Colecciones e Iteradores

Contenido

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

Clases genéricas

- 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. Este 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 (un record) que almacene dos elementos de otra clase.
  - No indicamos de qué clase son los elementos a almacenar.
  - Supongamos que son de la clase T donde T representa a cualquier clase.

```
public class Par <T>{
    private T primero, segundo;
    public Par(T p, T s) {
            primero = p;
            segundo = s;
    }
    public T getPrimero() {
            return primero;
    }
    public T getSegundo() {
            return segundo;
    }
    public void setPrimero(T p) {
            primero = p;
    }
    public void setSegundo(T s) {
            segundo = s;
    }
}
```

¿Cómo usar los objetos de esa clase?

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

Añadiendo <T> a la cabecera

```
public class Programa {
   public static void main(String[] args) {
      Par<String> conC = new Par<>("hola", "adios");
      Par<Integer> conI = new Par<>(4, 9);
   }
}
```

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<>("hola", "adios");
```

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

$$\texttt{<T extends A \& } I_1 \texttt{ \& } I_2 \texttt{ \& … \& } I_n \texttt{>}$$

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 getPrimero() {
            return primero;
        }

        public B getSegundo() {
            return segundo;
        }

        public void setPrimero(A a) {
            primero = a;
        }

        public void setSegundo(B b) {
            segundo = b;
        }
}
```

```
Pareja<String,Integer> p = new Pareja<>("hola", 10);
```

Tuplas nombradas genéricas

- Las tuplas nombradas también pueden ser genéricas:

```
public record Tupla2<A,B>(A fst, B snd){};
```

```
Tupla2<String,Integer> p = new Tupla2<>("hola", 10);
System.out.println(p.fst());       // "hola"
System.out.println(p.snd());       // 10
System.out.println(p);             // Tupla2[fst="hola", snd=10]
```

Genericidad y herencia

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

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();
```

Métodos genéricos

- Un método también puede incluir parámetros formales que representen clases o interfaces.
  - Supongamos una clase no genérica con el siguiente método:

```
public class Programa {
    static public <A,B> String aCadena(Pareja<A, B> par) {
            return "(" + par.primero() + "," + par.segundo() + ")";
    }
}
```

¿Cómo sabe que A y B son clases genéricas?

Añadiendo **<A,B>** a la cabecera del método

```
…
Pareja<String, Integer> p = new Pareja<>("hola", 10);

System.out.println(Programa.aCadena(p));
…
```

Parámetros anónimos (I)

- Cuando un parámetro formal no se utiliza en el cuerpo del método puede utilizarse el símbolo "?" del modo siguiente:

```
public class Programa {
    public static String aCadena(Pareja<?,?> par) {
       return "(" + par.primero() + "," + par.segundo() + ")";
    }
}
```

- Es equivalente a:

```
public class Programa {
    public static <A,B> String aCadena(Pareja<A,B> par) {
       return "(" + par.primero() + "," + par.segundo() + ")";
    }
}
```

Parámetros anónimos (II)

- Sobre un parámetro anónimo se pueden especificar además restricciones: "la clase anónima debe ser superclase de una clase dada".

`<? super T>`

```
class Programa {
   static public <T> void copiaPrimero(Par<T> orig, Par<? super T> dest) {
      dest.primero(orig.primero());
   }
   ...
}
```

El método `copiaPrimero(Par<T>, Par<? super T>)` sí es aplicable en la forma `copiaPrimero(Par<CocheImportado>, Par<Coche>)`

Para poder copiar el primer elemento de `orig` en el primero de `dest`, es necesario asegurar que la clase de los elementos de `dest` sea superclase de la clase de los elementos de `orig` (es decir, `T`).

De hecho, la mejor manera de definir el método anterior sería:

```
class Programa {
   static public <T> void
               copiaPrimero(Par<? extends T> orig, Par<? super T> dest) {
         dest.primero(orig.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:
  - Algoritmos. Para realizar determinadas operaciones sobre colecciones, como ordenaciones, búsquedas, etc.
  - Interfaces. Para manipularlas de forma independiente de la implementación.
  - Implementaciones. Implementan la funcionalidad de alguna manera.
- 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.

Algoritmos sobre colecciones

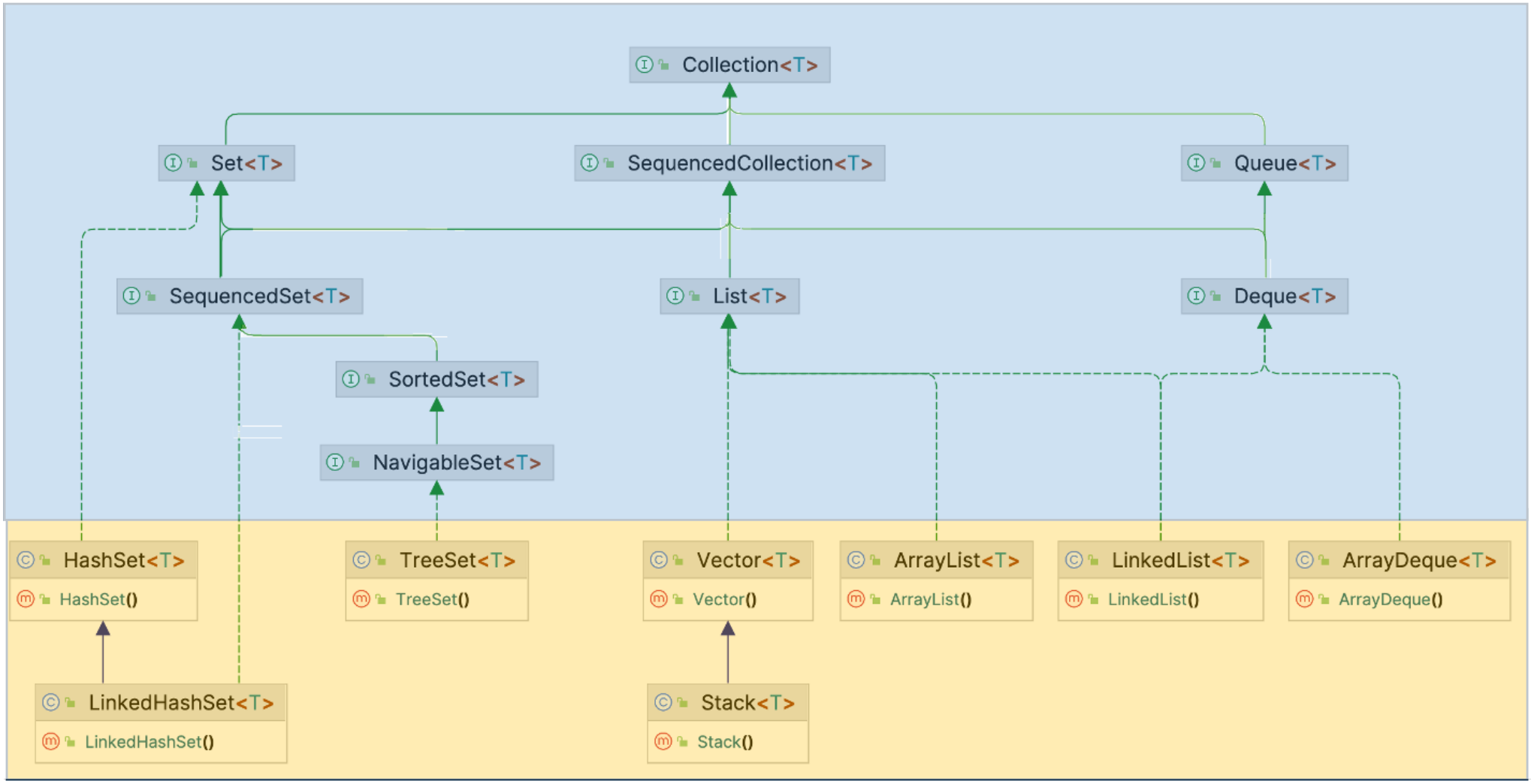
- La clase **`java.util.Collections`** proporciona:

  - Métodos estáticos públicos que implementan algoritmos polimórficos para varias operaciones sobre colecciones:

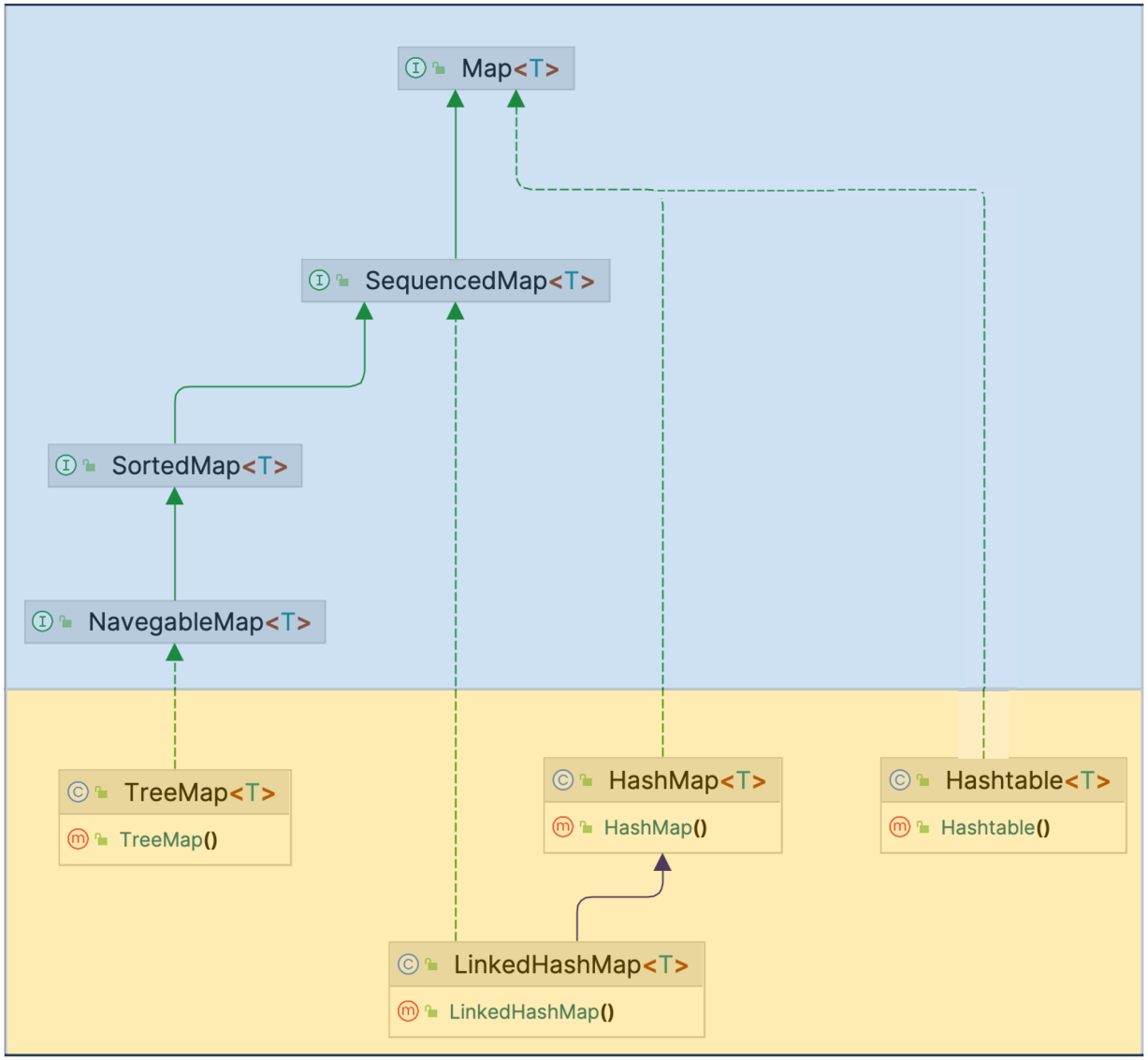
```
static void shuffle(List<?> list)
static void reverse(List<?> list)
static <T> void fill(List<? super T> list, T o)
static <T> void copy(List<? super T> dest, List<? extends T> src)
static <T> int
   binarySearch(List<? extends Comparable<? super T>> list, T key)
static <T extends Comparable<? super 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*).

Colecciones: Interfaces e implementaciones



Correspondencias. Interfaces e implementaciones



Resumen

Qué Cómo

```
Interfaz<...> col = new Implementacion<>();
```

Si Interfaz es:	La implemetación puede ser
Collection	HashSet, TreeSet, ArrayList, LinkedList, LinkedHashSet, ArrayDequeue
Set	HashSet, TreeSet, LinkedHashSet
List	ArrayList, LinkedList
Queue	ArrayDequeue, LinkedList
SortedSet	TreeSet
NavigableSet	TreeSet
Map	HashMap, TreeMap, LinkedHashMap
SortedMap	TreeMap
NavigableMap	TreeMap

Implementaciones

- No hay implementación directa de la interfaz `Collection<T>`, esta se utiliza solo para mejorar la inter-operación 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 una clase que implemente la interfaz `Collection<T>`.
- Lo mismo sucede con las implementaciones de `Map<K,V>`.
- Colecciones y correspondencias no son intercambiables.
- Todas las implementaciones descritas son modificables (implementan los métodos etiquetados como opcionales).
- Todas implementan `Cloneable` (y `Serializable`).

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
  default boolean removeIf(Predicate<? super T> pred);

  // 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[]);

  // Operaciones sobre stream
  Stream<T> stream();
}
```

La interfaz `Iterable<T>`

- La interfaz `Collection<T>` hereda de la interfaz `Iterable<T>`.
  - Esta interfaz incluye los métodos

```
public interface Iterable<T> {
   Iterator<T> iterator();
   void forEach(Consumer<? Super T> action);
}
```

- El método devuelve una instancia de alguna clase que implemente la interfaz `Iterator<T>`.
  - Con esta clase podemos realizar recorridos (iteraciones) sobre la colección.

```
Collection<String> c = new LinkedList<>();
Iterator<String> iter = c.iterator();
c.forEach(elem -> System.out.println(elem));
```

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();
    }
}
```

- El método `remove()` permite quitar elementos de la colección.
  - Por defecto lanza una excepción.
  - Esta es la única forma en que se recomienda se eliminen elementos durante la iteración (`ConcurrentModificationException`).
  - Solo 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(Collection<T> lista) {
      Iterator<T> it = lista.iterator();
      while (it.hasNext())
            System.out.println(it.next());
}
```

- Puede escribirse alternativamente como

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

La interfaz `Set<T>` extiende a `Collection<T>`

```
public interface Set<T> extends Collection<T> {
      static <E> Set<E> of(E e1) {...}     // inmodificables
      static <E> Set<E> of(T e1, T e2) {...} // inmodificables
      ...
      static <E> Set<E> of (T e1, ..., T e10) {...} // inmodificable
      static <E> Set<E> copyOf (Collection<? extends E> col) {...} // inmodificable
}
```

- No permite elementos duplicados.
- Los métodos definidos permiten realizar lógica de conjuntos:

`a.containsAll(b)`	$b \subseteq a$
`a.addAll(b)`	$a = aUb$
`a.removeAll(b)`	$a = a - b$
`a.retainAll(b)`	$a = a \cap b$
`a.clear()`	$a = \emptyset$

Implemetación de `Set<T>`

`java.util` proporciona una implementación directa 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.

```
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: > java Duplicados a b a f b c
duplicado: a
duplicado: b
4 palabras detectadas: [a, b, c, f]

La interfaz `SequencedCollection<T>` extiende a `Collection<T>`

- Colección de elementos secuenciados.

```
void addFirst(T o)
void addLast(T o)
T getFirst()
T getLast()
T removeFirst()
T removeLast()
SequencedCollection<T> reversed()
```

La interfaz **`SequencedSet<T>`** extiende a **`Set<T>`** y **`SequencedCollection<T>`**

```
SequencedSet<T> reversed()
```

`java.util` proporciona una implementaciones directa de esta interfaz:

- `LinkedHashSet<T>`
  - Hereda de **`HashSet<T>`**
  - Además, tiene una lista doblemente enlazada con los elementos
  - Un iterador recorre los elementos en el orden de inserción.

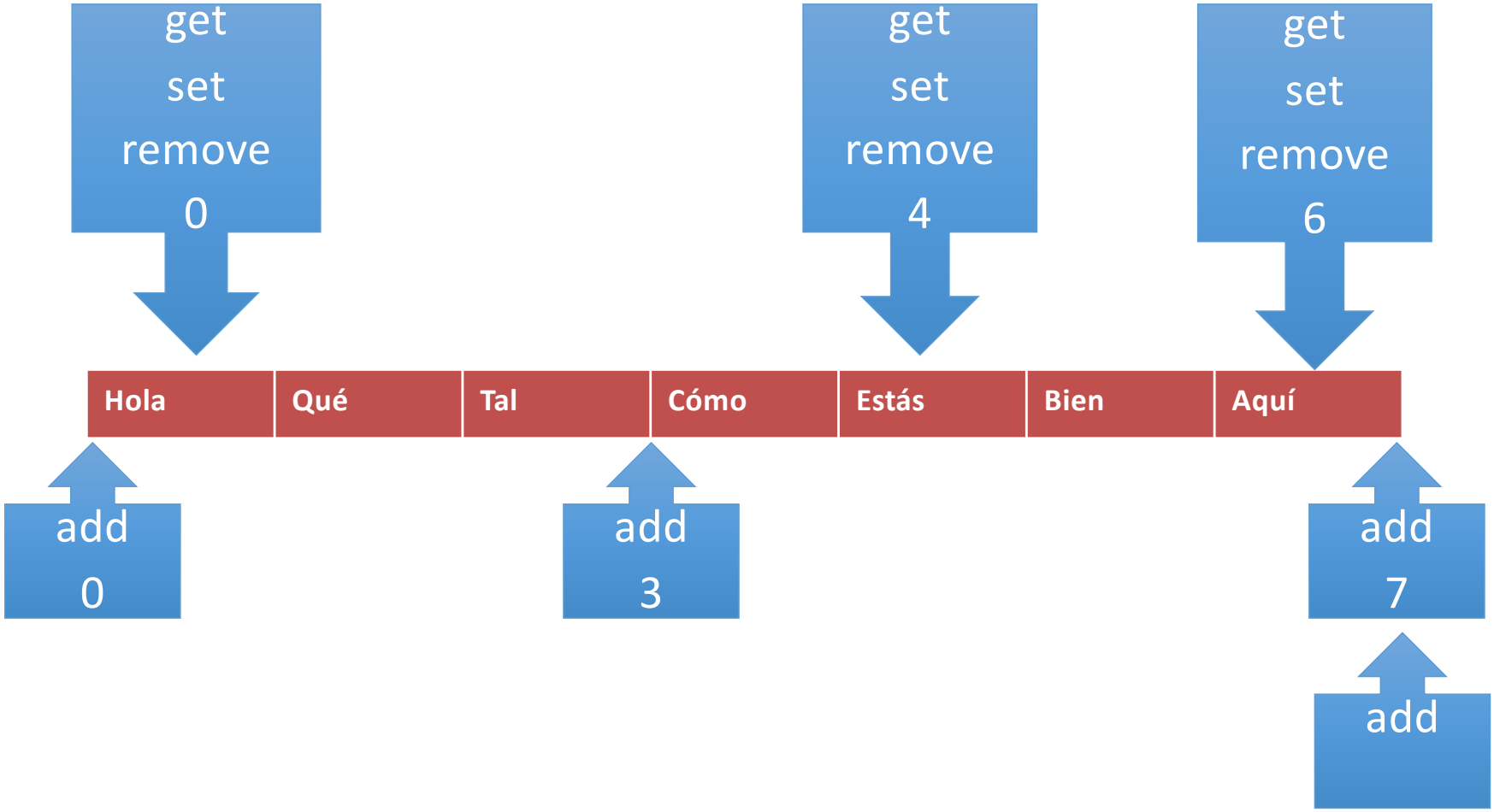
La interfaz `List<T>` extiende a `SequencedCollection<T>`

- Colección de elementos ordenados (por su posición).
  - Acceso por posición numérica (`0 ... 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 SequencedCollection<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
  static <E> List<E> of(E e1) {...}; // inmodificable
  ...
  static <E> List<E> of(E e1, ..., E e10) {...};                // inmodificable
  static <E> List<E> copyOf (Collection<? extends E> col) {...}; // inmodificable
   // 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>`



La interfaz `ListIterator<T>`

```
public interface ListIterator<T> extends Iterator<T> {
        // boolean hasNext();
        // T next();

        boolean hasPrevious();
        T previous();

        int nextIndex();
        int previousIndex();

        // void remove();                  // Opcional
        void set(T o);                     // Opcional
        void add(T o);                     // Opcional
}
```

Implementaciones de `List<T>`:

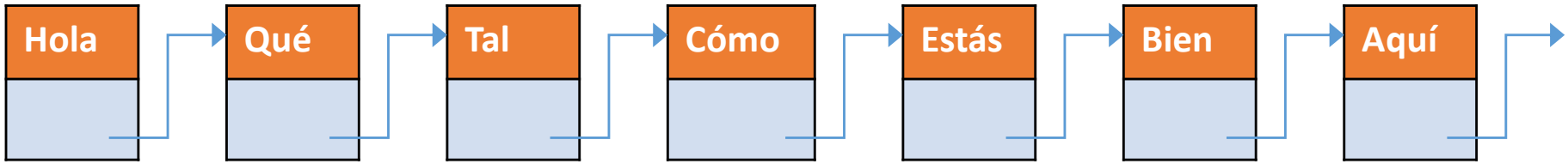
- **`java.util`** proporciona tres implementaciones de `List<T>`:
  - `ArrayList<T>`
    - ✓ Array redimensionable dinámicamente.
    - ✓ Inserción y eliminación (al principio) ineficientes.
    - ✓ Creación y consulta rápidas.
  - `Vector<T>`
    - ✓ Array redimensionable dinámicamente.
    - ✓ Operaciones 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.

Implementaciones de `List<T>`

ArrayList Vector

LinkedList



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

```
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.");
   }
}
```

SALIDA: > java Shuffle a b a b c c b c c b a d a f e e f a
[f, c, a, b, a, e, f, b, c, a, c, a, d, c, a, e, b, b]: 4 en el mismo sitio.

La interfaz `Queue<T>` extiende a `Collection<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 interfaz `Deque<T>` extiende a `Queue<T>` y `SequencedCollection<T>`

```
T peekFirst()
T peekLast()
T pollFirst()
T pollLast()
```

- Las clases `ArrayDequeue<T>` y `LinkedList<T>` implementan esta interfaz.

Correspondencias

- Son asociaciones de claves a valores.
  - Parametrizadas por la clave y el valor.
- Las claves deben ser únicas.
- Cada clave tiene asociado a lo sumo un valor.
- Si se intenta extraer el valor asociado a una clave que no existe se devuelve null.

La interfaz `Map<K,V>`

- `Map<K,V>` define correspondencias (*mappings*) de claves a valores.

- Una correspondencia no es una colección, y por esto la interfaz `Map<K,V>` no hereda de `Collection<T>`. Sin embargo, una correspondencia 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.

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)
  default V computeIfAbsent(K key, Function<? super T, ? extends V> mapp)
  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
  static <C,V> Map<C,V> of (C c1, V v1) {...}                         // inmodificable
  ...
  static <C,V> Map<C,V> of (C c1, V v1,..., C c10, V v10)) {...}      // inmodificable
  static <C,V> Map<C,V> copyOf (Map<? extends C, ? extends V> col) {...};// inmodificable
  ...
}
```

La interfaz **Map<K,V>**

```
  ...
  // 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 la correspondencia 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 la correspondencia lo asocia con `value`. Devuelve null si no está o el valor asociado si está.

La interfaz **Map<K,V>**

```
default V computeIfAbsent(K key,
                  Function<? super K, ? extends V> mapF) {
      V v = get(key));
      if (v == null) {
            v = mapF.apply(key);
            put(key, v);
      }
      return v;
}
```

Si la clave está en la correspondencia, devuelve el valor asociado. Si no está, ejecuta la función segundo argumento con la clave y el valor devuelto por la misma es asignado en el diccionario a esa clave. Además, éste es el valor que devuelve el método.

Implementación de `Map<K,V>`

- Dos implementaciones:
  - `HashMap<K,V>`
  - `Hashtable<K,V>` (*thread-safe*).
- `HashMap<K,V>`
  - Implementada en una tabla hash.
  - La comprobación de que las claves son únicas se hacen con equals() y hashCode() por lo que deben ser compatibles.
  - Con constructores estándares,
    - Sin argumentos,
    - con una correspondencia, y
    - otros en los que se puede especificar capacidad y factor de carga.

```
Map<String, Integer> frecs = new HashMap<>();
```

Crear una correspondencia rápida

El método estático *of* de la interfaz *Map* permite crear una correspondencia con las claves y valores pasadas por parámetro.

```
Map<String,Integer> map1 = Map.of(  "juan", 23,
                                    "Luis", 24,
                                    "maria",19);
```

Hay métodos *of* desde 1 hasta 10 parejas de argumentos. Para mas de 10 parejas o de forma alternativa puede usarse el método *ofEntries*:

```
Map<String,Integer> map2 = Map.ofEntries(
                              Map.entry("juan", 23),
                              Map.entry("Luis", 24),
                              Map.entry("maria",19));
```

Las correspondencias así creadas son inmodificables. Para hacerlas modificables:

```
Map<String,Integer> map3 = new HashMap<>(map1);
```

Ejemplo: **Map<K,V>**

```
import java.util.*;

public class FrecuenciasM {
  public static void main(String[] args) {
        Map<String, Integer> frecs = new HashMap<>();
        for (String arg : args) {
            // Incr. la frec. De arg., o la pone a 1 si es la 1ª
            int frec = frecs.getOrDefault(arg, 0);
            frecs.put(arg, frec + 1);
        }
        // Mostramos frecs. iterando sobre el conjunto de claves
        for (String valor: frecs.keySet()) {
            int frec = frecs.get(valor);
            char[] barra = new char[frec];
            Arrays.fill(barra, '*');
            System.out.println(valor + ": " + new String(barra));
        }
  }
}
```

java Frecuencias ao ir ea ea ir ea ao ea ir ed

```
ir: ***
ea: ****
ao: **
ed: *
```

La interfaz **SequencedMap<K,V>** extiende a **Map<K,V>**

```
public interface SequencedMap<K,V> extends Map<K,V> {
  V putFirst(K key, V value);
  V putLastt(K key, V value);
  default SequencedSet<K> sequencedKeySet()
  default SequencedCollection<K> sequencedValues()
  default SequencedKeySet<Map.Entry<K,V>> sequencedEntrySet()
       ...
}
```

- **java.util** proporciona una implementación directa de esta interfaz:
  - LinkedHashMap<K,V>
    - Extiende de HashMap<K,V>
    - Además, tiene una lista doblemente enlazada con las entradas
    - Un iterador devuelve las claves en el orden en que se introdujeron

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
`...`	

Comparable y tipos básicos

- Todos los envoltorios XXX de los tipos básicos xxx disponen del método

```
int compare(XXX d1, XXX d2)
```

Que devuelve

- Negativo si el primer argumento es menor que el segundo,
- 0 si son iguales
- Positivo si el primer argumento es mayor que el segundo.

Ejemplo

Integer.compare(45,98)

Double.compare(34.56, 23.76)

Ejemplo: Persona implementa Comparable<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 getNombre() {
        return nombre;
  }
  public int getEdad() {
        return edad;
  }
  public boolean equals(Object o) {
        return (o instanceof Persona p)
                && (edad == p.edad) && (p.nombre.equals(nombre));
  }
  public int hashCode() {
        return Object.hash(nombre, edad);
  }
  ...
```

Persona implementa Comparable<Persona>

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

Ejemplo Comparable<Persona>

```
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));
    }
}
```

La interfaz **Comparator<T>**

- Las clases que necesiten una relación de orden distinta del orden natural han de utilizar clases "satélite" que implementen la interfaz **Comparator<T>**.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    // Nuevas desde java 1.8
    default Comparator<T> reversed(){...};
    default Comparator<T>
                thenComparing(Comparator<T>){...};
    static <T extends Comparable<? super T>> Comparator<T>
                            naturalOrder(){...};
        ...
    }
```

- Orden alternativo para una clase.

$$\begin{cases} \text{negativo} & \text{si o1 menor que o2} \\ \text{cero} & \text{si o1 igual que o2} \\ \text{positivo} & \text{si o1 mayor que o2} \end{cases}$$

- **compare()** *no debe* entrar en contradicción con **equals()**.

Ordenes simples que implementan Comparator<Persona>

```
public class OrdenNombre implements Comparator<Persona> {
    // Se comparan por nombres
    public int compare(Persona p1, Persona p2) {
        return p1.getNombre().compareTo(p2.getNombre());
    }
}

public class OrdenEdad implements Comparator<Persona> {
    // Se comparan por edad
    public int compare(Persona p1, Persona p2) {
        return Integer.compare(p1.getEdad(),p2.getEdad());
    }
}
```

Ejemplo composición Comparator<Persona>

```
import java.util.*;

public class MainPersona2 {
    public static void main(String [] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        Comparator<Persona> op =
                new OrdenEdad().thenComparing(new OrdenNombre());
        System.out.println(op.compare(p1,p2));
        Comparator<Persona> op2 =
                new OrdenNombre().reversed().
                                thenComparing(new OrdenEdad());
        System.out.println(op2.compare(p1,p2));
        Comparator<Persona> op3 =
                new OrdenNombre().reversed().
                     thenComparing(Comparator.naturalOrder());
        System.out.println(op3.compare(p1,p2));
    }
}
```

La interfaz Comparator es funcional!

```
import java.util.*;

public class MainPersona2 {
    public static void main(String [] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        Comparator<Persona> orNombre =
                (p1, p2) -> p1.getNombre().compareTo(p2.getNopmbre));
        Comparator<Persona> orEdad =
                (p1, p2) -> Integer.compare(p1.getEdad(), p2.getEdad());
        // y ahora
        Comparator<Persona> op = orEdad.thenComparing(orNombre);
        System.out.println(op.compare(p1,p2));
        Comparator<Persona> op2 =
                orNombre.reversed().thenComparing(orEdad);
        System.out.println(op2.compare(p1,p2));
        Comparator<Persona> op3 =
                orNombre.reversed()
                        .thenComparing(Comparator.naturalOrder());
        System.out.println(op3.compare(p1,p2));
    }
```

Conjuntos ordenados

- 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 ordenados.
  - Los objetos que implementan un orden natural o alternativo pueden ser utilizados:
    - como elementos de un conjunto ordenado (`TreeSet<T>`)
  - En algoritmos de ordenación.
    - 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 "árbitro" (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>` y `SequencedSet<T>`

- Proporciona más funcionalidad para conjuntos con elementos ordenados.
- El orden utilizado es:
  - El orden natural o
  - Un orden alternativo dado en un `Comparator<T>` en el constructor.

La interfaz `SortedSet<T>`

```
public interface SortedSet<T> extends Set<T>, SequencedSet<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 `NavigableSet<T>` extiende a `SortedSet<T>`

```
public interface NavigableSet<T> extends SortedSet<T> {

  Iterator<T> descendingIterator();
  NavigableSet<T> descendingSet();

  T ceiling(T elem);                // menor elemento >= elem
  T floor(T elem);                  // mayor elemento <= elem
  T higher(T elem); //               // menor elemento > elem
  T lower(T elem);                  // mayor elemento < elem
  ...
}
```

Implementación de **NavigableSet<T>**: La clase **TreeSet<T>**

- Mantiene los elementos ordenados. El orden utilizado es:
  - el orden natural, o
  - el alternativo dado en un `Comparator<T>` en el constructor.

  `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

```
Set<Persona> setpon = new TreeSet<>();  // usa orden natural
Set<Persona> ssetpon = new TreeSet<>();  // usa orden natural

Comparator<Persona> op =
          new OrdenEdad().thenComparing(new OrdenNombre());
Set<Persona> setpoa = new TreeSet<>(op); // usa orden alternativo
```

Correspondencias 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 correspondencias
  - Los objetos que implementan un orden natural o alternativo pueden ser utilizados:
    - como claves en una correspondencia ordenada (`TreeMap<K,V>`)
  - 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 "árbitro" (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 `SortedMap<K,V>` extiende a `Map<K,V>`

- Proporciona más funcionalidad para correspondencias con claves ordenadas.

- El orden utilizado es:
  - el orden natural, o
  - el alternativo dado en un `Comparator<K>` en el momento de la creación.

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
 K firstKey();
 K lastKey();

 // acceso al comparador
 Comparator<? super K> comparator();
}
```

La interfaz `NavigableMap<K,V>` extiende a `SortedMap<K,V>`

```
public interface NavigableMap<K,V> extends SortedMap<K,V> {

 NavigableSet<K> descendingKeySet();
 NavigableMap<K,V> descendingMap();

 K ceilingKey(K elem);              // menor elemento >= elem
 K floorKey(K elem);                // mayor elemento <= elem
 K higherKey(K elem); //            // menor elemento > elem
 K lowerKey(K elem);                // mayor elemento < elem

 ...
}
```

Implementación de `NavigableMap<K,V>`: La clase `TreeMap<K,V>`

`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 alternativo
  - `TreeMap(Map<? extends K,? extends V> c)` // Orden natural
  - `TreeMap(SortedMap<? extends K,? extends V> c)` // mismo orden

```
Map<Persona, Integer> treeon = new TreeMap<>();  // usa orden natural
SortedMap<Persona, Integer> streeon = new TreeMap<>();  // usa orden natural
```

```
Comparator<Persona> op = new OrdenEdad().thenComparing(new OrdenNombre());
Map<Persona, Integer> treepoa = new TreeMap<>(op);   // orden alternativo
```

Ejemplo: frecuencias

Utilizamos SortedMap porque usamos métodos de esa interfaz

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

```
java Frecuencias  a b a b c c b c c b a d a f e e f a
```

```
b:      ****
c:      ****
d:      *
```

Ejemplo: Contar posiciones

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

Utilizamos Map porque NO usamos métodos de SortedMap

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

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

Ejemplo: Contar posiciones

Utilizamos Map porque NO usamos métodos de SortedMap

```
import java.util.*;
public class Posiciones{
  public static void main(String[] args) {
      Map<String,List<Integer>> mPos = new TreeMap<>();
      for (int i = 0; i < args.length; i++) {
            // Buscamos la lista asociada a args[i] en mPos
            List<Integer> lPos=
                  mPos.computeIfAbsent(args[i], key -> new ArrayList<>());
            // PosCondición: lPos existe y está asociado a arg en mPos
            lPos.add(i);
      }
      for (Map.Entry<String,List<Integer>> entrada : mPos.entrySet()) {
            String clave= entrada.getKey();
            List<Integer> lPos = entrada.getValue();
            System.out.println(clave + ":\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]
32:     [2]
4:      [1, 4, 8]
5:      [0, 10]
```

Ordenando

- La interfaz `List` incluye 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 existe los métodos

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

Añadiendo funcionalidad Decoradores

- Las clases decoradoras permiten añadir funcionalidad a las colecciones y aplicaciones:
  - Seguras contra tipos (comprobación dinámica de tipos)
  - Seguras ante tareas (*Thread-safe*)
  - No modificables

- La clase `Collections` proporciona métodos factoría para ello:

```
<E> Collection<E>          synchronizedCollection(Collection<E> c)
<E> List<E>                synchronizedList(List<E> c)
<K,V> Map<K,V>             synchronizedMap(Map<K,V> c)
<E> set<E>                 synchronizedSet(Set<E> c)
<E> SortedSet<E>           synchronizedSortedSet(SortedSet<E> c)
<E> Navigable<E>           synchronizedNavigabledSet(NavigableSet<E> c)
<K,V> SortedMap<K,V>       synchronizedSortedMap(SortedMap<K,V> c)
<K,V> Navigable<K,V>       synchronizedNavigableMap(NavigableMap<K,V> c)


<E> Collection<E>    unmodificableCollection(Collection<? extends E> c)
...
```

La clase Arrays I

- Contiene métodos estáticos que implementan algoritmos sobre arrays de elementos de tipo básico u `Object`.

`Tipo` representa un tipo básico u `Object`

```
static int binarySearch(Tipo[] ar, Tipo key);
```

- Devuelve el índice de la posición del elemento `key` en `ar`.
- Devuelve `-pi-1` si `key` no está, donde `pi` es la posición en la que se debería insertar para mantener `ar` ordenado.

- También existe una versión en la que se puede proporcionar un objeto `Comparator`:

```
static <T> int binarySearch(T[] ar, T key,
                            Comparator<? super T> c);
```

La clase Arrays II

```
static void fill(Tipo[] ar, Tipo key);
```

- Asigna el valor `key` a cada elemento de `ar`.

```
static void sort(Tipo[] ar)
```

- Ordena el array `ar` según el orden natural de los elementos.

- También existe una versión en la que se puede proporcionar un objeto `Comparator`:

```
static <T> void sort(T[] ar, Comparator<? super T> c);
```

- El método que devuelve la representación textual de un array:

```
static String toString(Tipo[] ar);
```