

Colecciones e Iteradores

Contenido

- Clases genéricas
- Colecciones
 - Las interfaces básicas y sus implementaciones.
 - Conjuntos, listas y aplicaciones.
- Clases ordenables
- Colecciones y aplicaciones ordenadas.
- Decoradores
- Algoritmos sobre arrays. La clase Arrays.

Clases genéricas (revisión)

- Las clases genéricas permiten, en una única definición, expresar comportamientos comunes para objetos pertenecientes a distintas clases. El ejemplo más habitual de clase genérica son las clases contenedoras: listas, pilas, árboles, etc.
- Desde la versión JDK1.5.0, Java dispone de mecanismos para definir e instanciar clases genéricas mediante el uso de parámetros.
 - Una clase o interfaz puede incorporar parámetros en su definición, que siempre representan clases o interfaces.
 - A la hora de instanciar la clase genérica, se especifica el valor concreto de los parámetros. Éste debe ser una clase o interfaz, nunca un tipo básico.
 - Una clase genérica puede instanciarse tantas veces como sea necesario, y los valores reales utilizados pueden variar.
 - En la definición pueden especificarse restricciones sobre los parámetros formales, que deberán ser satisfechos por los parámetros reales en la instanciación.

Un ejemplo simple

- Supongamos que queremos crear una clase que almacene dos elementos de otra clase.
 - No indicamos de qué clase son los elementos a almacenar, para que cualquiera se pueda utilizar.
 - Supongamos que son de la clase T donde T representa a cualquier clase.

```
public class Par {    <T>
    private T uno, otro;
    public Par(T u, T o) {
        uno= u;
        otro= o;
    }
    public T uno() {
        return uno;
    }
    public T otro() {
        return otro;
    }
}
```

¿Cómo sabe Java que T no es una clase concreta sino que representa a cualquiera?

Añadiendo <T> a la cabecera

¿Cómo usar los objetos de esa clase?

```
public class Programa {
    public static void main(String [] args) {
        Par<String> conC = new Par<String>("hola", "adios");
        Medico med = new Medico(...);
        Paciente pac = new Paciente(...);
        Par<Persona> consulta = new Par<med,pac>;
        Par<Integer> conI = new Par<Integer>(4, 9);
        Par<Integer> conN = new Par<>(4, 9);
        ...
    }
}
```

Desde la
versión
Java1.7

Otro ejemplo

```
public class Carta<P> {  
  
    private P palo;  
    private int valor;  
  
    public Carta(int v , P p) {  
        valor= v;  
        palo = p;  
    }  
  
    public P palo() {  
        return palo;  
    }  
  
    public int valor() {  
        return valor;  
    }  
  
    public String toString() {  
        return "[" + valor + " de " + palo + "];"  
    }  
}
```

```
public enum PaloFrances  
    {PICAS, TREBOLES, DIAMANTES, CORAZONES};  
  
public enum PaloEspañol  
    {OROS, COPAS, ESPADAS, BASTOS};
```

```
public class PruebaCartas {  
    public static void main(String[] args){  
        Carta<PaloFrances> naipe =  
            new Carta<>(12,PaloFrances.DIAMANTES);  
        Carta<PaloEspañol> carta=  
            new Carta<>(13,PaloEspañol.OROS);  
        System.out.println("El rey: "+ naipe + carta);  
    }  
}
```

Clases genéricas. Herencia

- Una clase puede definirse genérica relacionando su parámetro con el que tuviera su superclase o alguna interfaz que implemente.
 - Por ejemplo, la clase ParPeso tiene el mismo parámetro que la clase Par de la que hereda.

```
public class ParPeso<T> extends Par<T> {  
    int pesoPrimero;  
    int pesoSegundo;  
  
    public ParPeso(T p, int pp, T s, int ps) {  
        super(p, s);  
        pesoPrimero = pp;  
        pesoSegundo = ps;  
    }  
    ...  
}
```

```
ParPeso<String> parp =  
    new ParPeso<>("hola", 112, "adios", 65);
```

Clases genéricas. Restricciones

- Podemos imponer restricciones a los valores que toma un parámetro:
 - Que sea de una clase o subclase de una clase dada.
 - Que implementen una o varias interfaces.

```
public class ParNumerico<T extends Number> extends Par<T> {  
}
```

```
ParNumerico<Integer> p = new ParNumerico<>(10, 15);
```

```
ParNumerico<String> q = new ParNumerico<String>("Hola", "adios");
```

- La forma general de definir una restricción sobre el parámetro de una clase genérica es:

`<T extends A & I1 & I2 & ... & In>`

Genericidad y herencia

- Una clase puede definirse genérica relacionando su parámetro con el que tuviera su superclase o alguna interfaz que implemente.
 - Por ejemplo, la clase ParPeso tiene el mismo parámetro que la clase Par de la que hereda.

```
public class ParPeso<T> extends Par<T> {  
    int pesoPrimero;  
    int pesoSegundo;  
  
    public ParPeso(T p, int pp, T s, int ps) {  
        super(p, s);  
        pesoPrimero = pp;  
        pesoSegundo = ps;  
    }  
    ...  
}
```

```
ParPeso<String> parp = new ParPeso<>("hola", 112, "adios", 65);
```


Genericidad y herencia

*Si la clase D es subclase de B, entonces la clase F<D> **no** es subclase de F*

Ejemplo:

La clase String es subclase de Object pero la clase Par<String> **no** se puede considerar subclase de Par<Object>

Si fuera así, podrían producirse problemas:

```
Par<String> parS = new Par<>("hola", "adios");  
Par<Object> parO = parS; // Si se cumpliera la propiedad  
parO.primerO(new Object());  
String s = parS.primerO();
```



Clases con más de un parámetro

- Una clase genérica puede disponer de varios parámetros:

```
public class Pareja<A,B> {  
    private A primero;  
    private B segundo;  
  
    public Pareja(A a, B b) {  
        primero = a;  
        segundo = b;  
    }  
  
    public A primero() {  
        return primero;  
    }  
  
    public B segundo() {  
        return segundo;  
    }  
  
    public void primero(A a) {  
        primero = a;  
    }  
  
    public void segundo(B b) {  
        segundo = b;  
    }  
}
```

```
Pareja<String,Integer> aparicionesPalabra =  
    new Pareja<String,Integer>("hola", 10);  
Pareja<Paciente,Medico> atencion =  
    new Pareja<>(pac, pers);
```

**Desde la
versión Java1.7.**

Genericidad y herencia

- Las clases genéricas **no** cumplen la siguiente relación:

Si la clase D es subclase de B, entonces la clase F<D> es subclase de F

Ejemplo:

La clase String es subclase de Object pero la clase Par<String> **no** se puede considerar subclase de Par<Object>

- Si fuera así, podrían producirse problemas:

```
Par<String> parS = new Par<String>("hola", "adios");  
Par<Object> parO = parS; // Si se cumpliera la propiedad  
parO.primerO(new Object());  
String s = parS.primerO();
```



Colecciones

- El marco de colecciones del JDK presenta un conjunto de clases estándar útiles (en `java.util`) para el manejo de colecciones de datos.
- Proporciona:
 - Interfaces. Para manipularlas de forma independiente de la implementación.
 - Implementaciones. Implementan la funcionalidad de alguna manera.
 - Algoritmos. Para realizar determinadas operaciones sobre colecciones, como ordenaciones, búsquedas, etc.
- Beneficios de usar el marco de colecciones:
 - Reduce los esfuerzos de programación.
 - Incrementa velocidad y calidad.
 - Ayuda a la interoperabilidad y reemplazabilidad.
 - Reduce los esfuerzos de aprendizaje y diseño.

Interfaces básicas

Interfaz que define las operaciones que normalmente implementan las clases que representan colecciones de objetos.

Interfaz que define las operaciones que normalmente implementan las clases que representan aplicaciones de claves a valores.

Extiende Collection para conjuntos con elementos únicos.

«interface»
Collection<T>

«interface»
Map<K,V>

«interface»
Set<T>

«interface»
Queue<T>

«interface»
List<T>

«interface»
SortedMap<K,V>

«interface»
SortedSet<T>

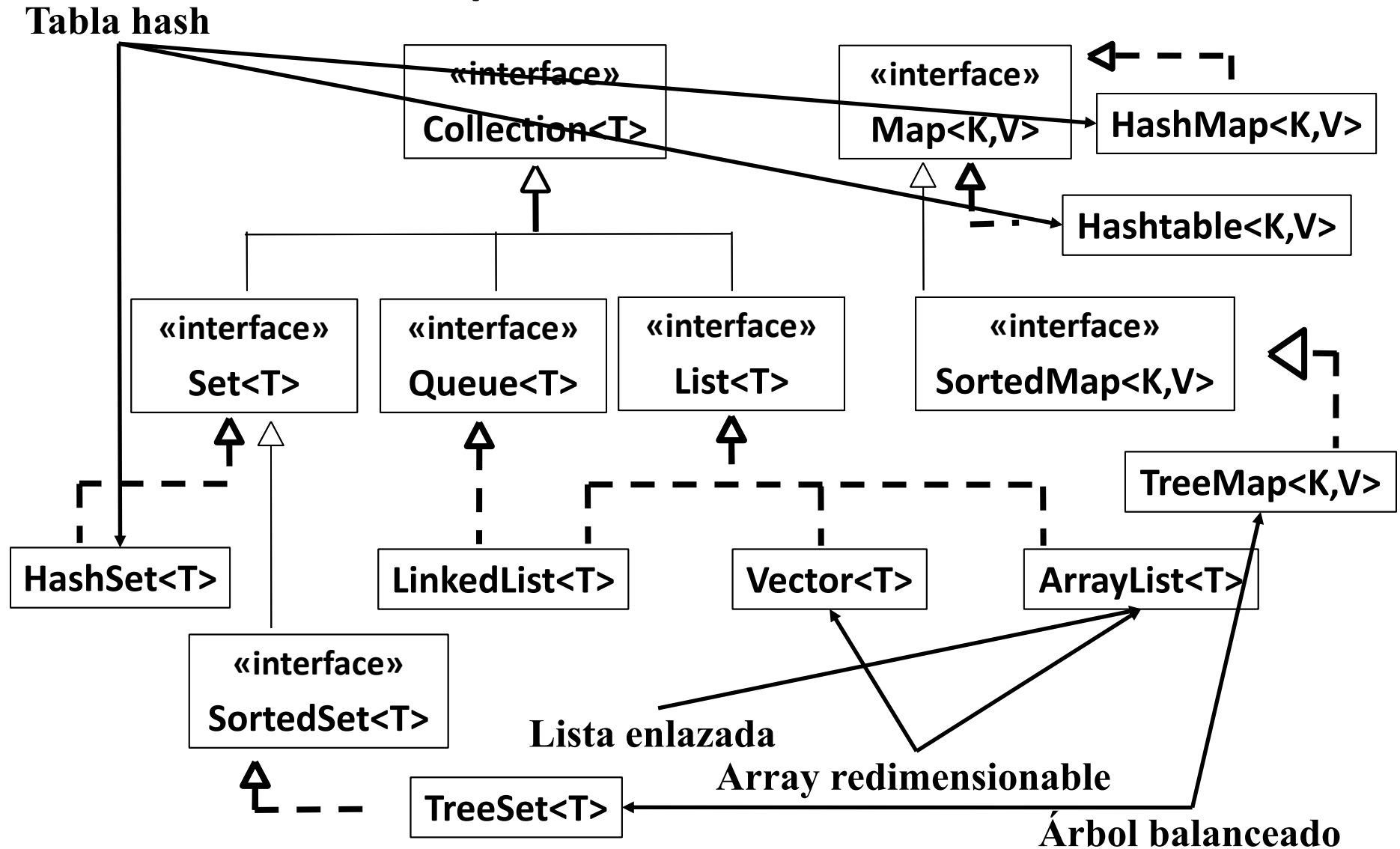
Colas de
elementos

Extiende Collection para secuencias de elementos, a los que se puede acceder atendiendo a su posición dentro de ésta.

Extiende Map para aplicaciones que mantienen sus relaciones ordenadas por sus claves.

Extiende Set para conjuntos que mantienen sus elementos ordenados.

Interfaces básicas y sus implementaciones



Implementaciones

- No hay implementación directa de la interfaz `Collection<T>` ésta se utiliza sólo para mejorar la interoperación de las distintas colecciones.
- Por convención, las clases que implementan colecciones proporcionan constructores para crear nuevas colecciones con los elementos de un objeto (que se le pasa como argumento) de clase que implemente la interfaz `Collection<T>`.
- Lo mismo sucede con las implementaciones de `Map<K,V>`.
- Colecciones y aplicaciones no son intercambiables.
- Todas las implementaciones descritas son modificables (implementan los métodos etiquetados como opcionales).
- Todas implementan `Cloneable` (y `Serializable`).

Convenciones sobre excepciones

- Las clases e interfaces de colecciones siguen las siguientes convenciones:
 - Los métodos *opcionales* “no implementados” lanzan la excepción
 - UnsupportedOperationException
 - Los métodos y constructores con elementos a ser incluidos en la correspondiente colección como argumentos lanzan la excepción
 - ClassCastException si dichos elementos no son del tipo apropiado
 - IllegalArgumentException si el valor del elemento no es apropiado para la colección
 - Los métodos que devuelven un elemento lanzan la excepción
 - NoSuchElementException si la colección es vacía
 - Los métodos y constructores que toman un parámetro de tipo referencia suelen lanzar una excepción
 - NullPointerException si se les pasa una referencia null

Algoritmos sobre colecciones

- La clase **java.util.Collections** proporciona:
 - Métodos estáticos públicos que implementan algoritmos polimórficos para varias operaciones sobre colecciones:

```
static void shuffle(List<?> list)

static void reverse(List<?> list)

static <T> void fill(List<? super T> list, T o)

static <T> void copy(List<? super T> dest, List<? extends T> src)

static <T> int
    binarySearch(List<? extends Comparable<? super T>> list, T key )

static <T extends Comparable<? super T>> void sort(List<T> list)

static <T extends Object & Comparable<? super T>> T
    max(Collection<T> coll)
```
 - Métodos para la creación de instancias de colecciones (fábricas de instancias o *factory methods*).

La interfaz Collection<T>

```
public interface Collection<T> extends Iterable<T> {  
    // Operaciones básicas  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(T element); // Opcional  
    boolean remove(Object element); // Opcional  
  
    // Operaciones con grupos de elementos  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends T> c); // Opcional  
    boolean removeAll(Collection<?> c); // Opcional  
    boolean retainAll(Collection<?> c); // Opcional  
    void clear(); // Opcional  
  
    // Operaciones con arrays  
    Object[] toArray();  
    <S> S[] toArray(S a[]);  
}
```

La interfaz Set<T>

- La interfaz Set<T> hereda de la interfaz Collection<T>.

```
public interface Set<T> extends Collection<T> {  
    }  
}
```
- No permite elementos duplicados (control con equals()).
- Los métodos definidos permiten realizar lógica de conjuntos:

a.containsAll(b)

$b \subseteq a$

a.addAll(b)

$a = a \cup b$

a.removeAll(b)

$a = a - b$

a.retainAll(b)

$a = a \cap b$

a.clear()

$a = \emptyset$

Implementaciones de `Set<T>`

`java.util` proporciona una implementación de `Set<T>`:

- `HashSet<T>`

- Guarda los datos en una tabla hash.
- Búsqueda, inserción y eliminación en tiempo (casi) constante.
- Constructores:
 - Sin argumentos,
 - con una colección como parámetro, y
 - constructores en los que se puede indicar la capacidad y el factor de carga de la tabla.

Ejemplo: uso de Set<T>

```
import java.util.*;

public class Duplicados {
    public static void main(String[] args) {
        Set<String> s = new HashSet<>();
        for (String arg : args) {
            if (!s.add(arg)) {
                System.out.println("duplicado: " + arg);
            }
        }
        System.out.println(
            s.size() + " palabras detectadas: " + s);
    }
}
```

SALIDA:	<pre>> java Duplicados uno dos cuatro dos tres cuatro cinco duplicado: dos duplicado: cuatro 5 palabras detectadas: [tres, dos, uno, cinco, cuatro]</pre>
----------------	--

La interfaz `Iterable<T>`

- La interfaz `Collection<T>` hereda de la interfaz `Iterable<T>`.

- Esta interfaz incluye un único método

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

- El método devuelve una instancia de alguna clase que implemente la interfaz `Iterator<T>`.
 - Con esta clase podemos realizar recorridos (iteraciones) sobre la colección.
 - Normalmente, esta clase se encuentra anidada no estática (no genérica) en la clase colección sobre la que va a iterar.

```
Collection<String> c = ...;  
Iterator<String> iter = c.iterator();
```

La interfaz `Iterator<T>`

- Un iterador permite el acceso secuencial a los elementos de una colección.

```
public interface Iterator<T> {  
    boolean hasNext();  
    T next();  
    default void remove() {  
        throw new UnsupportedOperationException("remove");  
    }  
}
```

- El método `remove()` permite quitar elementos de la colección.
 - Ésta es la única forma en que se recomienda se eliminen elementos durante la iteración (`ConcurrentModificationException`).
 - Sólo puede haber un mensaje `remove()` por cada mensaje `next()`. Si no se cumple se lanza una excepción `IllegalStateException`.
 - Si no hay siguiente `next()` lanza una excepción `NoSuchElementException`.

Ejemplo: uso de iteradores

- Mostrar una colección en pantalla.

```
static <T> void mostrar(Collection<T> c) {  
    Iterator<T> iter = c.iterator();  
    System.out.print("< ");  
    while (iter.hasNext()) {  
        System.out.print(iter.next() + " ");  
    }  
    System.out.println(">");  
}
```

- Eliminar las cadenas largas de una colección de cadenas.

```
static void filtro(Collection<String> c, int maxLong) {  
    Iterator<String> iter = c.iterator();  
    while (iter.hasNext()) {  
        if ((iter.next()).length() > maxLong) {  
            iter.remove();  
        }  
    }  
}
```


Nueva construcción **for**

- La sentencia `for` se ha extendido de manera que permite una nueva sintaxis. Ejemplo:

```
public <T> void mostrar(List<T> lista) {  
    Iterator<T> it = lista.iterator();  
    while (it.hasNext()) {  
        System.out.println(it.next());  
    }  
}
```

- Puede escribirse alternativamente como

```
public <T> void mostrar(List<T> lista) {  
    for(T t : lista) {  
        System.out.println(t);  
    }  
}
```

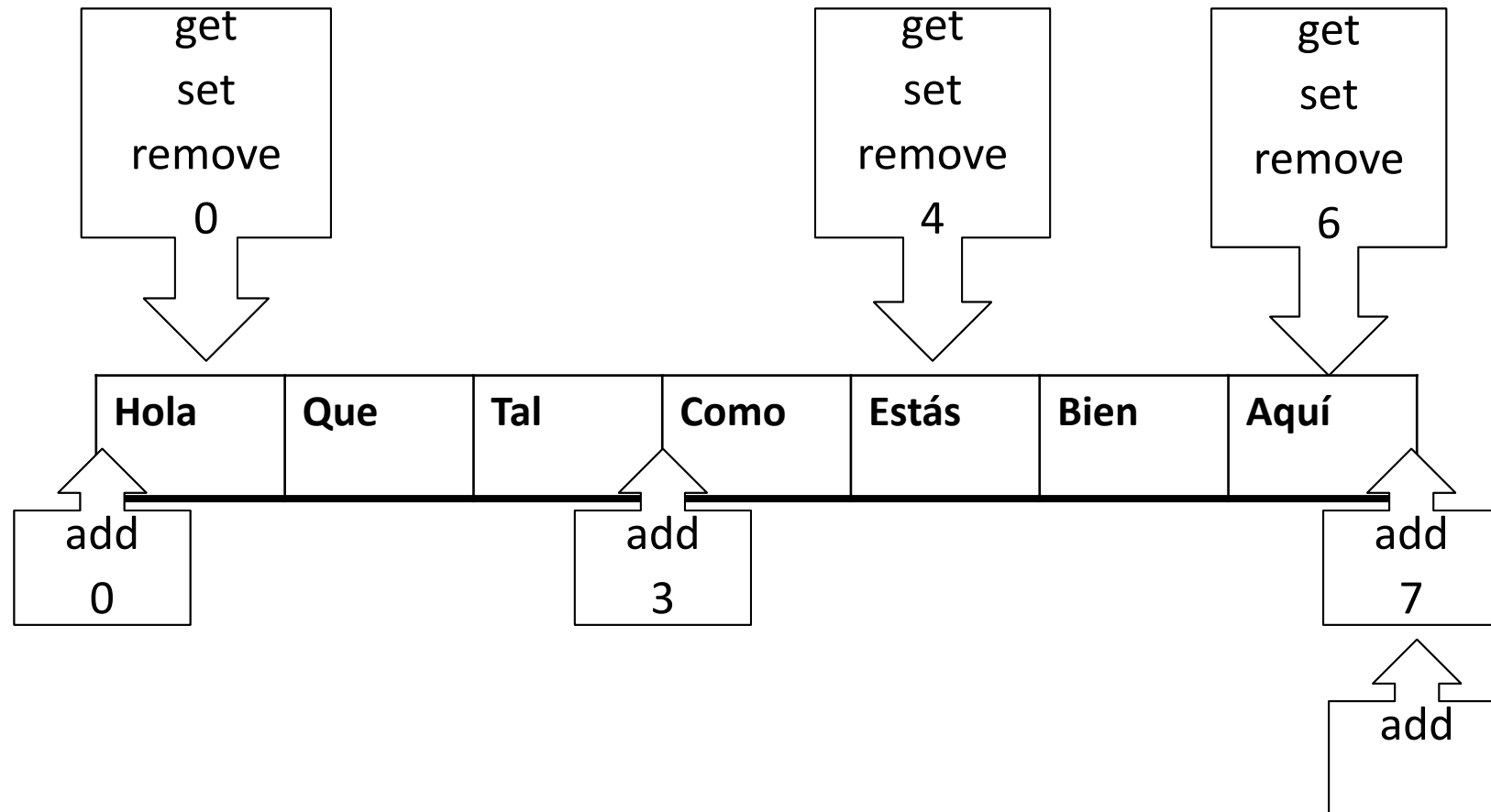
La interfaz `List<T>`

- Colección de elementos ordenados (por su posición).
 - Acceso por posición numérica ($0 \dots \text{size()} - 1$).
 - Un índice ilegal produce el lanzamiento de una excepción `IndexOutOfBoundsException`.
 - Iteradores especializados (`ListIterator<T>`).
 - Realiza operaciones con rangos de índices.

La interfaz `List<T>`

```
public interface List<T> extends Collection<T> {  
  
    // Acceso posicional  
    T get(int index);  
    T set(int index, T element); // Opcional  
    void add(int index, T element); // Opcional  
    T remove(int index); // Opcional  
    boolean addAll(int index, Collection<? extends T> c); // Opcional  
  
    // Búsqueda  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // Iteración  
    ListIterator<T> listIterator();  
    ListIterator<T> listIterator(int index);  
  
    // Vista de subrango  
    List<T> subList(int from, int to);  
}
```

La interfaz `List<T>`



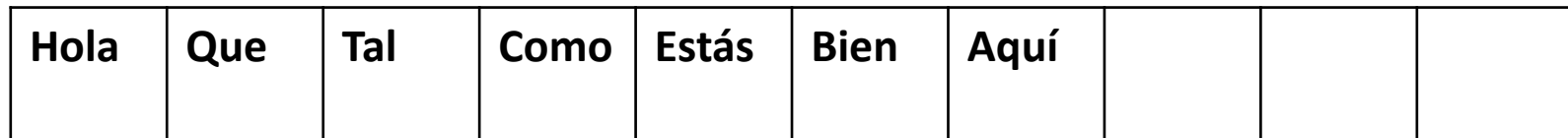
Implementaciones de `List<T>`

- `java.util` proporciona tres implementaciones de `List<T>`:
 - `ArrayList<T>`
 - ✓ Array redimensionable dinámicamente.
 - ✓ Inserción y eliminación (al principio) ineficientes.
 - ✓ Creación y consulta rápidas.
 - `Vector<T>`
 - ✓ Array redimensionable dinámicamente.
 - ✓ Operaciones concurrentes no comprometen su integridad (*thread-safe*).
 - `LinkedList<T>`
 - ✓ Lista (doblemente) enlazada.
 - ✓ Inserción rápida, acceso aleatorio ineficiente.
- Constructores:
 - Sin argumentos o con una colección como parámetro.
 - `ArrayList<T>` y `Vector<T>` tienen un tercer constructor en el que se puede indicar la capacidad inicial.
- Métodos especiales de `LinkedList<T>`:
 - void `addFirst(T o)`
 - void `addLast(T o)`
 - T `getFirst()`
 - T `getLast()`
 - T `removeFirst()`
 - T `removeLast()`

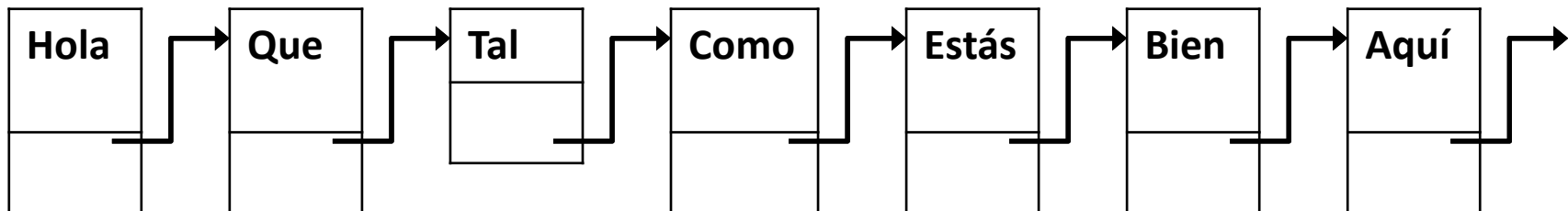
Implementaciones de `List<T>`

ArrayList

Vector



LinkedList



```
import java.util.*;
```

```
public class Shuffle {
```

```
    public static void main(String args[]) {
```

```
        // creamos la lista original
```

```
        List<String> original = new ArrayList<>();
```

```
        for (String arg : args) {
```

```
            original.add(arg);
```

```
        }
```

```
        // creamos la copia y la desordenamos
```

```
        List<String> duplicado = new ArrayList<>(original);
```

```
        Collections.shuffle(duplicado);
```

```
        // comparamos las dos copias con sendos iteradores
```

```
        Iterator<String> iterOriginal = original.iterator();
```

```
        Iterator<String> iterDuplicado = duplicado.iterator();
```

```
        int mismoSitio = 0;
```

```
        while (iterOriginal.hasNext()) {
```

```
            if (iterOriginal.next().equals(iterDuplicado.next())) {
```

```
                mismoSitio++;
```

```
            }
```

```
        }
```

```
        //mostramos el resultado en pantalla
```

```
        System.out.println(
```

```
            duplicado + ": " + mismoSitio + " en el mismo sitio.");
```

```
    }
```

```
}
```

Ejemplo: uso de **List<T>**

SALIDA: > java Shuffle uno dos tres cuatro cinco
 [cinco, dos, uno, tres, cuatro]: 1 en el mismo sitio.

La interfaz Queue<T>

```
public interface Queue<T> extends Collection<T> {  
  
    // Obtener el primero sin quitarlo  
    T element();           // NoSuchElementException si está vacía  
  
    T peek();              // null si está vacía  
  
    // Eliminar el primero (y devolverlo)  
    T remove();            // NoSuchElementException si está vacía  
  
    T poll();              // null si está vacía  
  
    // Introducir un elemento  
    boolean add(T e);      // IllegalStateException si no cabe  
  
    boolean offer(T e);    // false si no cabe  
}
```

- La clase **LinkedList<T>** implementa esta interfaz.

La interfaz `Map<K,V>`

- `Map<K,V>` define aplicaciones (*mappings*) de claves a valores.
 - Las claves son únicas (se controla con `equals()`).
 - Cada clave puede emparejarse con a lo sumo un valor.
- Una aplicación no es una colección, y por esto la interfaz `Map<K,V>` no hereda de `Collection<T>`. Sin embargo, una aplicación puede ser vista como una colección de varias formas:
 - un conjunto de claves,
 - una colección de valores, o
 - un conjunto de pares `<clave, valor>`.
- Como en `Collection<T>`, algunas de las operaciones son *opcionales*, y si se invoca una operación no implementada se lanza la excepción `UnsupportedOperationException`.
 - Las implementaciones del paquete `java.util` implementan todas las operaciones.
- Dos implementaciones: `HashMap<T>` y `Hashtable<T>` (*thread-safe*).
 - Con constructores estándares,
 - con una aplicación, y
 - otros en los que se puede especificar capacidad y factor de carga.

La interfaz Map<K, V>

```
public interface Map<K,V> {  
    // Operaciones básicas  
    V put(K key, V value); // opcional  
    default V putIfAbsent(K key, V value)  
    V get(Object key);  
    default V getOrDefault(Object key, V defaultValue)  
    default V remove(Object key); // opcional  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
  
    // Operaciones con grupos de elementos  
    void putAll(Map<? extends K,? extends V> t); // opcional  
    void clear(); // opcional  
  
    // Vistas como colecciones  
    Set<K> keySet();  
    Collection<V> values();  
    Set<Map.Entry<K,V>> entrySet();  
  
    // Interfaz para los pares de la aplicación  
    interface Entry<K,V> {  
        K getKey();  
        V getValue();  
        V setValue(V value);  
        ...  
    }  
}
```

La interfaz `Map<K, V>`

Los métodos `getOrDefault` y `putIfAbsent` son métodos por defecto introducidos en java1.8:

```
default V getOrDefault(Object key, V defaultValue) {  
    V v = get(key);  
    return (v != null) ? v : defaultValue;  
}
```

- Si `key` se encuentra en el diccionario devuelve su valor. En otro caso devuelve `defaultValue`

```
default V putIfAbsent(K key, V value) {  
    V v = get(key);  
    if (v == null) {  
        v = put(key, value);  
    }  
    return v;  
}
```

- Si `key` no se encuentra en el diccionario lo asocia con `value` y devuelve null.
- En otro caso no hace nada y devuelve el valor asociado

Ejemplo:

Map<K, V>

```
import java.util.*;

public class Frecuencias {
    public static void main(String[] args) {
        Map<Integer, Integer> frecs = new HashMap<>();
        for (String arg : args) {
            // Incr. la frec. de la cad., o la pone a 1 si es la 1ª
            int enteroArg= Integer.parseInt(arg);
            int frec = frecs.getDefault(enteroArg,0);
            frecs.put(enteroArg, frec + 1);
        }
        // Mostramos frecs. iterando sobre el conjunto de claves
        for (Integer clave: frecs.keySet()) {
            int frec = frecs.get(clave);
            char[] barra = new char[frec];
            Arrays.fill(barra, '*');
            System.out.println(clave + ":\t" + new String(barra));
        }
    }
}
```

```
java Frecuencias 5 4 32 3 4 3 2 3 4 2 5 2 3
```

5:	**
4:	***
3:	****
2:	***
32:	*

Ejemplo:

Map<K, V>

```
import java.util.*;

public class Frecuencias {
    public static void main(String[] args) {
        Map<Integer, Integer> frecs = new HashMap<>();
        for (String arg : args) {
            // Incr. la frec. de la cad., o la pone a 1 si es la 1ª
            int enteroArg= Integer.parseInt(arg);
            Integer frec = frecs.get(enteroArg);
            if (frec == null) {
                frecs.put(enteroArg, 1);
            } else {
                frecs.put(enteroArg, frec + 1);
            }
        }
        // Mostramos frecs. iterando sobre el conjunto de claves
        for (Integer clave: frecs.keySet()) {
            int frec = frecs.get(clave);
            char[] barra = new char[frec];
            Arrays.fill(barra, '*');
            System.out.println(clave + ":\t" + new String(barra));
        }
    }
}
```

```
java Frecuencias 5 4 32 3 4 3 2 3 4 2 5 2 3
```

5:	**
4:	***
3:	****
2:	***
32:	*

Clases ordenables

- Una clase puede especificar una relación de orden por medio de:
 - la interfaz **Comparable**<T> (*orden natural*)
 - la interfaz **Comparator**<T> (*orden alternativo*)
- Sólo es posible definir un orden natural, aunque pueden especificarse varios órdenes alternativos.
 - El orden natural se define en la propia clase.

```
public class Persona implements Comparable<Persona> {  
    ...  
}
```
 - Cada uno de los órdenes alternativos debe implementarse en una clase diferente.

```
public class SatPersona implements Comparator<Persona> {  
    ...  
}  
public class OrdPersona implements Comparator<Persona> {  
    ...  
}
```
- Si se intentan comparar dos objetos no comparables se lanza una excepción **ClassCastException**.

La interfaz Comparable<T>

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

- *Orden natural* para una clase.

{	negativo	si receptor	menor	que o
	cero	si receptor	igual	que o
	positivo	si receptor	mayor	que o
- **compareTo()** *no debe* entrar en contradicción con **equals()**.
- Muchas de las clases estándares en la API de Java implementan esta interfaz:

Clase	Orden natural
Byte, Long, Integer, Short, Double y Float	numérico
Character	numérico (sin signo)
String	lexicográfico
Date	cronológico

Orden natural: Persona

```
import java.util.*;
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String nombre() {
        return nombre;
    }
    public int edad() {
        return edad;
    }
    public boolean equals(Object o) {
        boolean res = o instanceof Persona;
        Persona p = res ? (Persona)o : null;
        return res && (edad == p.edad) && (p.nombre.equals(nombre));
    }
    public int hashCode() {
        return (nombre.hashCode() + (new Integer(edad)).hashCode()) / 2;
    }
}
```


Persona implementa Comparable<Persona>

```
...  
// Se comparan por edad, y a igualdad de edad, por nombres  
public int compareTo(Persona p) {  
    int resultado = edad - p.edad;  
    if (resultado == 0) {  
        resultado = nombre.compareTo(p.nombre);  
    }  
    return resultado;  
}
```

Comparable: ejemplo de uso

```
import java.util.*;

public class MainPersona1 {
    public static void main(String [] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        System.out.println(p1.compareTo(p2));
    }
}
```

Conjuntos y aplicaciones ordenadas

- Sabemos que una clase puede especificar una relación de orden por medio de:
 - la interfaz `Comparable<T>` (*orden natural*)
 - la interfaz `Comparator<T>` (*orden alternativo*)
- Estas relaciones se utilizan en conjuntos y aplicaciones ordenadas así como en algoritmos de ordenación.
 - Los objetos que implementan un orden natural o alternativo pueden ser utilizados:
 - como elementos (T) de un conjunto ordenado (`SortedSet<T>`)
 - como claves (K) en una aplicación ordenada (`SortedMap<K, V>`), o
 - en listas ordenables con los métodos `Collections.sort(...)`, ...
 - Por defecto, cuando se requiere una relación de orden se utiliza el orden natural (es decir, el definido en la interfaz `Comparable<T>`).
 - En cualquier caso, es posible indicar un objeto “satélite” (es decir, que implemente la interfaz `Comparator<T>`) para ordenar por la relación alternativa que define en lugar de usar el orden natural.

La interfaz `SortedSet<T>`

- Extiende `Set<T>` para proporcionar la funcionalidad para conjuntos con elementos ordenados.
- El orden utilizado es:
 - el orden natural, o
 - el alternativo dado en un `Comparator<T>` en el constructor.
- `java.util` proporciona la siguiente implementación:
 - `TreeSet<T>`
 - Utiliza árboles binarios equilibrados.
 - Búsqueda y modificación más lenta que en `HashSet<T>`.
 - Constructores:
 - `TreeSet()` // Orden natural
 - `TreeSet(Comparator<? super T> o)` // Orden alternativo o
 - `TreeSet(Collection<? extends T> c)` // Orden natural
 - `TreeSet(SortedSet<T> s)` // Mismo orden que s

La interfaz `SortedSet<T>`

```
public interface SortedSet<T> extends Set<T> {  
    // Vistas de rangos  
    SortedSet<T> headSet(T toElement);  
    SortedSet<T> tailSet(T fromElement);  
    SortedSet<T> subSet(T fromElement, T toElement);  
  
    // elementos mínimo y máximo  
    T first();  
    T last();  
  
    // acceso al comparador  
    Comparator<? super T> comparator();  
}
```

Devuelve el `Comparator<T>` asociado con el conjunto ordenado, o null si éste usa el orden natural.

La interfaz `SortedMap<K, V>`

- Extiende `Map<K,V>` para proporcionar la funcionalidad para aplicaciones con claves ordenadas.
- El orden utilizado es:
 - el orden natural, o
 - el alternativo dado en un `Comparator<K>` en el momento de la creación.
- `java.util` proporciona la siguiente implementación :
 - `TreeMap<K,V>`
 - Utiliza árboles binarios equilibrados.
 - Búsqueda y modificación más lenta que en `HashMap<K,V>`.
 - Constructores:
 - `TreeMap()` // Orden natural
 - `TreeMap(Comparator<? super K> o)` // Orden alternat. o
 - `TreeMap(Map<? extends K,? super V> c)` // Orden natural
 - `TreeMap(SortedMap<K,? extends V> s)` // Mismo orden que s

La interfaz `SortedMap<K, V>`

```
public interface SortedMap<K,V> extends Map<K,V>{  
    // Vistas de rangos  
    SortedMap<K,V> headMap(K toKey);  
    SortedMap<K,V> tailMap(K fromKey);  
    SortedMap<K,V> subMap(K fromKey, K toKey);  
  
    // claves mínima y máxima  
    T firstKey();  
    T lastKey();  
  
    // acceso al comparador  
    Comparator<? super K> comparator();  
}
```

Ejemplo: frecuencias

SortedMap<K, V>

```
import java.util.*;

public class Frecuencias {
    public static void main(String[] args) {
        SortedMap<Integer,Integer> mFrecs = new TreeMap<>();
        for (String arg : args) {
            // Incr. la frec. de la cad., o la pone a 1 si es la 1ª
            int num = Integer.parseInt(arg);
            int frec = mFrecs.getOrDefault(num,0);
            mFrecs.put(num, frec + 1);
        }
        // Muestra frecs. de subrango iterando sobre conj. ordenado de entradas
        SortedMap<Integer,Integer> subFrecs = mFrecs.subMap(1, 10);
        for (Map.Entry<Integer,Integer> entrada : subFrecs.entrySet()) {
            int valor = entrada.getKey();
            int frec = entrada.getValue();
            char[] barra = new char[frec];
            Arrays.fill(barra, '*');
            System.out.println(valor + ":\t" + new String(barra));
        }
    }
}
```

java Frecuencias 5 4 3 2 3 4 3 2 3 4 2 5 2 3

2:	***
3:	****
4:	***
5:	**

Ejemplo: frecuencias

SortedMap<K, V>

```
import java.util.*;

public class Frecuencias {
    public static void main(String[] args) {
        SortedMap<Integer,Integer> mFrecs = new TreeMap<>();
        for (String arg : args) {
            // Incr. la frec. de la cad., o la pone a 1 si es la 1ª
            int num = Integer.parseInt(arg);
            Integer frec = mFrecs.get(num);
            if (frec == null) {
                mFrecs.put(valor, 1);
            } else {
                mFrecs.put(valor, frec + 1);
            }
        }
        // Muestra frecs. de subrango iterando sobre conj. ordenado de entradas
        SortedMap<Integer,Integer> subFrecs = mFrecs.subMap(1, 10);
        for (Map.Entry<Integer,Integer> entrada : subFrecs.entrySet()) {
            int valor = entrada.getKey();
            int frec = entrada.getValue();
            char[] barra = new char[frec];
            Arrays.fill(barra, '*');
            System.out.println(valor + ":\t" + new String(barra));
        }
    }
}
```

java Frecuencias 5 4 3 2 3 4 3 2 3 4 2 5 2 3

2:	***
3:	****
4:	***
5:	**

Ejemplo: Contar posiciones

SortedMap<K, V>

```
import java.util.*;

public class Posiciones{
    public static void main(String[] args) {
        SortedMap<Integer,List<Integer>> mPos = new TreeMap<>();
        for (int i = 0; i < args.length; i++) {
            int num = Integer.parseInt(args[i]);
            // Buscamos la lista asociada a num en mPos
            List<Integer> lPos= mPos.getOrDefault(num, new ArrayList<Integer>());
            mPos.putIfAbsent(num,lPos); // y se asocia(si no lo estaba) a num en mPos
            // PosCondición: lPos existe y está asociado a num en mPos
            lPos.add(i);
        }
        for (Map.Entry<Integer,Integer> entrada : mPos.entrySet()) {
            int num = entrada.getKey();
            List<Integer> lPos = entrada.getValue();
            System.out.println(num + ":\t" + lPos);
        }
    }
}
```

```
java Posiciones 5 4 32 3 4 3 2 3 4 2 5 2 3
```

2:	[6, 9, 11]
3:	[3, 5, 7, 12]
4:	[1, 4, 8]
5:	[0, 10]
32:	[2]

Ejemplo: Contar posiciones

SortedMap<K, V>

```
import java.util.*;
```

```
public class Posiciones{
    public static void main(String[] args) {
        SortedMap<Integer,List<Integer>> mPos = new TreeMap<>();
        for (int i = 0; i < args.length; i++) {
            int num = Integer.parseInt(args[i]);
            // Buscamos la lista asociada a num en mPos
            List<Integer> lPos= mPos.get(num);
            if (lPos == null) {
                lPos = new ArrayList<Integer>();      // se crea lPos
                mPos.put(num,lPos)                    // y se asigna a num en mPos
            }
            // PosCondición: lPos existe y está asociado a num en mPos
            lPos.add(i);
        }
        for (Map.Entry<Integer,Integer> entrada : mPos.entrySet()) {
            int num = entrada.getKey();
            List<Integer> lPos = entrada.getValue();
            System.out.println(num + ":\t" + lPos);
        }
    }
}
```

```
java Posiciones 5 4 32 3 4 3 2 3 4 2 5 2 3
```

2:	[6, 9, 11]
3:	[3, 5, 7, 12]
4:	[1, 4, 8]
5:	[0, 10]
32:	[2]

Ordenando

- La interfaz `List` incluye desde java 1.8 un método por defecto que permite ordenar listas:

```
default void sort(Comparator<? super E> c) {  
    ...  
}
```

— Si `c` es `null` se usa el orden natural.

- Recordemos que en `Collections` existen los métodos

```
default void sort(List<E> list, Comparator<? super E> c);  
default void sort(List<E> list);
```