UNIVERSIDAD DE ALMERÍA

Grado en Ingeniería Informática
Metodología de la Programación

UAL. Área de Lenguajes y Sistemas Informáticos. Mercedes Martínez Durbán

Tema 5. Colecciones

- Introducción
- Colecciones
- La interface *List*
- Métodos estáticos para listas y colecciones
- La clase Vector y Stack
- Colas y Colas de Prioridad

Introducción

Una *estructura de datos* es una colección de datos organizada de alguna manera. La estructura no solo guarda datos sino que también debe proporcionar o soportar operaciones para acceder y manipular los datos.

En el paradigma de la POO, una estructura de datos, también conocida como ***contenedor*** o ***contenedor de objetos,*** es un objeto que almacena otros objetos y estos se referencian como datos o elementos. Para definir una **estructura de datos** es esencial definir una **clase**. La clase para una estructura de datos utilizará propiedades para almacenar datos y proporciona métodos para soportar operaciones tales como búsqueda, inserción y eliminación. Crear una estructura de datos es instanciar un objeto de la clase. Podremos a continuación usar los métodos de instancia para manipular la estructura.

Java proporciona distintas estructuras de datos que permiten organizar y manipular los datos eficientemente. Son comúnmente conocidas como ***Java Collections Framework***. En este tema introduciremos el estudio de listas, vectores, pilas, colas y colas de prioridad. En otros cursos se analizarán otras que completen el estudio de las estructuras de datos.

Collection

La interface **Collection** define operaciones comunes para listas, vectores, pilas, colas, colas de prioridad y conjuntos.

Java Collection Framework soporta dos tipos de contenedores:

- ✓ Uno para almacenar una colección de elementos, se conoce como *collection.*
- ✓ El otro, para almacenar pares de clave/valor, se conoce como *map.*

Los map son estructuras de datos más eficientes para búsquedas rápidas de un elemento usando una clave.

Las colecciones las componen:

Set almacena un grupo de elementos no duplicados.

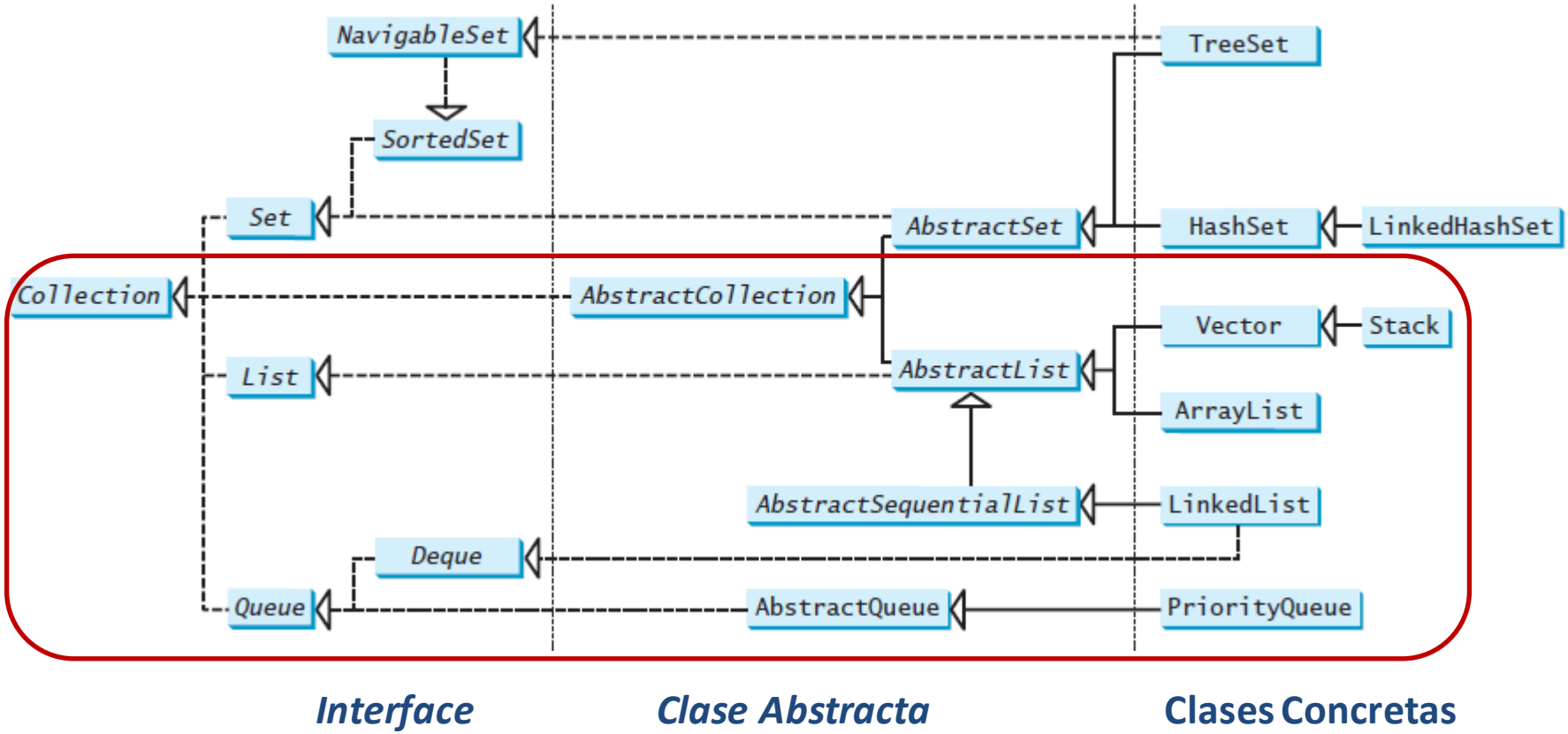
List almacena una colección de elementos en orden secuencial.

Stack almacena objetos que se procesan último en entrar, primero en salir (lifo).

Queue almacena objetos que se procesan primero en entrar, primero en salir (fifo).

PriorityQueue almacena objetos que se procesan en orden de su prioridad.

Las características comunes de las colecciones se definen en la interface y una implementación concreta se proporciona en las clases tal y como se muestra.



Todas las interfaces y clases definidas en Java Collection Framework se encuentran agrupadas en el paquete **java.util**.

El diseño de Java Collection Framework es un excelente ejemplo de uso de interfaces, clases abstractas y clases concretas. Las interfaces definen la estructura o esqueleto. Las clases abstractas proporcionan una implementación parcial. Las clases concretas implementan las interfaces con una estructura de datos concretas. Proporcionar una clase abstracta que implemente parcialmente una interface facilita la tarea de escribir código para el usuario. Este simplemente definirá una clase concreta que heredará de la clase abstracta y que implementará todos los métodos de la interface. La clase abstracta **AbstractCollection** se proporciona por conveniencia. Por esta razón se la conoce como *clase abstracta por conveniencia.*

La interface **Collection** es la interface raíz para manipular una colección de objetos. Sus métodos públicos se muestran en las siguientes diapositivas. La clase abstracta **AbstractCollection** proporciona una implementación parcial de **Collection**. Implementa todos los métodos de **Collection** excepto **add**, **size** e **iterator**. Estos se implementan en la subclase concreta.

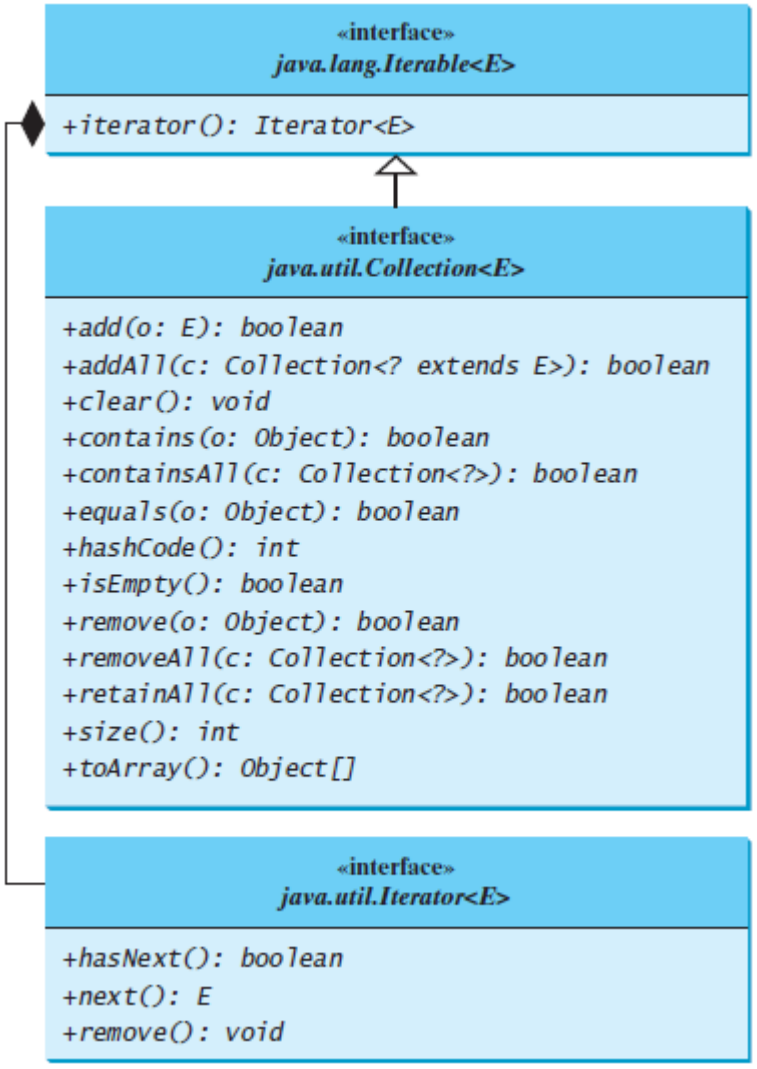
La interface **Collection** proporciona las *operaciones básicas* para añadir y eliminar elementos en una colección. El método **add** añade un elemento a la colección. El método **addAll** añade todos los elementos de una colección específica en esta colección. El método **remove** elimina un elemento de la colección. El método **removeAll** elimina los elementos de la colección que se presentan en la colección específica . El método **retainAll** mantiene los elementos de la colección que se presentan en la colección específica (elimina el resto). Todos estos métodos devuelven un booleano, verdadero si la colección se ha cambiado como resultado de la ejecución del método y falso en caso contrario. El método **clear** elimina todos los elementos de la colección.

La interface **Collection** proporciona varias *operaciones de consulta (query).* El método **size** devuelve el número de elementos de la colección. El método **contains** verifica cuando la colección contiene un elemento específico. El método **containsAll** comprueba cuando la colección contiene todos los elementos de la colección específica. El método **isEmpty** devuelve verdadero si la colección está vacía.

La interface **Collection** proporciona el método **toArray** que devuelve un array que representa a la colección.

Algunos métodos de la interface **Collection** no pueden ser implementados en la subclase concreta. En este caso, el método deberá lanzar la excepción **java.lang. UnsupportedException**, una subclase de **RuntimeException.**

La interface *Collection*



Devuelve un iterador para los elementos de esta colección.

Añade un nuevo elemento a la colección.
Añade todos los elementos de la colección c a esta colección.
Elimina todos los elementos de esta colección.
Devuelve verdadero si esta colección contiene el elemento o.
Devuelve verdadero si esta colección contiene todos los elementos de c.
Devuelve verdadero si esta colección es igual a la otra colección o.
Devuelve el hash code para esta colección.
Devuelve verdadero si esta colección no contiene elementos.
Elimina el elemento o de esta colección.
Elimina todos los elementos de c en esta colección.
Mantiene todos los elementos de c en esta colección.
Devuelve el número de elementos de esta colección.
Devuelve un array de Object con los elementos de la colección.

Devuelve verdadero si el iterador tiene más elemento que recorrer.
Devuelve el siguiente elemento de ese iterador.
Elimina el último elemento obtenido usando el método next.

Salida

```
Una lista de ciudades de la colección 1:
[Almeria, Granada, Cadiz, Jaen]

¿Está Almería en la colección 1? true

Ahora tiene 3 ciudades la colección 1

Una lista de ciudades de la colección 2:
[Madrid, Barcelona, San Sebastian, Granada]

Ciudades en la colección 1 o en la colección 2:
[Granada, Cadiz, Jaen, Madrid, Barcelona, San Sebastian, Granada]

Ciudades en la colección 1 y en la colección 2: [Granada]

Ciudades en la colección 1, pero no en la 2: [Cadiz, Jaen]
```

```java
package org.mp.tema05;
import java.util.*;

public class TestColeccion {
  public static void main(String[] args) {
    ArrayList<String> coleccion1 = new ArrayList<String>();
    coleccion1.add("Almeria");
    coleccion1.add("Granada");
    coleccion1.add("Cadiz");
    coleccion1.add("Jaen");

    System.out.println("Una lista de ciudades de la colección 1:");
    System.out.println(coleccion1);

    System.out.println("\n¿Está Almería en la colección 1? "
      + coleccion1.contains("Almeria"));

    coleccion1.remove("Almeria");
    System.out.println("\n"+
       "Ahora tiene "+ coleccion1.size() + " ciudades la colección 1");

    Collection<String> coleccion2 = new ArrayList<String>();
    coleccion2.add("Madrid");
    coleccion2.add("Barcelona");
    coleccion2.add("San Sebastian");
    coleccion2.add("Granada");

    System.out.println("\nUna lista de ciudades de la colección 2:");
    System.out.println(coleccion2);

    ArrayList<String> c1 = (ArrayList<String>)(coleccion1.clone());
    c1.addAll(coleccion2);
    System.out.println("\nCiudades en la colección 1 o en la colección 2: ");
    System.out.println(c1);

    c1 = (ArrayList<String>)(coleccion1.clone());
    c1.retainAll(coleccion2);
    System.out.print("\nCiudades en la colección 1 y en la colección 2: ");
    System.out.println(c1);

    c1 = (ArrayList<String>)(coleccion1.clone());
    c1.removeAll(coleccion2);
    System.out.print("\nCiudades en la colección 1, pero no en la 2: ");
    System.out.println(c1);
  }
}
```

Cada colección es **Iterable**. Podemos obtener un objeto **Iterator** para recorrer todos los elementos en la colección.

La interface **Collection** hereda de la interface **Iterable.** La interface **Iterable** define el método **iterator**, el cual devuelve un iterador. La interface **Iterator** proporciona una misma manera para recorrer elementos en los distintos tipos de colecciones. Para ello, se utilizarán los métodos **next( )**, **hasNext( )** y **remove( ).**

```
1  package org.mp.tema05;
2  import java.util.*;
3
4  public class TestIterator {
5    public static void main(String[] args) {
6      Collection<String> collection = new ArrayList<>();
7      collection.add("Almeria");
8      collection.add("Granada");
9      collection.add("Cadiz");
10     collection.add("Jaen");
11
12     Iterator<String> iterator = collection.iterator();
13     while (iterator.hasNext()) {
14       System.out.print(iterator.next().toUpperCase() + " ");
15     }
16     System.out.println();
17   }
18 }
```

Salida

```
ALMERIA GRANADA CADIZ JAEN
```

La interface *List*

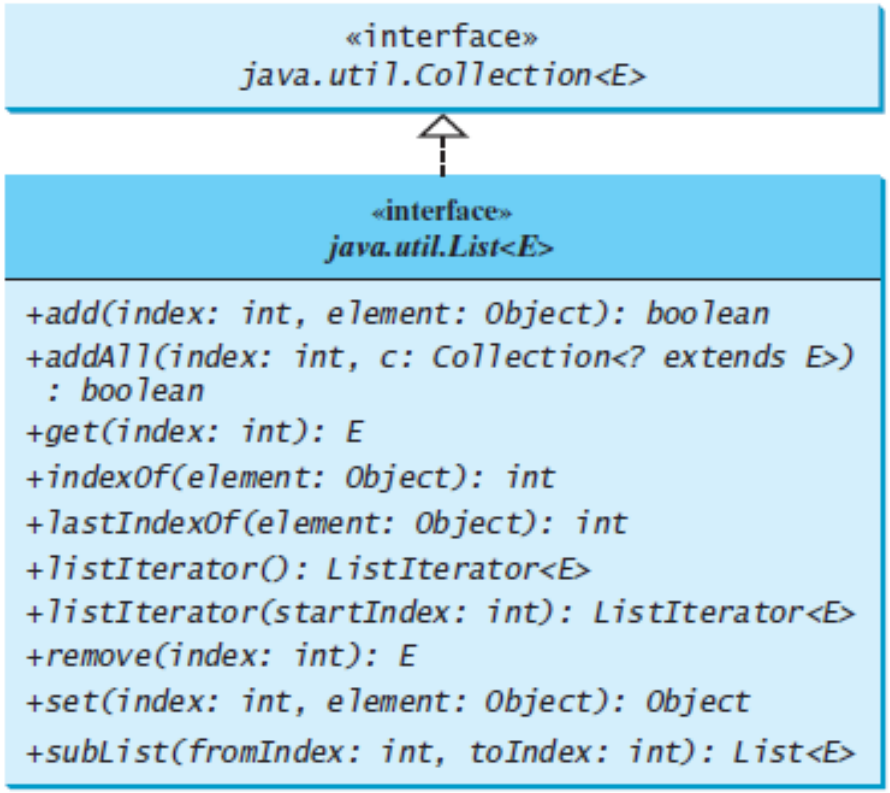
La interface **List** hereda de la interface **Collection** y define una colección para guardar elementos en ***orden secuencial***. Para crear una lista, usaremos una de las dos implementaciones concretas de clases: **ArrayList** o **LinkedList**.

Las clases **ArrayList** y **LinkedList** se definen de acuerdo a la interface **List** que implementan. La interface **List** añade *operaciones relacionadas con la posición* así como un *nuevo iterador* que permite al usuario recorrer la lista bidireccionalmente.

El método **add(index, element)** se usa para insertar un elemento en la posición especificada por index y **add(index, collection)** inserta una colección de elementos en la posición especifica por index. El método **remove(index)** se usa para eliminar un elemento en la posición especificada por index. Un nuevo elemento puede ponerse en una posición especificada por index con el método **set(index, element)**.

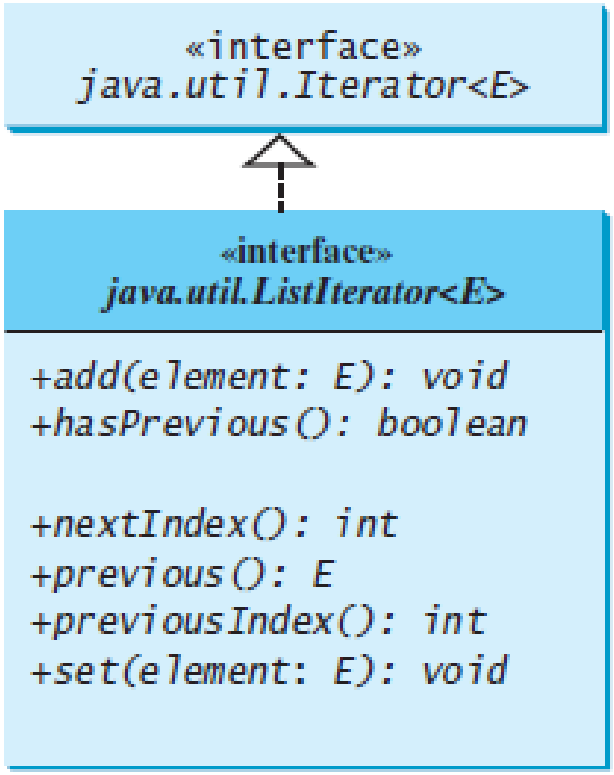
El método **indexOf(element)** se usa para obtener la posición de la primera ocurrencia del elemento especificado (element). El método **lastIndexOf(element)** se usa para obtener la posición de la última aparición. Se puede obtener una sublista utilizando el método **subList(fromIndex, toIndex)**.

La interface *List*



+add(index: int, element: Object): boolean	Añade un nuevo elemento en la posición especificada.
+addAll(index: int, c: Collection<? extends E>) : boolean	Añade todos los elementos de c a la lista en la posición especificada por index.
+get(index: int): E	Devuelve el elemento de la lista que está en la posición index.
+indexOf(element: Object): int	Devuelve la posición de la primera ocurrencia de element.
+lastIndexOf(element: Object): int	Devuelve la posición de la última ocurrencia de element.
+listIterator(): ListIterator<E>	Devuelve un iterador para los elementos de la lista.
+listIterator(startIndex: int): ListIterator<E>	Devuelve un iterador para los elementos de la lista a partir de startIndex
+remove(index: int): E	Elimina el elemento en la posición especificada por index.
+set(index: int, element: Object): Object	Pone el elemento en la posición especificada por index.
+subList(fromIndex: int, toIndex: int): List<E>	Devuelve una sublista desde fromIndex hasta toIndex.

La interface *ListIterator*



El método **add(element)** inserta un elemento especificado. El elemento se inserta inmediatamente antes del elemento devuelto por el método **next()** definido en la interface **Iterator**, si los hay y después del elemento devuelto por el método **previous()**, si los hay. Si la lista no tienen ningún elemento, el nuevo elemento se convierte en el único de la lista.

Añade el objeto especificado en la lista.
Devuelve verdadero si el iterador de la lista tiene más elementos para recorrer hacía atrás.
Devuelve la posición del siguiente elemento en el iterador.
Devuelve el elemento anterior del iterador de la lista.
Devuelve la posición del elemento anterior en el iterador.
Sustituye el último elemento devuelto por el método **previous** o **next** por el elemento especificado como element.

El método **listIterator** o **listIterator(starIndex)** devuelven una instancia de **ListIterator**, esta hereda de la interface **Iterator**. Permitirá recorrer una lista bidireccionalmente.

Las clases ArrayList y LinkedList

La clase **AbstractList** proporciona una implementación parcial para la interface **List.** La clase **AbstractSequentialList** hereda de **AbstractList** para proporcionar soporte a las listas enlazadas.

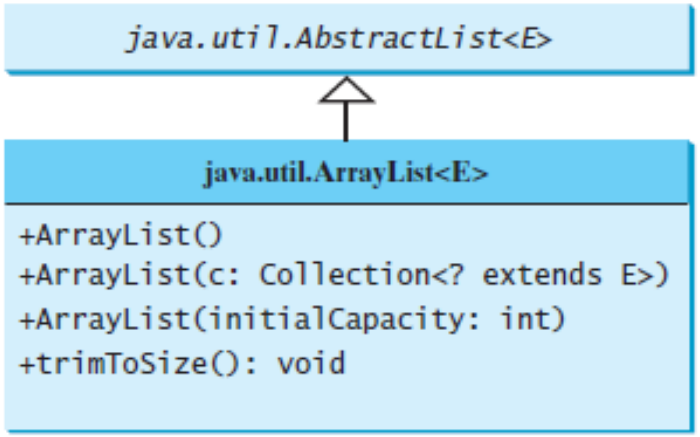
Las clase **ArrayList** y **LinkedList** son dos implementaciones concretas de la interface **List.** **ArrayList** almacena elementos en un array. El array se crea dinámicamente. Si la capacidad se excede, se crea un array nuevo mayor y todos los elementos del array se copian en el nuevo array. **LinkedList** almacena elementos en una lista enlazada.

¿Que clase utilizar? Dependerá de las necesidades del problema.

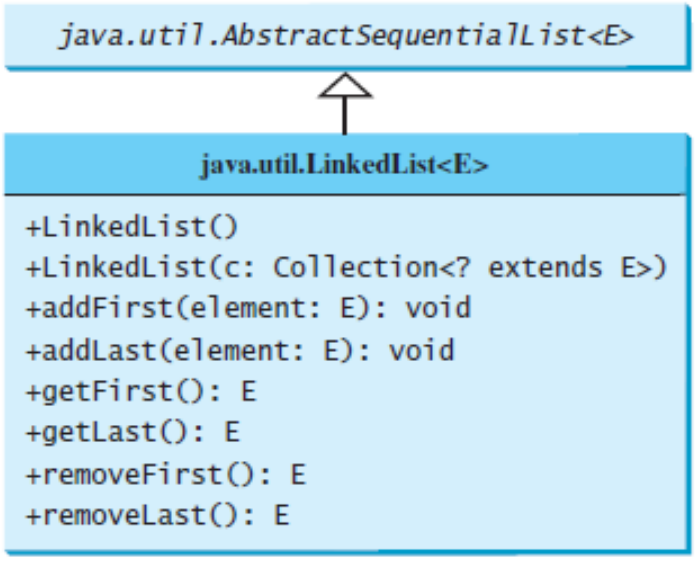
Si se necesitan accesos aleatorios a partir de un índice sin inserciones o eliminaciones al principio de la lista, lo más eficiente es **ArrayList.**

Si se necesitan inserciones y eliminaciones de elementos al principio de la lista, elegiremos **LinkedList**.

La clase ArrayList y LinkedList



Crea una lista vacía con una capacidad inicial por defecto.
Crea un array list a partir de una colección
Crea una lista vacía con una capacidad inicial especificada.
Ajusta la capacidad de la instancia de ArrayList al tamaño actual de la lista.



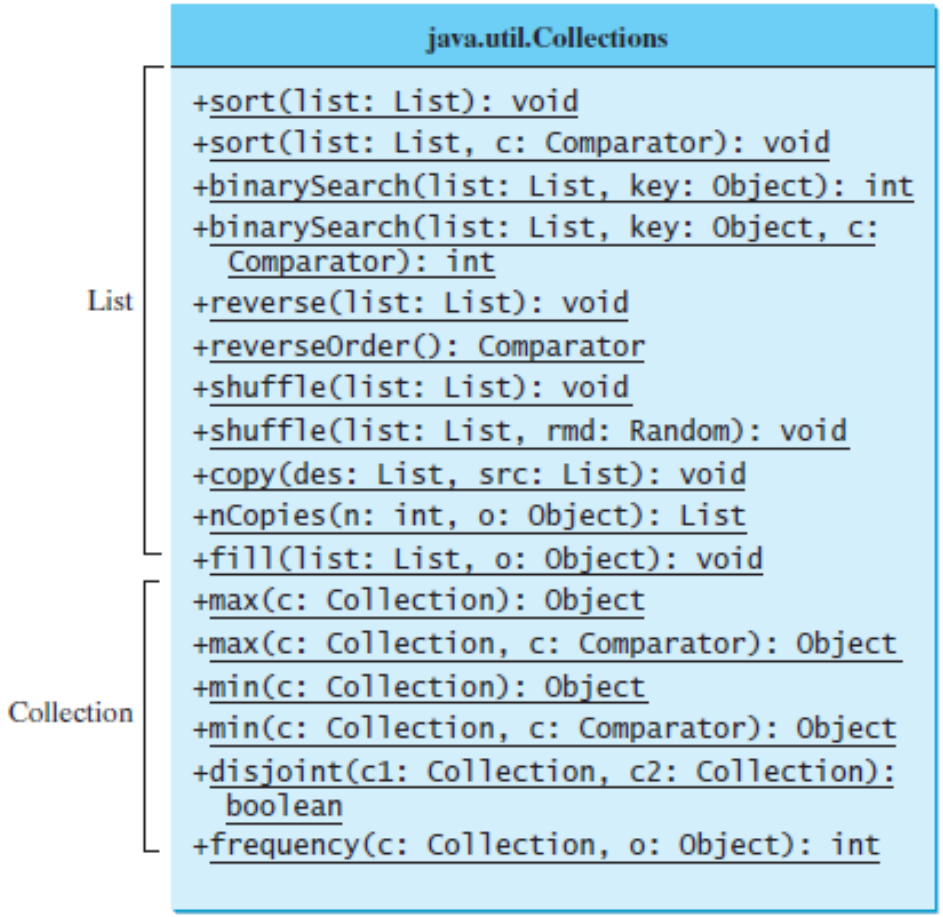
Crea una lista enlazada vacía.
Crea una lista enlazada a partir de una colección existentes.
Añade el elemento a la cabeza de la lista.
Añade el elemento a la cola de la lista.
Devuelve el último elemento de la lista.
Devuelve y elimina el primer elemento de la lista.
Devuelve y elimina el último elemento de la lista.

```java
package org.mp.tema05;
import java.util.*;

public class TestArrayYLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(1); // autoboxing
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);

    System.out.println("Lista de enteros con ArrayList:");
    System.out.println(arrayList);

    LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
    linkedList.add(1, "rojo");
    linkedList.removeLast();
    linkedList.addFirst("verde");

    System.out.println("Muestra la lista hacia delante:");
    ListIterator<Object> listIterator = linkedList.listIterator();
    while (listIterator.hasNext()) {
        System.out.print(listIterator.next() + " ");
    }
    System.out.println();

    System.out.println("Muestra la lista hacia atrás:");
    listIterator = linkedList.listIterator(linkedList.size());
    while (listIterator.hasPrevious()) {
        System.out.print(listIterator.previous() + " ");
    }
    }
}
```

Salida

```
Lista de enteros con ArrayList:
[10, 1, 2, 30, 3, 1, 4]
Muestra la lista hacia delante:
verde 10 rojo 1 2 30 3 1
Muestra la lista hacia atrás:
1 3 30 2 1 rojo 10 verde
```

Métodos estáticos de la clase Collections



+sort(list: List): void	Ordena la lista especificada.
+sort(list: List, c: Comparator): void	Ordena con el comparador la lista especificada.
+binarySearch(list: List, key: Object): int	Busca la clave en la lista ordenada usando la búsqueda binaria.
+binarySearch(list: List, key: Object, c: Comparator): int	Busca la clave en la lista ordenada usando la búsqueda binaria y un comparador.
+reverse(list: List): void	Invierte la lista especificada.
+reverseOrder(): Comparator	Devuelve un comparador con el orden inverso.
+shuffle(list: List): void	Baraja la lista especificada aleatoriamente.
+shuffle(list: List, rmd: Random): void	Baraja la lista especificada con un objeto aleatorio.
+copy(des: List, src: List): void	Copia la lista fuente en la lista destino.
+nCopies(n: int, o: Object): List	Devuelve una lista que contiene n copias de un objeto.
+fill(list: List, o: Object): void	Rellena la lista con el objeto o.
+max(c: Collection): Object	Devuelve el máximo objeto de la colección.
+max(c: Collection, c: Comparator): Object	Devuelve el máximo objeto usando un comparador.
+min(c: Collection): Object	Devuelve el mínimo objeto de la colección.
+min(c: Collection, c: Comparator): Object	Devuelve el mínimo objeto usando un comparador.
+disjoint(c1: Collection, c2: Collection): boolean	Devuelve verdadero si c1 y c2 no tienen elementos en común.
+frequency(c: Collection, o: Object): int	Devuelve el número de ocurrencias en la colección de un elemento especificado.

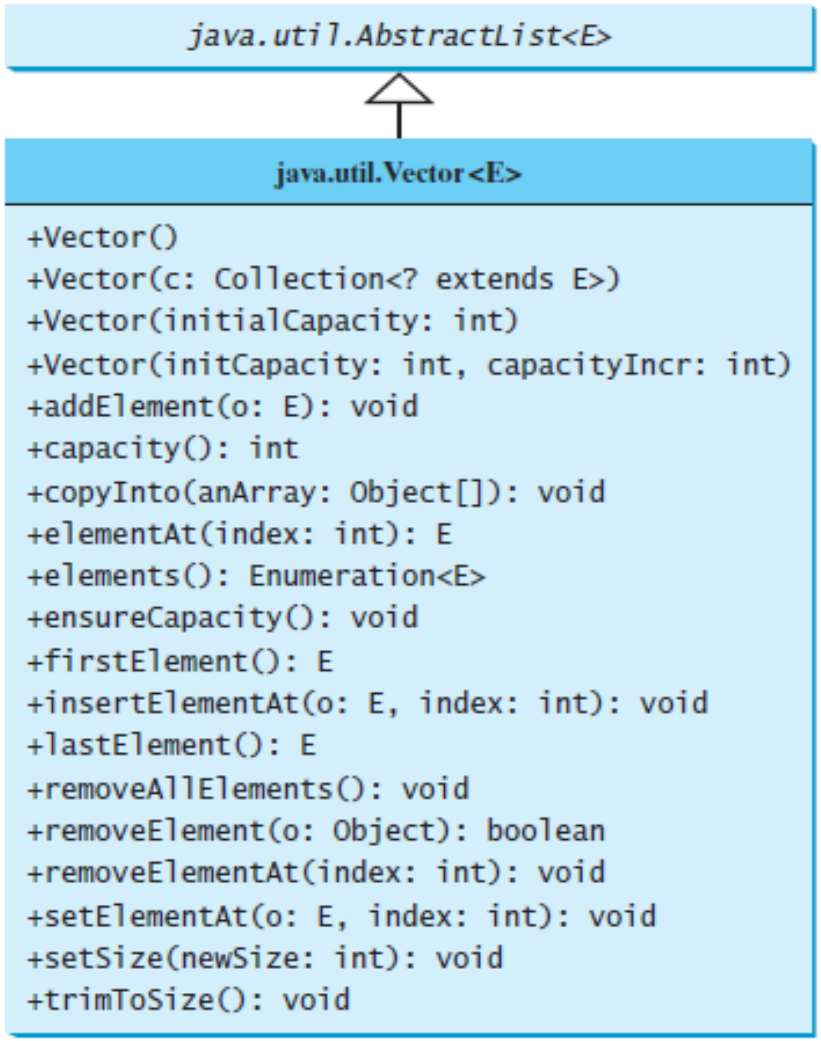
Las clase Vector y Stack

Vector es una subclase de **AbstractList** y **Stack** es una subclase de **Vector.**

Java Collections Framework fué introducida en Java 2. Algunas estructuras de datos estaban soportadas con anterioridad, entre ellas **Vector** y **Stack.** Estas clases se rediseñaron para encajar en las colecciones pero todos sus métodos antiguos se conservan por compatibilidad.

Vector es muy similar a **ArrayList.**

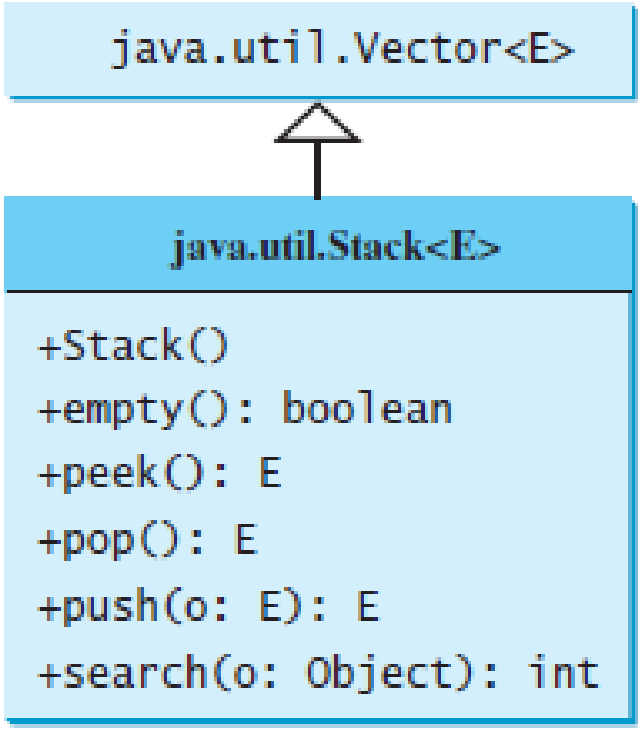
La clase Vector



A partir de Java 2 , la clase Vector hereda de **AbstractLis**t y también conserva todos los métodos antiguos.

Crea un vector vacío con una capacidad inicial de 10.
Crea un vector a partir de una colección existente.
Crea un vector con una capacidad inicial especificada.
Crea un vector con una capacidad inicial especificada e incrementada.
Añade un elemento al final del vector.
Devuelve la capacidad actual del vector.
Copia los elementos del array en el vector.
Devuelve el objeto que está en la posición especificada.
Devuelve una enumeración de ese vector.
Incrementa la capacidad del vector.
Devuelve el primer elemento del vector.
Inserta el elemento o en el vector en la posición especificada.
Devuelve el último elemento del vector.
Elimina todos los elementos del vector.
Elimina el elemento del vector en la posición especificada.
Pone un nuevo elemento en la posición especificada.
Pone un nuevo tamaño al vector.
Ajusta la capacida del vector al tamaño.

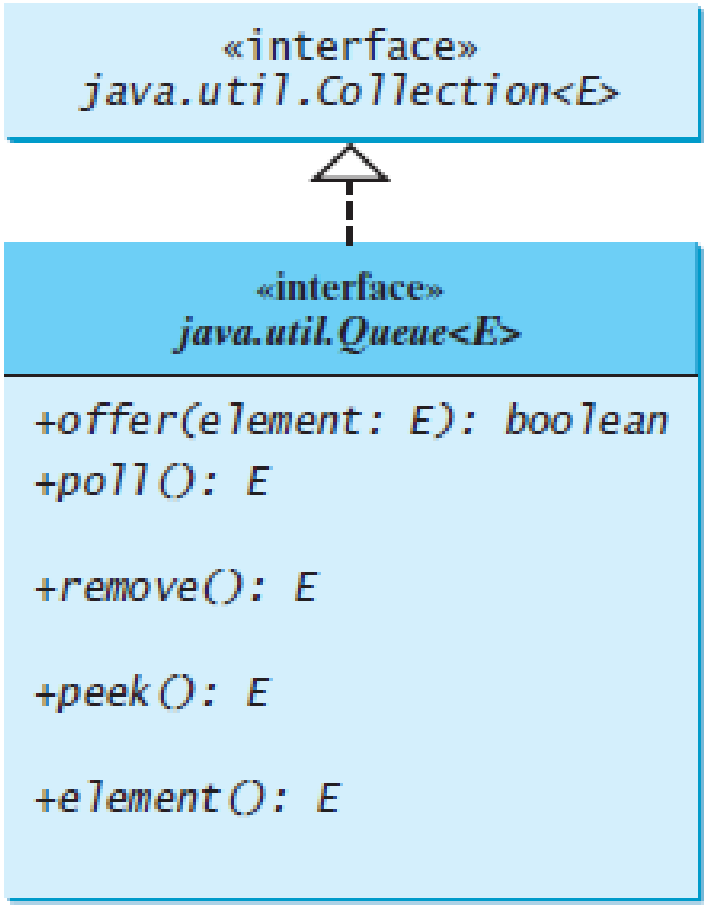
La clase Stack



En Java Collection Framework, **Stack** se implementa como una extensión de **Vector**

Crea una pila vacía.
Devuelve verdadero si la pila está vacía.
Devuelve la cima de la pila.
Devuelve y elimina la cima de la pila.
Añade un elemento a la cima de la pila.
Devuelve la posición del elemento especificado en la pila.

La Interface *Queue*



Una cola es una estructura de datos en la que el primer elemento en entrar es el primero en salir. Los elementos se añaden por el final y se eliminan por el principio. En una cola de prioridad, a los elementos se le asignan prioridades. El elemento con más prioridad es el que se elimina el primero.

Inserta un elemento en la cola.

Devuelve y elimina la cabeza de la cola o devuelve null si la cola está vacía.

Devuelve y elimina la cabeza de la cola y lanza una excepción si la cola está vacía.

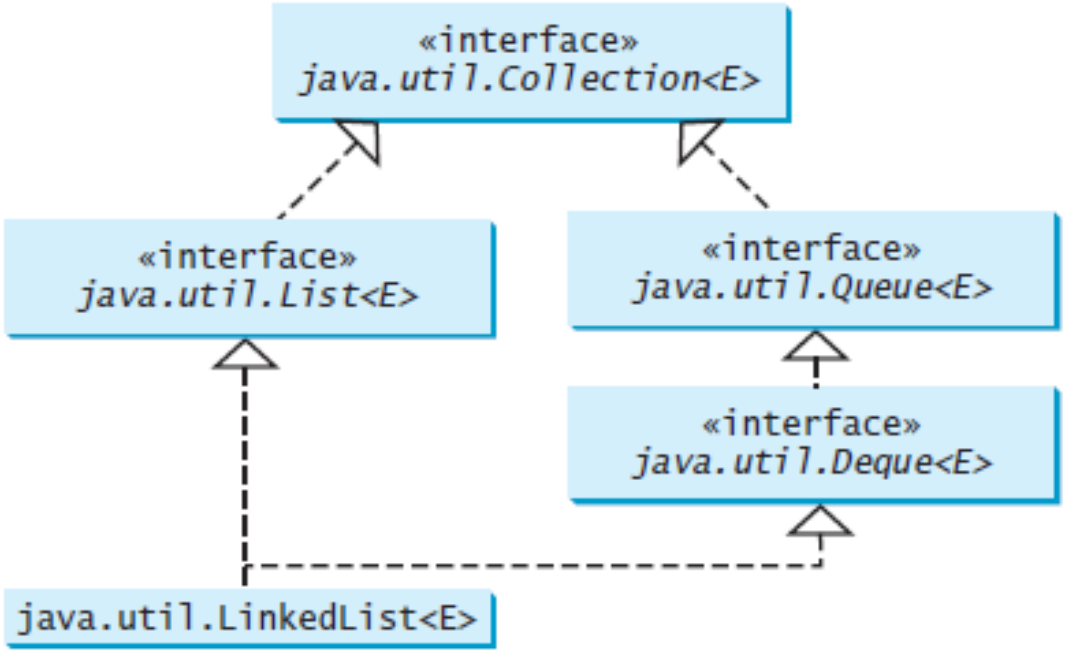
Devuelve y no elimina, la cabeza de la cola, devuelve null si la cola está vacía.

Devuelve y no elimina, la cabeza de la cola, lanza una excepción si la cola está vacía.

La Interface *Deque* y la clase LinkedList

La clase **LinkedList** implementa la interface **Deque** la cual hereda de la interface **Queue**. Por lo tanto, usaremos **LinkedList** para crear una cola. **LinkedList** es ideal para operaciones de una cola porque es eficiente para insertar y eliminar elementos por ambos extremos de la lista.

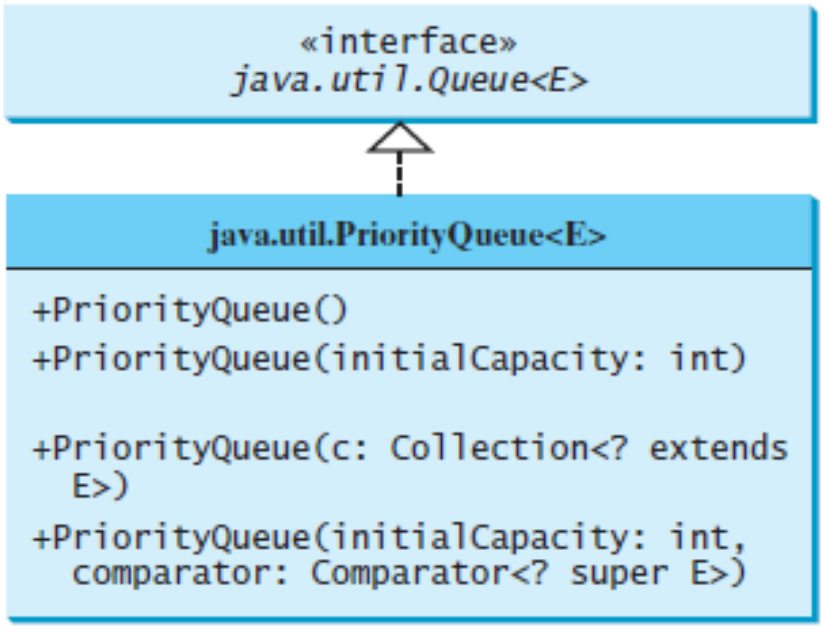
Deque soporta la inserción y eliminación de elementos por ambos extremos. *Deque* es el nombre abreviado de "double ended queue" (cola de doble extremo). Normalmente "deck". Los métodos **addFirst(e), removeFirst(), addLast(e), removeLast(), getFirst(), getLast()** se definen en **Deque**



La clase PriorityQueue

Esta clase implementa una cola de prioridad. Por defecto, la cola de prioridad ordena sus elementos de acuerdo a su orden natural usando **Comparable.** El elemento con el menor valor se le asigna la prioridad más alta y se elimina el primero.

Podemos especificar un orden usando **Comparator** en el constructor de la cola de prioridad.



Crea una cola de prioridad por defecto con una capacidad 11.
Crea una cola por defecto con una capacidad inicial especificada.

Crea una cola de prioridad con una colección especificada.

Crea una cola de prioridad con una capacidad especificada y un comparador.

```java
package org.mp.tema05;
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> cola1 = new PriorityQueue<String>();
    cola1.offer("Oviedo");
    cola1.offer("Huesca");
    cola1.offer("Granada");
    cola1.offer("Teruel");

    System.out.println("Cola de prioridad usando Comparable:");
    while (cola1.size() > 0) {
      System.out.print(cola1.remove() + " ");
    }
    System.out.println();

    PriorityQueue<String> cola2 = new PriorityQueue<String>(
      4, Collections.reverseOrder());
    cola2.offer("Oviedo");
    cola2.offer("Huesca");
    cola2.offer("Granda");
    cola2.offer("Terual");

    System.out.println("\nCola de prioridad usando Comparator:");
    while (cola2.size() > 0) {
      System.out.print(cola2.remove() + " ");
    }
  }
}
```

Salida

```
Cola de prioridad usando Comparable:
Granada Huesca Oviedo Teruel

Cola de prioridad usando Comparator:
Terual Oviedo Huesca Granda
```

Resumen

- Java Collections Framwork soporta *set, list, queue* y *map*.
- Una lista almacena una colección de elementos.
- Todas las clases concretas excepto **PriorityQueue** implementan las interfaces **Cloneable .** Por tanto, sus instancias pueden clonarse.
- Para almacenar elementos duplicados en una colección usaremos una lista. El usuario puede acceder a los elementos a partir de un índice.
- Dos tipos de listas están soportadas: **ArrayList** y **LinkedList.** Ambas implementan la interface **List.**
- La clase **Vector** hereda de **AbstractList**. La clase **Stack** hereda de **Vector.**
- La interface **Queue** representa una cola. La clase **PriorityQueue** implementa a **Queue** y representa una cola de prioridad.

¡MUCHAS GRACIAS!

UNIVERSIDAD DE ALMERÍA