación de tipos en tiempo de compilación: el compilador emite mensajes de error si utiliza tipos inapropiados en sus colecciones.

Una vez que especificamos el tipo almacenado en una colección genérica, cualquier referencia que recupere de la colección tendrá ese tipo. Esto elimina la necesidad de conversiones de tipos explícitos que pueden generar **ClassCastExceptions** si el objeto al que se hace referencia no es del tipo apropiado. Además, las colecciones genéricas son compatibles con el código Java que se escribió antes de que se introdujeran los genéricos.

Buena Práctica de Programación

Evite reinventar la rueda: en lugar de crear sus propias estructuras de datos, use las interfaces y las colecciones del marco de las colecciones de Java, que se han probado y ajustado cuidadosamente para satisfacer la mayoría de los requisitos de las aplicaciones.

Eligiendo una Colección

La documentación de cada colección analiza sus requisitos de memoria y las características de rendimiento de sus métodos para operaciones como agregar y eliminar

elementos, buscar elementos, ordenar elementos y más. Antes de elegir una colección, revise la documentación en línea para la categoría de colección que está considerando (**Set**, **List**, **Map**, **Queue**, etc.), luego elija la implementación que mejor se adapte a las necesidades de su aplicación.

3.- *Clases envoltura de tipo*

Cada tipo primitivo tiene una *clase envoltura de tipo* correspondiente (en el paquete **java.lang**). Estas clases se llaman **Boolean**, **Byte**, **Character**, **Double**, **Float**, **Integer**, **Long** y **Short**. Esto nos permite manipular valores de tipo primitivo como objetos. Esto es importante porque las estructuras de datos que reutilizamos o desarrollamos de ahora en adelante manipulan y comparten objetos, no pueden manipular variables de tipos primitivos. Sin embargo, podemos manipular objetos de las clases envoltura de tipo, porque cada clase deriva finalmente de **Object**.

Cada una de las clases envoltura de tipo numérico (**Byte**, **Short**, **Integer**, **Long**, **Float** y **Double**) extiende la clase **Number**. Además, las clases envoltura de tipo son clases **final**, por lo que no pueden extenderse. Los tipos primitivos no tienen métodos, por lo que los métodos relacionados con un tipo primitivo se ubican en la clase envoltura de tipo correspondiente (por ejemplo, el método **parseInt**, que convierte una cadena en un valor **int**, se encuentra en la clase **Integer**).

4.- Auto-empaque y Auto-desempaque

Java proporciona conversiones de empaque y desempaque que convierten automáticamente entre valores de tipo primitivo y objetos envoltura de tipo. Una conversión de empaque convierte un valor de un tipo primitivo en un objeto de la clase envoltura de tipo correspondiente. Una conversión de desempaque convierte un objeto de una clase envoltura de tipo a un valor del tipo primitivo correspondiente. Estas conversiones se realizan de forma automática, lo que se denomina auto-empaque y auto-desempaque [ver AutoBoxing.java.]

Las conversiones de empaque también ocurren en condiciones que pueden evaluar valores primitivos **boolean** u objetos **Boolean**. Muchos de los ejemplos que se verán usan estas conversiones para almacenar valores primitivos y recuperarlos de las estructuras de datos.

5.- Interfaz **Collection** y Clase **Collections**

La interfaz **Collection** contiene *operaciones masivas* (es decir, operaciones realizadas en una colección completa) para operaciones tales como *agregar, borrar* y *comparar* objetos (o elementos) en una colección. Una **Collection** también puede convertirse en un arreglo. Además, la interfaz **Collection** proporciona un método que devuelve un objeto **Iterator**, que permite que un programa recorra la colección y elimine elementos durante la iteración. Otros métodos de la interfaz **Collection** permiten determinar el *tamaño* de una colección y si una colección está *vacía*.

Observación de Ingeniería de Software

**Collection** se usa comúnmente como un tipo de parámetro en los métodos para permitir el procesamiento polimórfico de todos los objetos que implementan la interfaz **Collection**.

Observación de Ingeniería de Software

La mayoría de las implementaciones de colecciones proporcionan un constructor que toma un argumento **Collection**, lo que permite construir una nueva colección que contenga los elementos de la colección especificada.

La clase **Collections** proporciona métodos estáticos que buscan, clasifican y realizan otras operaciones en las colecciones. En la sección 7 discutiremos más sobre los métodos de **Collections**.

## 6.- Las **List**

Una **List** (a veces llamada secuencia) es una **Collection** ordenada que puede contener elementos duplicados. Al igual que los índices de un arreglo, los índices de una **List** se basan en cero (es decir, el índice del primer elemento es cero). Además de los métodos heredados de **Collection**, **List** proporciona métodos para manipular elementos a través de sus índices, manipular un rango específico de elementos, buscar elementos y obtener un **ListIterator** para acceder a los elementos.

La interfaz **List** se implementa mediante varias clases, como **ArrayList**, **LinkedList** y **Vector**. El auto-empaque se produce cuando se agrega valores de tipo primitivo a los objetos de estas clases, porque almacenan solo referencias a objetos. Clases **ArrayList** y **Vector** son implementaciones de arreglos de tamaño variable de **List**.

Insertar un elemento entre elementos existentes de un **ArrayList** o **Vector** es una operación ineficiente: todos los elementos después del nuevo deben eliminarse, lo que podría ser una operación costosa en una colección con un gran número de elementos. Una **LinkedList** permite la inserción (o eliminación) *eficiente* de elementos en medio de una colección, pero es mucho menos eficiente que un **ArrayList** para saltar a un elemento específico de la colección.

**ArrayList** y **Vector** tienen comportamientos casi idénticos. Las operaciones en **Vector** están sincronizadas por defecto, mientras que las de **ArrayList** no lo están. Además, la clase **Vector** es de Java 1.0, antes de que el marco de las colecciones se agregara a Java. Como tal, **Vector** tiene algunos métodos que no forman parte de la interfaz **List** y no están implementados en la clase **ArrayList**. Por ejemplo, los métodos **Vector addElement** y **add** ambos agregan un elemento a un **Vector**, pero solo el método **add** se especifica en la interfaz **List** y es implementado por **ArrayList**.

Las colecciones no sincronizadas proporcionan un mejor rendimiento que las sincronizadas. Por esta razón, **ArrayList** se suele preferir a **Vector** en programas que no comparten una colección entre subprocesos.

Observación de Ingeniería de Software

Los **LinkedList** se puede utilizar para crear pilas, colas y colas doblemente terminadas (colas de doble final). El marco de colecciones proporciona implementaciones de algunas de estas estructuras de datos.

A continuación se demuestra las capacidades de **List** y **Collection**:

6.1.- Eliminación de elementos de un **ArrayList** con un **Iterator** [ver CollectionTest.java.]

Error Común de Programación

Si una colección es modificada por uno de sus métodos después de que se crea un iterador para esa colección, el iterador se vuelve inmediatamente inválido: cualquier operación realizada con el iterador falla de inmediato y lanza una **ConcurrentModificationException**. Por esta razón, se dice que los iteradores "fallan rápido". Los iteradores que fallan rápido ayudan a garantizar que una colección modificable no sea manipulada por dos o más subprocesos al mismo tiempo, lo que podría corromper la colección.

Observación de Ingeniería de Software

Nos referimos a los **ArrayList** en este ejemplo a través de las variables **List**. Esto hace que nuestro código sea más flexible y más fácil de modificar; si más adelante determinamos que **LinkedList** sería más apropiado, solo las líneas donde creamos los objetos **ArrayList** (líneas 14 y 21) deben modificarse. En general, cuando crea un objeto de colección, refiérase a ese objeto con una variable del tipo de interfaz de colección correspondiente.

6.2.- Uso de **ListIterator** y varios métodos específicos de **List** y **LinkedList** [ver ListTest.java y UsingToArray.java]

Error Común de Programación

Pasar un arreglo que contiene datos a **toArray** puede causar errores lógicos. Si el número de elementos en el arreglo es menor que el número de elementos en la lista en la que se llama **toArray**, se asigna un nuevo arreglo para almacenar los elementos de la lista, sin conservar los elementos del argumento del arreglo. Si el número de elementos en el arreglo es mayor que el número de elementos en la lista, los elementos del arreglo (que comienzan en el índice cero) se sobrescriben con los elementos de la lista. Los elementos del arreglo que no se sobrescriben conservan sus valores.