Tema 2.7
Colecciones y Genericidad

Análisis y Diseño de Software

2º Ingeniería Informática

Universidad Autónoma de Madrid

Indice

- **Introducción**
- Interfaces
- Comparando objetos
- Implementaciones
- Algoritmos

- Genericidad

Introducción

- Una **colección** o **contenedor** es un objeto que agrupa múltiples elementos en una unidad
- Las colecciones se usan para almacenar, recuperar, manipular, y comunicar datos agregados
- Representan elementos de datos que forman un grupo de manera natural
  - Una mano en el mus (una colección de cartas)
  - Una carpeta de correo (una colección de e-mails)
  - Un directorio telefónico (un diccionario que mapea nombres a números de teléfono)

La Java Collection Framework

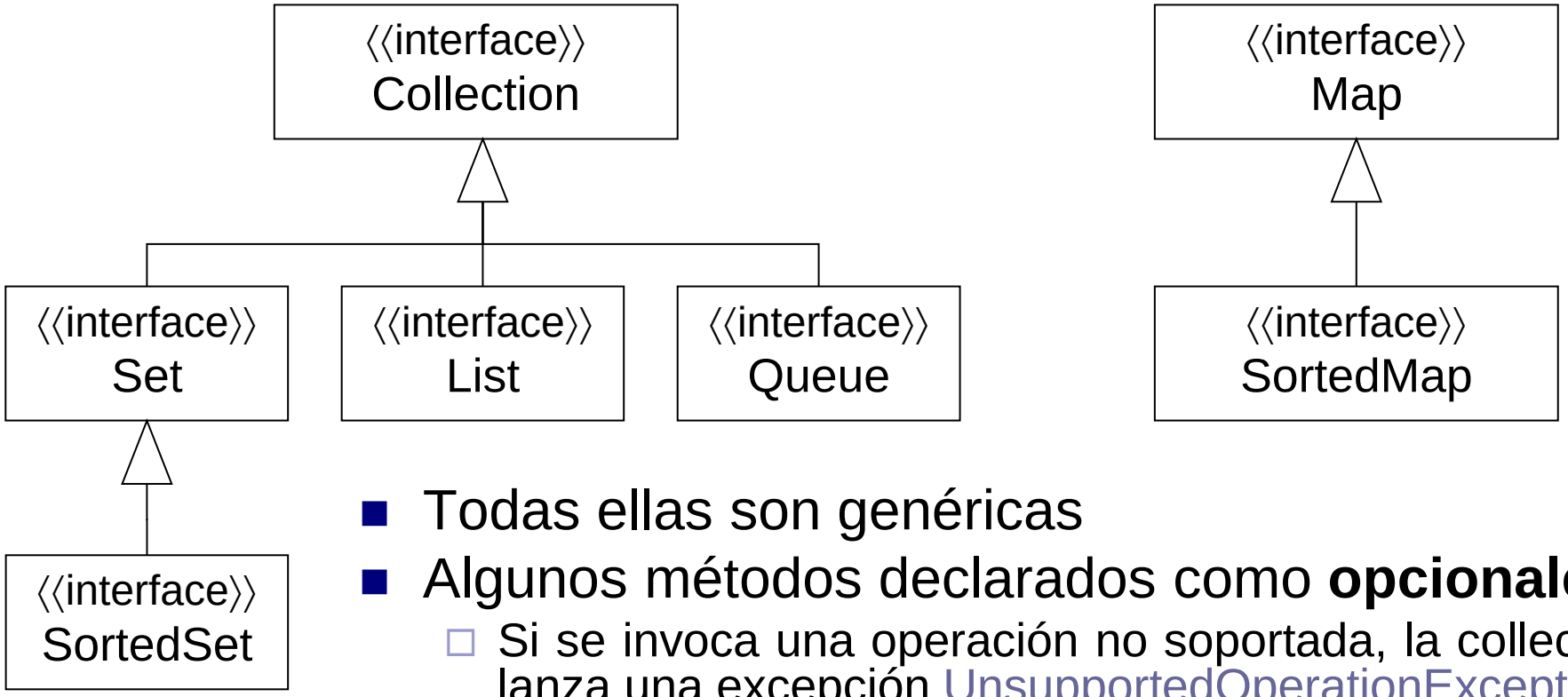
- Una arquitectura unificada para representar y manipular colecciones
- Contiene
  - **Interfaces**. Permiten manipular las colecciones independientemente de su implementación
  - **Implementaciones**. Las implementaciones concretas de las interfaces
  - **Algoritmos**. Métodos que realizan computaciones útiles, como búsqueda y ordenación, sobre objetos que implementan las interfaces de la colección
- Otros frameworks similares: La STL sobre C++

Ventajas del uso de colecciones

- Al seguir patrones comunes reduce el tiempo de aprendizaje y **aumenta la productividad** de la codificación mediante la reutilización
- **Eficiencia**: Las implementaciones están optimizadas para los usos típicos de las colecciones
- Interoperabilidad entre librerías.
  - Las librerías usan las mismas interfaces de colecciones, lo cual facilita su integración

Interfaces de las colecciones

Las interfaces principales

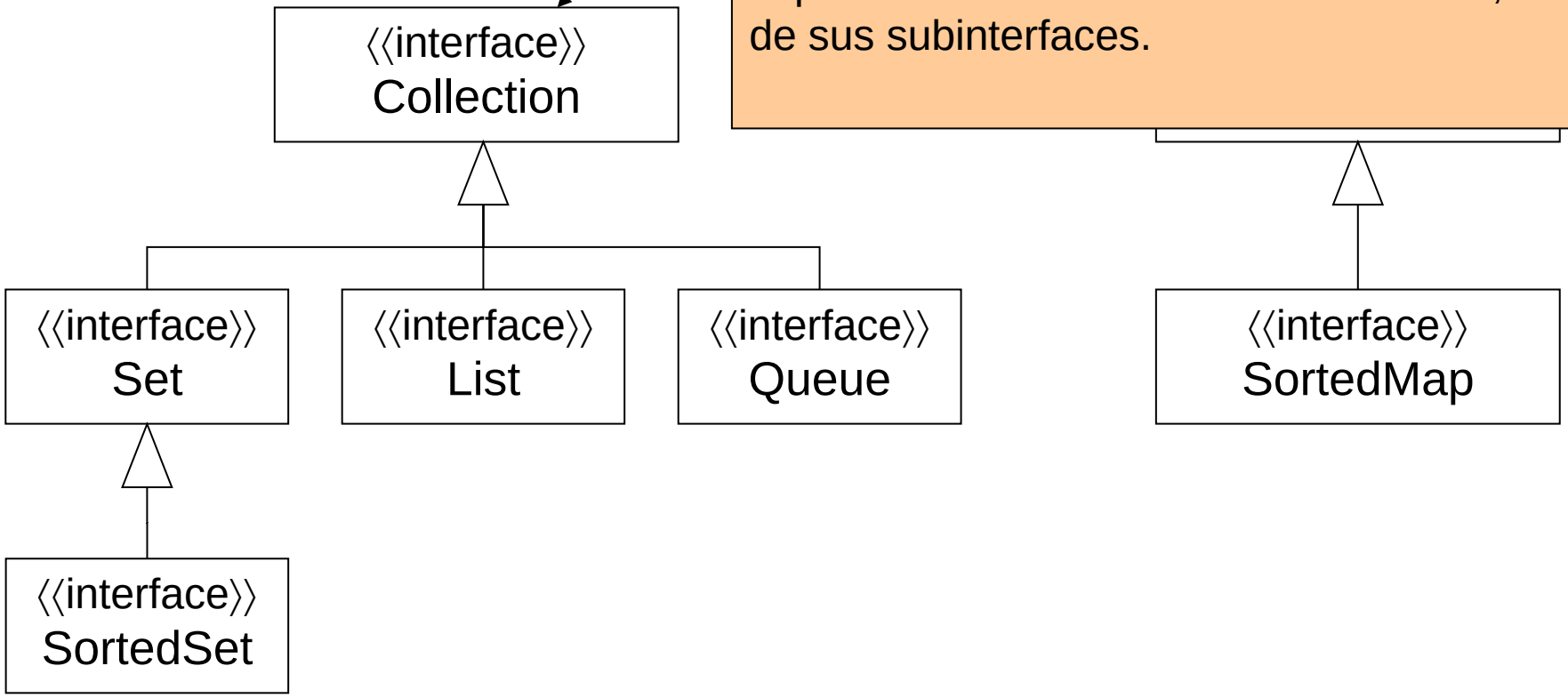


- Todas ellas son genéricas
- Algunos métodos declarados como **opcionales**
  - Si se invoca una operación no soportada, la colleción lanza una excepción UnsupportedOperationException

Interfaces de las

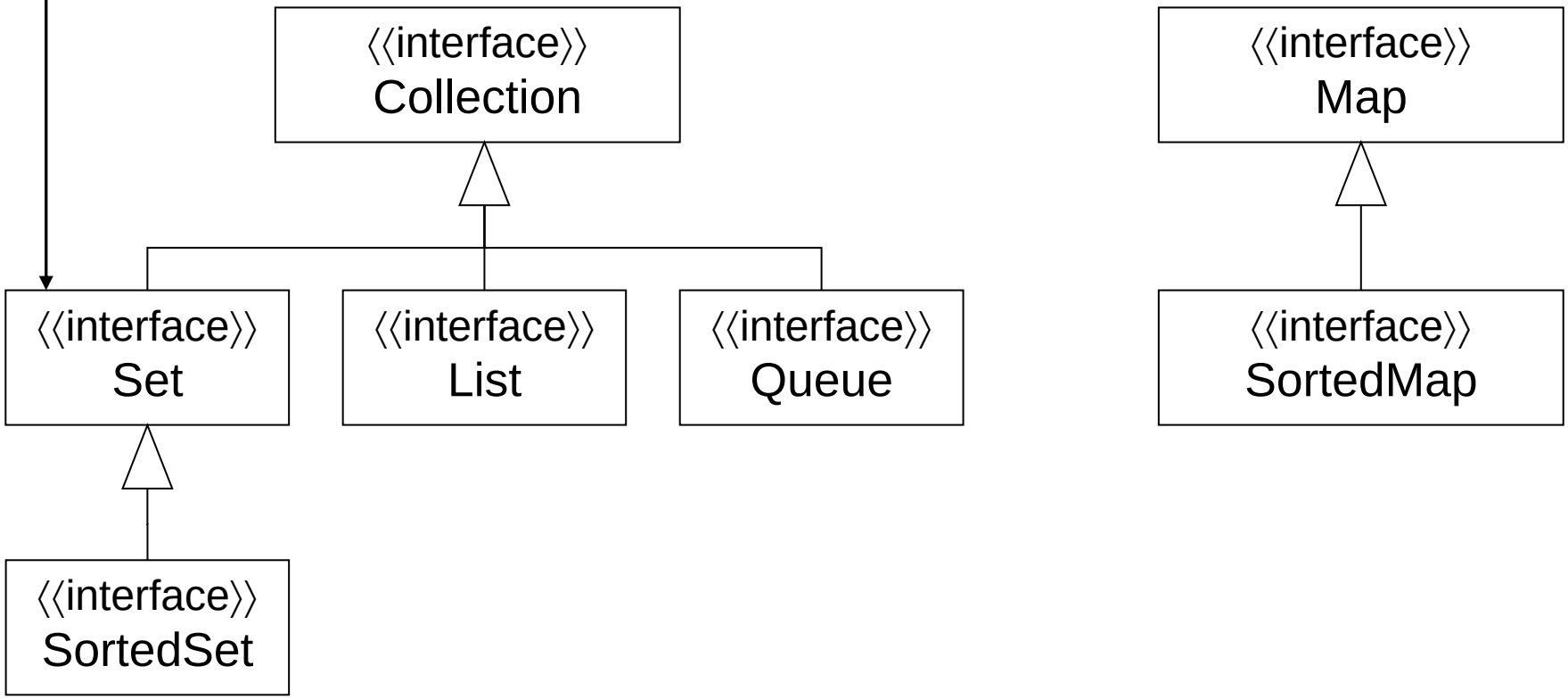
Las interfaces principales

- La raiz de la jerarquía
- Representa el mínimo común denominador que todas las colecciones implementan
- Útil para intercambiar colecciones cuando se busca la máxima generalidad
- Las colecciones de Java no dan una implementación directa de esta interfaz, sino de sus subinterfaces.



Interfaces de las colecciones

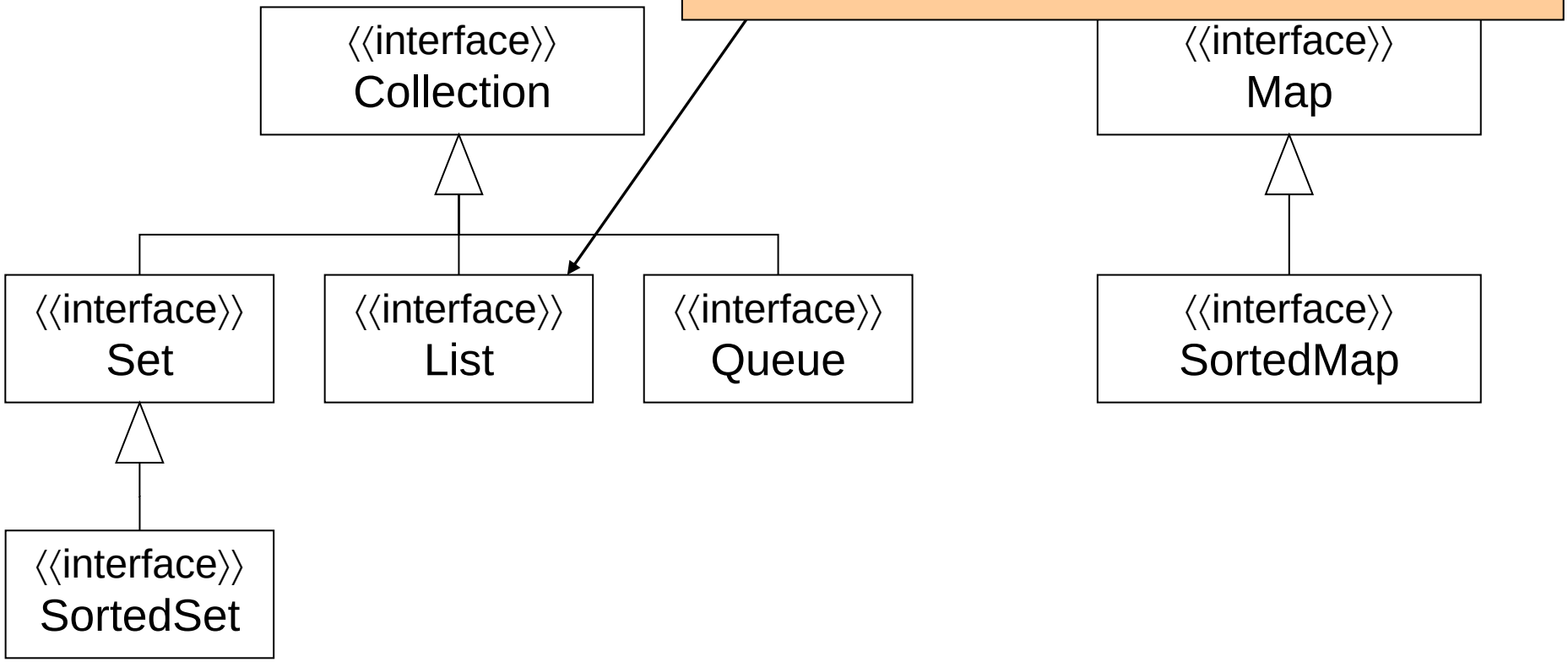
- Una colección que no admite duplicados
- Modela la abstracción matemática de “conjunto”



Interfaces de las colecciones

Las interfaces principales

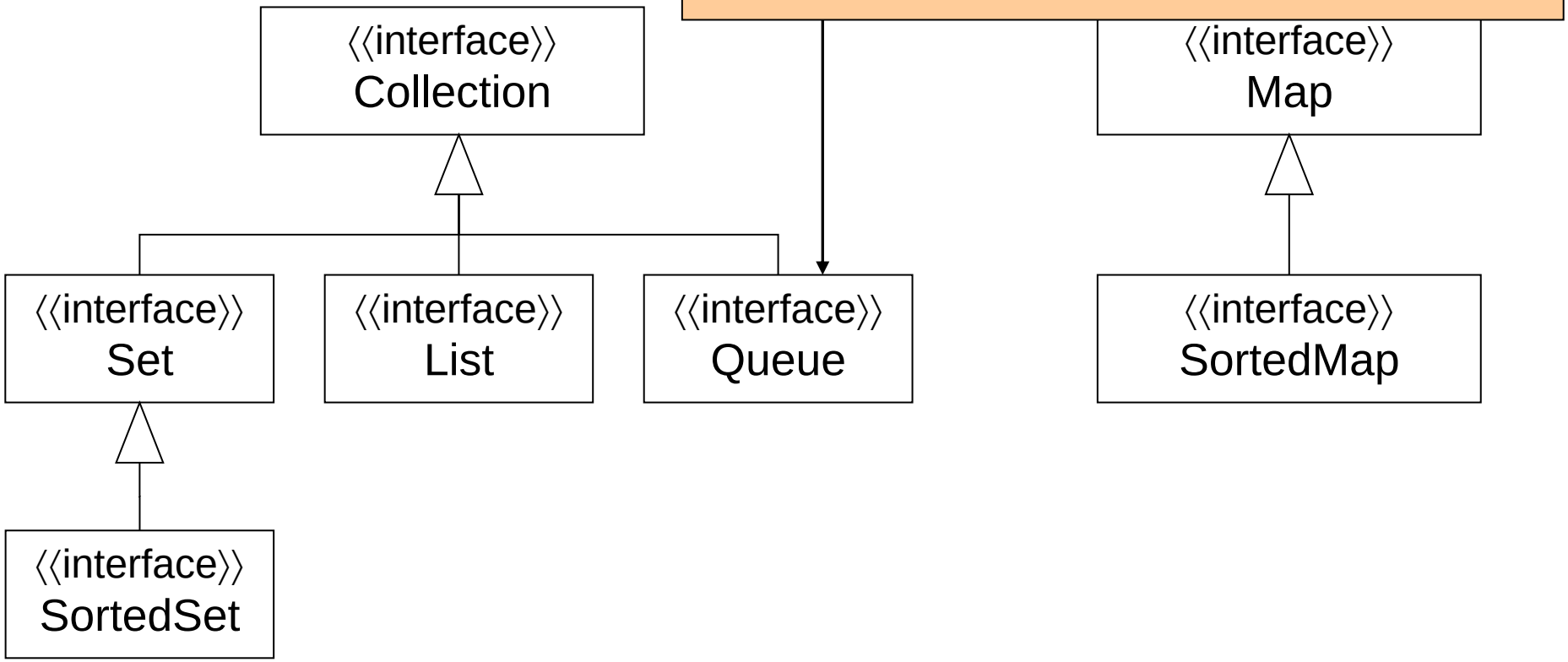
- Una colección ordenada, también llamada “secuencia”
- Pueden contener elementos duplicados



Interfaces de las colecciones

Las interfaces principales

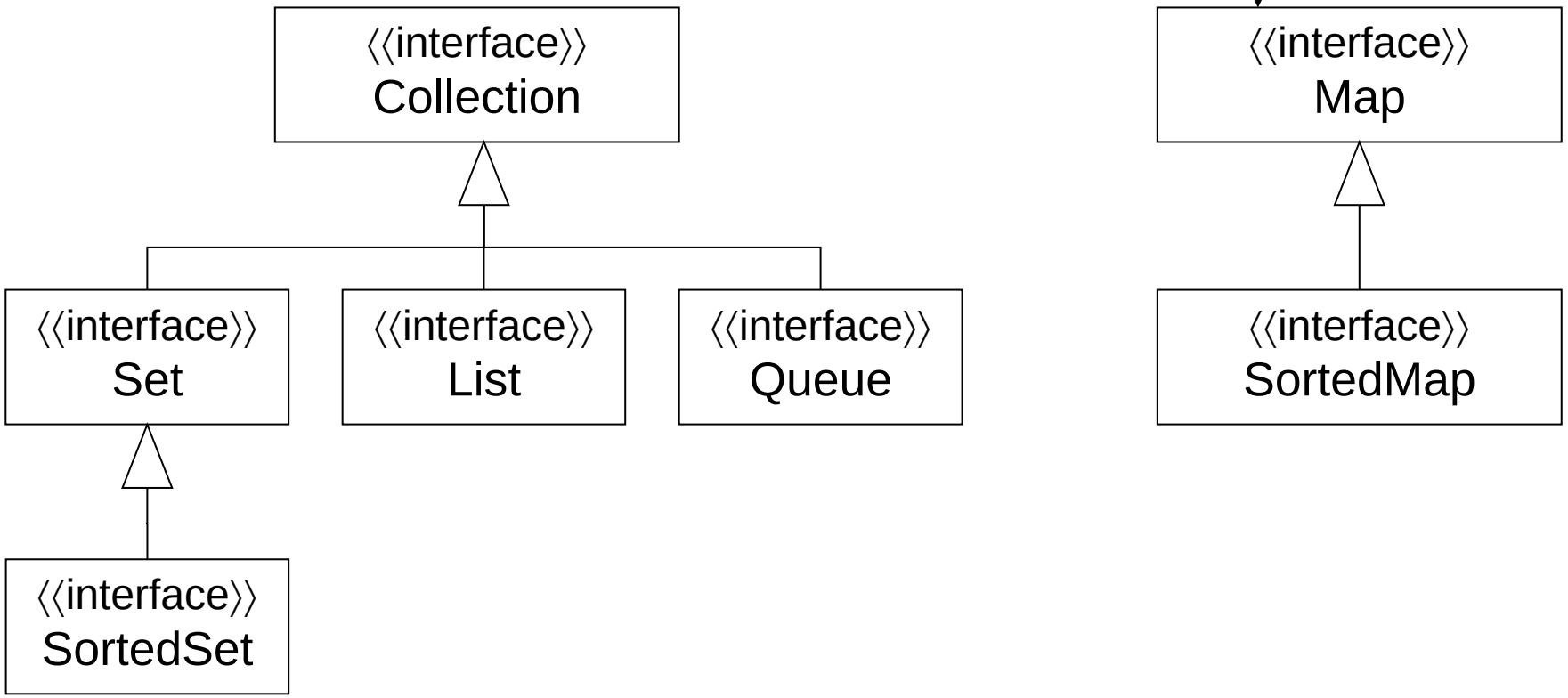
- Una cola, típicamente ordena elementos de Manera FIFO
- Colas con prioridad



Interfaces de l

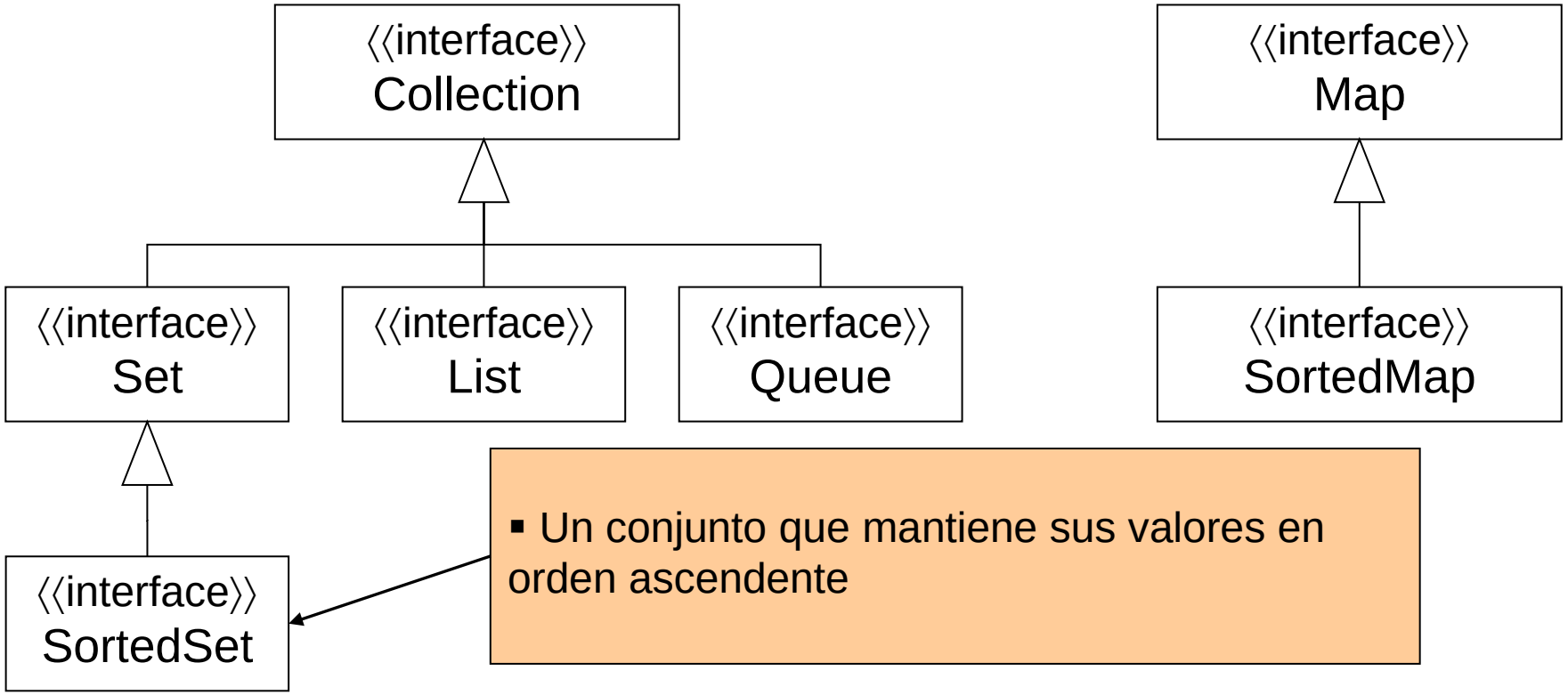
Las interfaces principales

- Un diccionario de pares (clave, valor)
- No puede contener claves duplicadas
- Concepto matemático de *función*



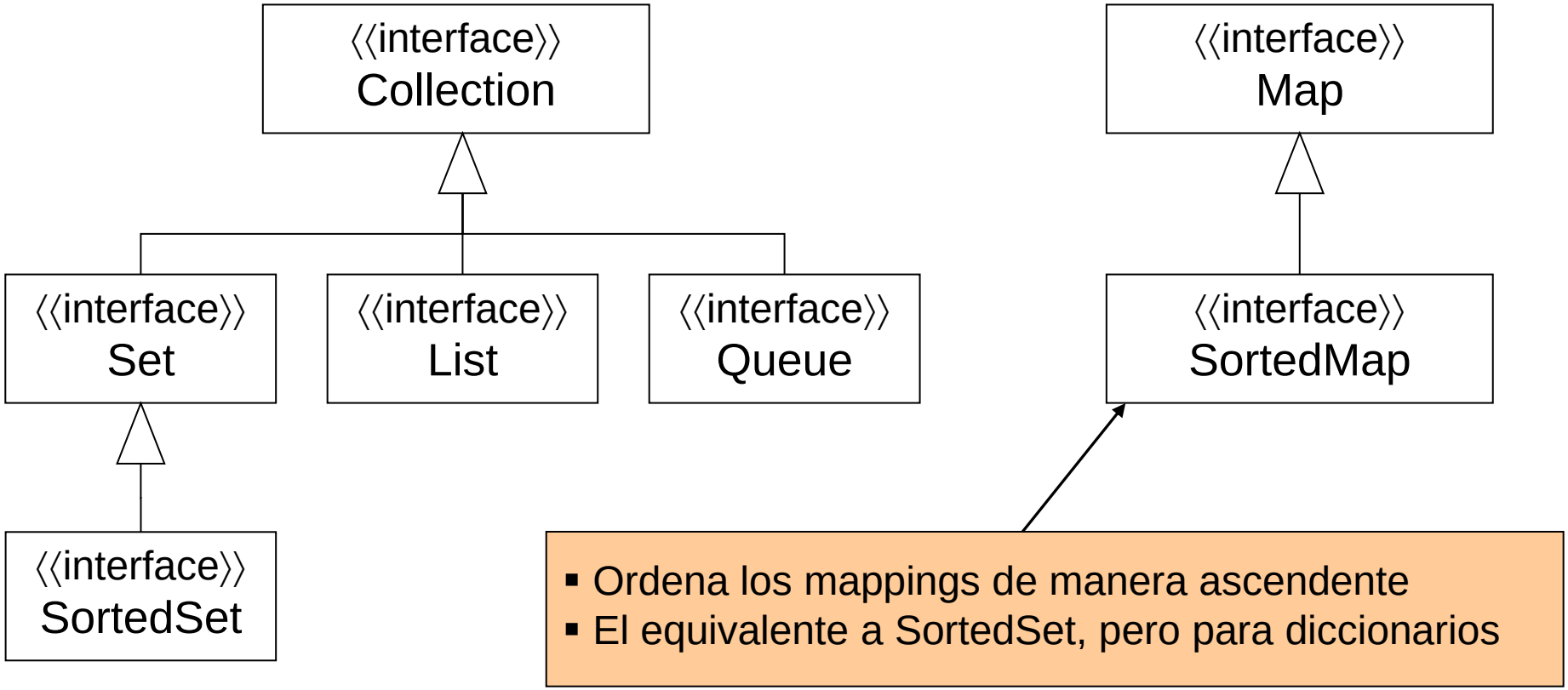
Interfaces de las colecciones

Las interfaces principales



Interfaces de las colecciones

Las interfaces principales



La interfaz `Collection`

- Útil para el intercambio de colecciones con máxima generalidad
- Por ejemplo, todas las implementaciones tienen un constructor que toma como argumento un objeto de tipo `Collection`, para inicializar su contenido
- Facilita la conversión entre colecciones:

```
Collection<String> c = new HashSet<String>();
c.add("Uno");
c.add("Dos");
c.add("Tres);
// initializing a list out of the set
List<String> list = new ArrayList<String>(c);
```

La interfaz Collection

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);                          //optional
    boolean remove(Object element);                  //optional
    Iterator<E> iterator();
    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);              //optional
    boolean retainAll(Collection<?> c);              //optional
    void clear();                                    //optional
    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Cómo recorrer colecciones

- Con un **for mejorado**:

```
for (Object o : collection)
    System.out.println(o);
```

- ¡Ojo! No se permite borrar elementos dentro del bloque del for (se lanza una `ConcurrentModificationException`)
- Con un **iterador**. Un iterador es un objeto que permite recorrer colecciones (incluso varias en paralelo) y borrar elementos

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next())) it.remove();
}
```

La interfaz `Set`

- Una colección que no admite duplicados
- Contiene sólo métodos heredados de `Collection` y añade la restricción de no duplicados
- Tres implementaciones
  - `HashSet:` Guarda los elementos en una hash
  - `TreeSet:` Guarda los elementos en un árbol rojo-negro. Elementos ordenados, necesitan implementar `Comparable`
  - `LinkedHashSet:` Guarda los elementos en una hash encadenada

La interfaz Set

Ejemplos

- Eliminar los repetidos de una colección *c*

```
Collection<Type> noDups = new HashSet<Type>(c);
```

- Imprime las palabras repetidas

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicado: " + a);
        System.out.println(s.size() + " palabras diferentes:”
                                    + s);
    }
}
```

La interfaz `List`

- Es una colección ordenada (una secuencia) y puede contener duplicados
- Además de los métodos de *Collection*, añade
  - **Acceso posicional** — obtener elementos dada su posición numérica en la lista
  - **Búsqueda** — busca un elemento específico en la lista y devuelve su posición
  - **Iteración** — extiende la semántica del `Iterator` para aprovechar la naturaleza secuencial de la lista
  - **Rango** — permite operaciones de rango sobre la lista
- Tres implementaciones: `ArrayList`, `LinkedList`, y `Vector`

La interfaz `List`

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);                          //optional
    boolean add(E element);                               //optional
    void add(int index, E element);                       //optional
    E remove(int index);                                  //optional
    boolean addAll(int index, Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

Iteradores de lista

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

- Iterando desde el final al principio

```
for (ListIterator<Type> it = list.listIterator(list.size());
     it.hasPrevious(); )
{
    Type t = it.previous();
    ...
}
```

LinkedList vs. ArrayList

LinkedList<E>

- get(int index) is O(n)
- add(E element) is O(1)
- add(int index, E element) is O(n)
- remove(int index) is O(n)
- Iterator.remove() is O(1) ← **beneficio principal de LinkedList**
- ListIterator.add(E element) is O(1) ← **beneficio principal LinkedList**

ArrayList<E>

- get(int index) is O(1) ←**beneficio principal ArrayList**
- add(E element) es O(1) amortizado, pero O(n) en el caso peor, porque el array debe redimensionarse y copiarse
- add(int index, E element) is O(n - index) amortizado, pero O(n) en el caso peor
- remove(int index) is O(n - index) (borrar el último es O(1))
- Iterator.remove() is O(n - index)
- ListIterator.add(E element) is O(n - index)

Ejercicio

- Dadas dos listas de enteros, devuelve una colección con los elementos comunes, sin repetición

[inténtalo tú mismo y luego mira la explicación paso a paso en este video: **https://www.youtube.com/watch?v=BWtK8_E3uQk**]

La interfaz Queue

```
public interface Queue<E> extends Collection<E> {
   E element();
   boolean offer(E e);
   E peek();
   E poll();
   E remove();
}
```

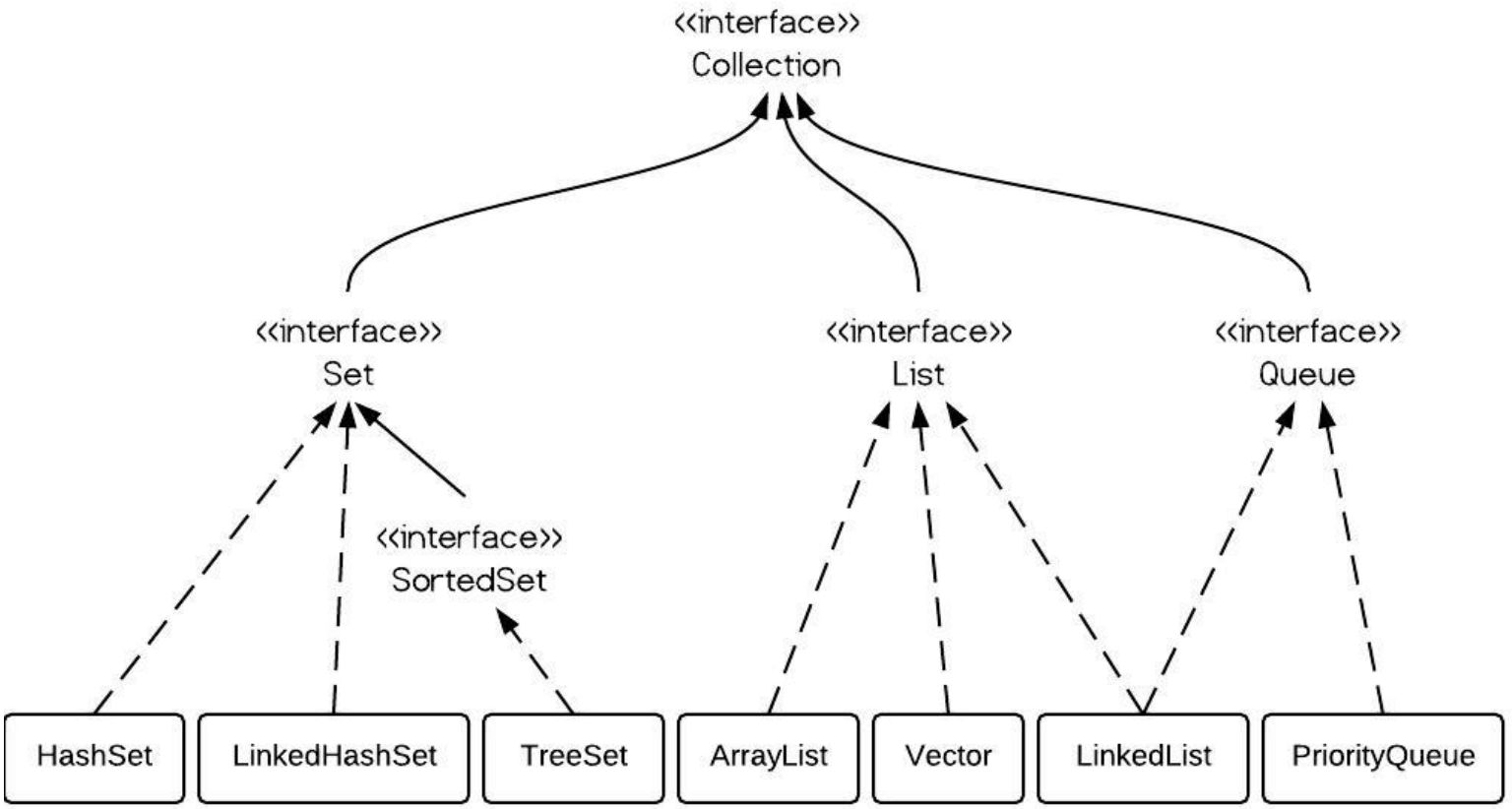
	Lanza una excepción	Devuelve un valor especial (null/false)
Insertar	`add(e)`	`offer(e)`
Eliminar	`remove()`	`poll()`
Examinar	`element()`	`peek()`

La interfaz Queue

```
import java.util.*;

public class Countdown {
    public static void main(String[] args) throws InterruptedException{
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--)
            queue.add(i);
        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

Implementaciones de algunas colecciones



http://www.programcreek.com/

La interfaz `Map`

- Un objeto que asigna valores a claves
- No puede contener claves duplicadas
- Cada clave puede tener asignados como máximo un valor
- Modela el concepto matemático de *función*
- Tres implementaciones: `HashMap`, `TreeMap`, y `LinkedHashMap`

La interfaz Map

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

La interfaz Map

Ejemplo

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

La interfaz Map

Ejemplo

- Iterar sobre todas las claves:

```
for (KeyType key : m.keySet())
  System.out.println(key);
```

- Iterar sobre todos los pares clave, valor:

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

Ordenación

- Una colección c se puede ordenar así:

```
Collections.sort(c);
```

- Ordenación de acuerdo al orden natural: interfaz `Comparable` (método `compareTo`)

- Otra posibilidad: interface `Comparator<T>` y el método `compare(T, T)`

Interface SortedSet

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    // Endpoints
    E first();
    E last();
    // Comparator access
    Comparator<? super E> comparator();
}
```

- Mantiene sus elementos en order ascendente, de acuerdo al ***orden natural*** de sus claves, o de acuerdo al `Comparator` proporcionado (opcionalmente) en el constructor de `SortedSet`

Interface *SortedSet*

Ejemplos

- Palabras entre “doorbell” y “pickle”, excluyendo esta última

```
int count = dictionary.subSet("doorbell", "pickle").size();
```

- Borrar todos los elementos que empiecen por f

```
dictionary.subSet("f", "g").clear();
```

Interface SortedMap

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

- Mantiene sus entradas en orden ascendente, ordenadas de acuerdo al orden natural de las claves, o a un `Comparator` que se da cuando se crea el `SortedMap`

Opciones de las colecciones

- No hay tipos para las variantes (ej.: tamaño fijo, sólo lectura, o inmutable)
- Java no permite declarar objetos constantes, como en C++ (const)
- Solución: Algunos métodos de las interfaces son opcionales
  - Un método opcional es un método con una implementación por defecto que que lanza `UnsupportedOperationException`
  - Se ha de documentar qué métodos no están soportados

Ejercicio

- Crear una clase que mantenga el valor bursátil de distintas compañías
- Existen objetos observador pueden registrarse para ser notificados de cambios en los valores
- Crea un diseño general, que permta distintos tipos de objetos observador:
  - Uno que imprima por consola los cambios en los valores
  - Otro que imprima en un fichero

- Modifica el programa para que los observadores se puedan registrar para recibir cambios en compañías específicas, y no todos los cambios.

Nota: Usaremos el patrón de diseño Observer:
https://en.wikipedia.org/wiki/Observer_pattern

Indice

- Introducción
- Interfaces
- **Comparando objetos**
- Implementaciones
- Algoritmos

- Genericidad

equals()

- Es un método de la clase `Object`,
  - La manera estándar de comparer objetos, en vez de '=='
- Cualquier clase puede sobreescribirlo, para definir una noción específica de igualdad
- Sobreescribir *equals* facilita la búsqueda en arrays, colecciones y mapas
  - `<list>.contains(obj)`
  - La búsqueda implica especificar una clave de búsqueda, que se comparará con los objetos de la colección mediante `equals`
- Si cada objeto es considerado único, no hace falta sobreescribir (equals por defecto implementa igualdad referencial, con '==')

equals()

- Una implementación de `equals` debe satisfacer las siguientes propiedades:
  - **Reflexiva**: `this.equals(this)` es true
  - **Simétrica**: si x e y son dos referencias, `x.equals(y)` es true si y solo si `y.equals(x)` es true
  - **Transitiva**: sean tres referencias x, y, z, si `x.equals(y)` es true, `y.equals(z)` es true entonces `x.equals(z)` es true
  - **Consistente**: `x.equals(y)` debe dar lo mismo en sucesivas invocaciones si x e y no han sufrido modificaciones que modifiquen la comparación
  - **Comparación a null**: `obj.equals(null)` debe dar false si obj es distinto de null

Ejemplo

```
public class VersionNumber {
   private int release;
   private int revision;
   private int patch;

   public VersionNumber(int release, int revision, int patch) {
      this.release = release;
      this.revision = revision;
      this.patch = patch;
   }
   @Override
   public String toString() {
      return "("+release+", "+revision+", "+patch+")";
   }
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return vn.patch == this.patch &&
             vn.revision == this.revision &&
             vn.release == this.release;
   }
}
```

hashCode()

- Hashing: técnica eficiente para almacenar y recuperar datos (e.j., interfaz Map)
- Necesita calcular un índice (código hash) a partir de un elemento
- Este cálculo lo realiza una función hash
- Si hay colisiones (dos elementos distintos que tienen el mismo valor hash) se guardan en una lista encadenada

Contrato del método hashCode

- **Consistencia**: varias invocaciones a `hashCode` deben devolver el mismo valor si el objeto no se modifica de tal manera que haga cambiar el valor que devuelve `equals`
- **Igualdad** con `equals()` implica mismo valor de `hashCode()`
- **Desigualdad** con `equals()` no implica necesariamente distinto valor de `hashCode()`

Ejemplo

```
public class VersionNumber {
   private int release;
   private int revision;
   private int patch;

   //…
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return vn.patch == this.patch &&
             vn.revision == this.revision &&
             vn.release == this.release;
   }
   @Override
   public int hashCode() {
      int hashValue = 11;
      hashValue = 31*hashValue+release;
      hashValue = 31*hashValue+revision;
      hashValue = 31*hashValue+patch;
      return hashValue;
   }
}
```

Consistencia: Ambos métodos usan patch, revision y release para el cálculo

La interfaz `Comparable<E>`

- El orden natural de los objetos de una clase se especifica implementando la interfaz `Comparable<E>`
  - Método `compareTo(E o)`
  - Muchas clases estándar lo implementan: `String`, `Date`, `File`
  - Podemos usar objetos Comparables en: conjuntos ordenados, claves en un mapa ordenado, elementos en colecciones que se ordenen manualmente mediante `Collections.sort()`
- También se puede especificar un orden total implementando un comparador, que implemente la interfaz `Comparator`

Ejemplo

```
public class VersionNumber implements Comparable<VersionNumber>{
   private int release;
   private int revision;
   private int patch;

   //...
   @Override
   public boolean compareTo(VersionNumber vno) {
      int releaseDiff = release-vno.release;
      if (releaseDiff!=0) return releaseDiff;
      int revisionDiff = revision-vno.revision;
      if (revisionDiff!=0) return revisionDiff;
      int patchDiff = patch-vno.patch;
      if (patchDiff!=0) return patchDiff;
      return 0;
   }
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return vn.patch == this.patch &&
             vn.revision == this.revision &&
             vn.release == this.release;
   }
}
```

Ejemplo

```
public class VersionNumber implements Comparable<VersionNumber>{
   private int release;
   private int revision;
   private int patch;

   //...
   @Override
   public int compareTo(VersionNumber vno) {
      int releaseDiff = release-vno.release;
      if (releaseDiff!=0) return releaseDiff;
      int revisionDiff = revision-vno.revision;
      if (revisionDiff!=0) return revisionDiff;
      int patchDiff = patch-vno.patch;
      if (patchDiff!=0) return patchDiff;
      return 0;
   }
   @Override
   public boolean equals(Object obj) {
      if (obj==this) return true;
      if (!(obj instanceof VersionNumber)) return false;
      VersionNumber vn = (VersionNumber)obj;
      return this.compareTo(vn)==0;
   } // Simplifcamos el método equals reutilizando compareTo
}
```

Ejercicio

- Crea una clase Zapato, con un número, modelo y color
- Añade métodos para comparar por modelo alfabéticamente, después por color alfabéticamente, y por último por número en orden creciente

Indice

- Introducción
- Interfaces
- Comparando objetos
- **Implementaciones**
- Algoritmos

- Genericidad

Implementaciones

Interfaces	Implementaciones				
	Tabla Hash	Array Redimens.	Arbol	Lista Enlazada	Tabla Hash+ Lista enlazada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList PriorityQueue	
Map	HashMap		TreeMap		LinkedHashMap

Implementaciones de `Set`

- `HashSet, TreeSet y LinkedHashSe`
  - HashSet: mantiene elementos “ordenados” por hash
  - TreeSet: ordena por orden natural (requiere elementos comparables)
  - LinkedHashSet: orden de inserción
- Podemos inicializarlo a un tamaño:

```
Set<String> s = new HashSet<String>(64);
```

- El conjunto se redimensiona como sea necesario
- Dos implementaciones de propósito especial:
  - EnumSet: para tipos enumerados (representación interna mediante un array de bits)
  - CopyOnWriteArraySet.

Implementaciones de `List`

- `ArrayList, LinkedList, Vector`
  - ArrayList: usa internamente un array para almacenar los elementos
  - LinkedList: usa una lista doblemente enlazada
  - Vector: una clase más antígua similar a ArrayList
- Implementaciones de propósito especial:
  - `CopyOnWriteArrayList`.

Implementaciones de `Map`

- HashMap, TreeMap, LinkedHashMap
  - HashMap: Almacena las claves usando hash
  - TreeMap: mantiene las claves en orden usando orden natural (requiere claves comparables)
  - LinkedHashMap: almacena claves usando orden de inserción
- Implementaciones de propósito especial:
  - EnumMap: para claves de tipo enumerado.
  - WeakHashMap: las claves se pueden eliminar (mediante recogida de basura) cuando ya no se referencian.
  - IdentityHashMap.
- Otras implementaciones concurrentes.

Indice

- Introducción
- Interfaces
- Comparando objetos
- Implementaciones
- **Algoritmos**

- Genericidad

Conversiones Array-Collection

- Convertir en Array de Object, sin genérico

```
Collection c;
Object[] r= c.toArray();
```

- Convirtiendo a un tipo concreto

```
String[] s= c.toArray(new String[0]);
```

  - Se usa el array vacío para que `toArray` cree uno del mismo tipo con la longitud apropiada
  - Generará una excepción si los tipos en tiempo de ejecución no coinciden o no son subtipos del tipo del array

Conversiones Array-Collection

- Creación de una lista

```
T a, b, c;
List<T> r=Arrays.asList(a, b, c);
```

- Convirtiendo array en lista

```
T[] a;
List<T> r= Arrays.asList(a);
```

- En ambos casos se utiliza el mismo método `asList`
  - El método `asList` admite una secuencia de argumentos de longitud variable, `T…a`, que se convierte en un Array automáticamente
  - Si se modifica la lista se modifica el array, pero no es posible cambiar el tamaño de la lista, ya que la lista usa el array para almacenar los elementos, siendo solo una vista

Algoritmos de propósito general

- Las clases `Arrays` y `Collections` contienen métodos estáticos con diferentes algoritmos
- Ordenación
  - `Collections.sort(List l)`. Ordena usando merge sort, y el orden natural (interfaz `Comparable`)
  - `Collections.sort(List l, Comparator c)`. Ordena usando un comparador
- Desordenación
  - `Collections.shuffle(List l)`. Permuta aleatoriamente la lista

Métodos de manipulación

- Collections.
  - `reverse(List l)`: Invierte la lista
  - `fill(List<T>, T e)`: reemplaza todos los elementos por el elemento e.
  - `copy(List<T> dest, List<T> src)`: Copia de la lista «src» a la «dest»
  - `swap(List<T> l, int i, int j)`: Intercambia dos posiciones
  - `addAll(Collection<T> c, T… elementos)`: Añade elementos a la colección c

Algoritmos de búsqueda

- Collections.
  - `binarySearch(List<T> l, T e)`. Han de estar ordenados de forma ascendente y T implementar comparable<T>
    - También admite un comparador como tercer elemento, si no se quiere usar el orden natural
  - `frequency(Collection c, Object o)`: Cuanta las veces que aparece un elemento o en c
  - `disjoint(Collection a, Collection b)`: Indica si son disjuntos
  - `min, max`: Búsqueda del mínimo y máximo

Otros métodos de Collections

- `emptyList, emptyMap, emptySet`: Devuelven colecciones vacías inmutables.
- `singletonList, singletonMap`: Devuelve un elemento o un par clave-valor como un lista o un mapa. Ambos son inmutables.
- `List<T> nCopies(int n, T e)`: Devuelve una lista inmutable con n copias de «e». Internamente solo almacena una

Otros métodos de Collections

- `public static <T> List<T> unmodifiableList(List<? extends T> list)`
  - Crea un envoltorio inmutable de la lista. No es una copia de la lista, si no que delega en ella.
  - Es útil para métodos que han de devolver List<A> a partir de un atributo de tipo List<B>, ya que aunque B sea subclase de A, List<B> no es subclase de List<A>
  - Al ser inmutable no se puede añadir un elemento de tipo C, aunque sea subclase de A
  - También existe el método equivalente para Map y Set

Ejercicio

- Airport (examen final 2014)

Indice

- Introducción
- Interfaces
- Comparando objetos
- Implementaciones
- Algoritmos

- **Genericidad (tipos genéricos)**

Tipos genéricos

- Un mecanismo que permite crear tipos que tienen otros tipos como parámetros
- Permite crear una implementación parametrizable con diferentes tipos. Por ejemplo:
  - `Map<String, List<Integer>> m`: m es una mapa donde la clave es una cadena y el valor una lista de enteros
- Sin genéricos:
  - `Map m`: m es una mapa donde la clave es un objeto y el valor otro objeto.

Cómo funcionan

- Al definir la clase o el método se define el tipo genérico entre los símbolos < y >

```
public interface List<E>{
    void add (E e);
    Iterator<E> iterator();
}
```

- El compilador es el encargado de comprobar los tipos
- Realmente sólo se crea una clase, en tiempo de ejecución la información de los tipos genéricos se borra
  - Esta limitación impide usar tipos básicos (int, double, …) como parámetros de tipos genéricos
- En C++ sí se genera una clase para cada combinación de clases y tipos, lo cual permite usar tipos básicos como parámetros

Ejemplo

```
public class Pair<A, B> {
    private A first;
    private B second;
    public Pair(A first, B second) {
        this.first=first;   this.second=second;
    }
    public int hashCode(){
        return first.hashCode()+second.hashCode();
    }
    public boolean equals(Object obj){
        if (obj instanceof Pair){
            Pair o=(Pair) obj;
            return first.equals(o.first) &&
            second.equals(o.second);
        }
        else return super.equals(obj);
    }
    public String toString(){
        return "{"+first+", "+second+"}";
    }
    public A getFirst() {
        return first;
    }

    public void setFirst(A first) {
        this.first = first;
    }

    public B getSecond() {
        return second;
    }

    public void setSecond(B second) {
        this.second = second;
    }
}
```

Limitatación: Los tipos A y B no pueden insanciarse

Requisitos para los tipos paramétricos

- En una clase genérica, podemos exigir ciertos requitos a sus parámetros:

```
public class ReqComparable<A extends Comparable<A> & Serializable>
{
  private List<A> elems = new ArrayList<A>();

  public ReqComparable(A ...params) {
     elems.addAll(Arrays.asList(params));
     Collections.sort(elems);
  }

  @Override public String toString() {
    return this.elems.toString();
  }
}
```

Herencia y genericidad

- Cuando creamos una subclase de una clase genérica, podemos:
  - Instanciar los tipos paramétricos:

```
public class ListStrings extends ArrayList<String>{
  …
} // ListStrings no es un tipo genérico
```

  - Dejarlos abiertos:

```
public class MyList<T> extends ArrayList<T>{
  …
} // MyList<T> es genérica, y debemos instanciar sus
  // parámetros cuando creemos objetos
```

Herencia y genericidad

- ¿Qué ocurre en el siguiente ejemplo?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

- ¿Existe algún error?
  - La clase `List<String>` no «hereda» de `List<Object>`
- ¿Por qué?

```
lo.add(new Object());
String s = ls.get(0);
```

  - Si heredase de List<Object> se podrían añadir objetos que no son del tipo adecuado.
  - Los método podrían usarse con parámetros contra-variantes (Object en vez de String)

Herencia y genericidad

- Usando Object como parámetro

```
public void imprime(List<Object> l){
   //código de imprime
}
```

- ¿Podemos usar imprime con cualquier tipo de lista?
  - No, es más restrictivo que usar imprime(List l), ya que no hay herencia de List<Object> a listas de otros tipos

- Solución: Añadir flexibilidad a los parámetros usando el comodín ?

```
public void imprime(List<?> l){
   //código de imprime
}
```

El comodín ?

- Dentro del método imprime podemos obtener los elementos con get

```
List<?> l;
Object o= l.get(0);
```

  - Funciona, ya que aunque «get» devuelve un objeto de tipo desconocido «?», todos los tipos heredan de «Object»
- ¿Podemos usar add?

```
Object o;
l.add(o);
```

- La lista es de tipo desconocido, por lo tanto no podemos usar add con una instancia de Object, ya que Object no hereda de «?». No podremos llamar a ningún método que reciba por argumento un tipo desconocido

Comodín: permitiendo la herencia

```
public void añadeVehículos(List<? extends Vehículo> l){
  //añadimos al atributo List<Vehículo > v;
  v.addAll(l);
}
```

- Ahora sí podemos hacer

```
List<Coche> c;
o.añadeVehículos(c);
//funciona si Coche hereda de Vehículo
```

Comodín super

- `? super T` denota un tipo desconocido que es un supertipo de T (o T).
- Por ejemplo, el siguiente método funciona:

```
public void anyade(List<? super Number> lista) {
  lista.add(8);
  lista.add(new Float(8));
}
```

- Pero este no, ¿por qué?

```
public void anyade(List<? extends Number> lista) {
  lista.add(8);
  lista.add(new Float(8));
}
```

Comodín super

```
public class ComparablePair<A extends Comparable<? super A>, B extends Comparable<?
super B>> extends Pair<A, B> implements
    Comparable<ComparablePair<A, B>>{

    /** Creates a new instance of ComparablePair */
    public ComparablePair(A a, B b) {
        super (a, b);
    }

    public int compareTo(ComparablePair<A, B> o){
        int r1=getFirst().compareTo(o.getFirst());
        if (r1!=0) return r1;
        else return getSecond().compareTo(o.getSecond());
    }
}
```

Ejemplo

```
public interface Transform <A, B> {
//convert from A to B using a function
    B transform (A element);
}
…
public class StringTransform implements Transform<Object, String> {
    //Creamos transformer como singleton
    public final static StringTransform transformer= new StringTransform();

    public String transform(Object element){
      return element.toString();
    }
}
```

Ejemplo

```
public class TransformList<A, B> extends AbstractList<B> implements List<B>{
    Transform<? super A,? extends B> transformer;
    List<? extends A> l;
    /** Constructor que crea una vista transformada de la lista, mediante un transformer*/
    public TransformList(List<? extends A> l, Transform<? super A,? extends B> transformer) {
        this.l=l;
        this.transformer=transformer;
    }
    //Delegamos los métodos get y size, los únicos necesarios para List usando AbstractList
    public B get(int pos){
        return transformer.transform(l.get(pos));
    }
    public int size(){
        return l.size();
    }
}
```

Genericidad de métodos

- Los métodos también pueden ser genéricos

```
public static <T> void copyToArray(T[] array,
                                   List<? extends T> lista)
{
  int i = 0;
  for (T t : lista)
    array[i++] = t;
}

…
List<String> ls = new ArrayList<String>();
ls.add("elemento");
ls.add("elemento2");

Object[] array = new Object[2];
copyToArray(array, ls);
```

Genericidad de métodos

- Métodos con más de un tipo genérico

```
public static <T, S extends T>
   boolean copiaArray(T[] dest, S[] src)
   {
      if (dest.length < src.length) return false;
      for (int i= 0; i<src.length; i++)
         dest[i] = src[i];
      return true;
   }
…
String src[] = {"dos", "cadenas"};
Object dest[] = new Object[2];
copiaArray(dest, src);
```

Genericidad de métodos

- Ligadura explícita del tipo en la llamada:

```
public class Util {
  public static <R> R as(List<? super R> lista, int pos) {
      return (R)lista.get(pos); // unchecked!!
  }
}
...
List<Object> lista2 = new ArrayList<Object>();
lista2.add(3);

int n = 2 + Util.<Integer>as(lista2, 0);

System.out.println(n);
```