


Colecciones

Contenido

- Clases e Interfaces genéricas 
- Interfaces para objetos ordenables
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias

Clases e Interfaces genéricas

- Las clases e interfaces genéricas permiten en una única definición expresar comportamientos comunes para objetos pertenecientes a distintas clases.
- El ejemplo más habitual es una clase contenedora (una colección): lista, conjunto, etc. De hecho, desde el Tema 2, ya hemos usado la clase genérica **ArrayList<T>** (ArrayList<Integer>, ArrayList<Punto>, ArrayList<Estudiante>, ...)
- Desde la versión JDK1.5.0, Java dispone de mecanismos para manejar clases e interfaces genéricas mediante el uso de parámetros.
 - Una clase o interfaz puede incorporar parámetros en su definición.
 - A la hora de usar una clase o interfaz genérica, se especifican los valores concretos de los parámetros. Éstos deben ser clases o interfaces, nunca tipos básicos.
 - En la definición pueden especificarse restricciones sobre los parámetros, que deberán ser satisfechos por los valores concretos en la utilización posterior.

Un ejemplo simple

- Supongamos que queremos crear una clase que almacene dos elementos de otra clase.
 - No indicamos de qué clase concreta son los elementos a almacenar.
Suponemos que son de una clase T, donde T representa a cualquier clase.

```
public class Pareja {  
    private T primero, segundo;  
    public Pareja(T p, T s) {  
        primero = p;  
        segundo = s;  
    }  
    public T primero() {  
        return primero;  
    }  
    public T segundo() {  
        return segundo;  
    }  
}
```

```
public void primero(T p) {  
    primero = p;  
}  
public void segundo(T s) {  
    segundo = s;  
}  
...  
}
```

Un ejemplo simple

- Supongamos que queremos crear una clase que almacene dos elementos de otra clase.
 - No indicamos de qué clase concreta son los elementos a almacenar.
Suponemos que son de una clase T, donde T representa a cualquier clase.

```
public class Pareja <T> {  
    private T primero, segundo;  
    public Pareja(T p, T s) {  
        primero = p;  
        segundo = s;  
    }  
    public T primero() {  
        return primero;  
    }  
    public T segundo() {  
        return segundo;  
    }  
}
```

```
public void primero(T p) {  
    primero = p;  
}  
public void segundo(T s) {  
    segundo = s;  
}  
...  
}
```

¿Cómo sabe el compilador que T no es una clase concreta, sino que representa a cualquier clase?

Añadiendo <T> a la cabecera

Un ejemplo simple

- Supongamos que queremos crear una clase que almacene dos elementos de otra clase.
 - No indicamos de qué clase concreta son los elementos a almacenar.
- Suponemos que son de una clase T, donde T representa a cualquier clase.

```
public class Pareja <T> {  
    private T primero, segundo;  
    public Pareja(T p, T s) {  
        primero = p;  
        segundo = s;  
    }  
    public T primero() {  
        return primero;  
    }  
    public T segundo() {  
        return segundo;  
    }  
}
```

```
public void primero(T p) {  
    primero = p;  
}  
public void segundo(T s) {  
    segundo = s;  
}  
...  
}
```

¿Cómo sabe el compilador que T no es una clase concreta, sino que representa a cualquier clase?

Añadiendo <T> a la cabecera

¿Cómo usar los objetos de esa clase?

```
public class Programa {  
    public static void main(String [] args) {  
        Pareja<String> parC = new Pareja<String>("hola", "adios");  
        Pareja<Integer> parE = new Pareja<Integer>(4, 9);  
        ...  
    }  
}
```

Un ejemplo simple

- Supongamos que queremos crear una clase que almacene dos elementos de otra clase.
 - No indicamos de qué clase concreta son los elementos a almacenar.
- Suponemos que son de una clase T, donde T representa a cualquier clase.

```
public class Pareja <T> {  
    private T primero, segundo;  
    public Pareja(T p, T s) {  
        primero = p;  
        segundo = s;  
    }  
    public T primero() {  
        return primero;  
    }  
    public T segundo() {  
        return segundo;  
    }  
}
```

```
public void primero(T p) {  
    primero = p;  
}  
public void segundo(T s) {  
    segundo = s;  
}  
...  
}
```

¿Cómo sabe el compilador que T no es una clase concreta, sino que representa a cualquier clase?

Añadiendo <T> a la cabecera

¿Cómo usar los objetos de esa clase?

```
public class Programa {  
    public static void main(String [] args) { // desde versión Java 1.7  
        Pareja<String> parC = new Pareja<>("hola", "adios");  
        Pareja<Integer> parE = new Pareja<>(4, 9);  
        ...  
    }  
}
```

Otro ejemplo: Clase Optional<T>

(vista en Tema 4)

- Un objeto Optional<T> puede contener o no un dato de la clase T.

```
Optional<String> o1 = Optional.of("hola");  
Optional<String> o2 = Optional.empty();
```

- Métodos de instancia:

```
boolean isPresent() // indica si hay dato o no  
T get()           // devuelve el dato almacenado o lanza  
                  // NoSuchElementException si no hay nada  
T orElse(T v)     // devuelve el dato almacenado o  
                  // devuelve v si no hay nada
```

- La clase tiene correctamente definidos equals y hashCode

Otro ejemplo: Clase Optional<T>

(vista en Tema 4)

```
public static Optional<Persona> buscar(List<Persona> datos, String nombre) {  
    int i = 0;  
    while ((i < datos.size()) && (!nombre.equals(datos.get(i).nombre())))  
        i++;  
    return (i < datos.size()) ? Optional.of(datos.get(i)) : Optional.empty();  
}
```

```
public static void main(String [] args) {  
    List<Persona> datos = new ArrayList<>() ;  
    ...  
    Optional<Persona> op = buscar(datos, "Pepe");  
    if (op.isPresent()) {  
        System.out.println(op.get());  
    } ...  
}
```

Clases genéricas con más de un parámetro

- Una clase genérica puede disponer de varios parámetros

– Ejemplo

```
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> p = new Pareja<>("Antonio", 10);
```

Métodos genéricos

- Un método también puede ser genérico, independientemente de si la clase en la que se define dicho método es o no genérica

```
public class Clase {  
    public static <T> String aCadena(Pareja<T> par) {  
        return "(" + par.primer() + "," + par.segundo() + ")";  
    }  
}
```

¿Cómo sabe el compilador que T no es una clase concreta?

Añadiendo <T> a la cabecera del método

```
...  
Pareja<String> parC = new Pareja<>("hola", "adios");  
System.out.println(Clase.aCadena(parC));  
...
```

(hola,adios)

Métodos genéricos

- Un método también puede ser genérico, independientemente de si la clase en la que se define dicho método es o no genérica

```
public class Clase {  
    public static <T>String aCadena(Pareja<T> par) {  
        return "(" + par.primer() + "," + par.segundo() + ")";  
    }  
}
```

¿Cómo sabe el compilador que T no es una clase concreta?

Añadiendo <T> a la cabecera del método

```
...  
Pareja<Integer> parE = new Pareja<>(4,9);  
System.out.println(Clase.aCadena(parE));  
...
```

(4,9)

Parámetros anónimos

- Cuando un parámetro no se utiliza en el cuerpo del método genérico, puede utilizarse el símbolo “?” del modo siguiente:

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

Es equivalente a definir

```
public class Clase {  
    public static <T> String aCadena(Pareja<T> par) {  
        return "(" + par.primer() + "," + par.segundo() + ")";  
    }  
}
```

Restricciones sobre los parámetros

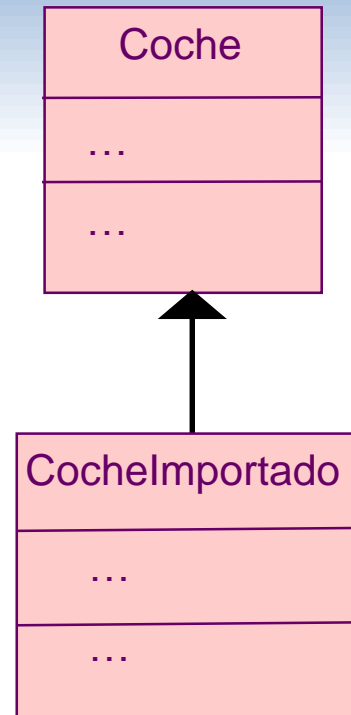
- Supongamos la siguiente clase con un método genérico:

```
public class Programa {  
    public static <T> void añadir(Coleccion<T> orig, Coleccion<T> dest) {  
        ...  
    }  
  
    ...  
}
```

Restricciones sobre los parámetros

- Supongamos la siguiente clase con un método genérico:

```
public class Programa {  
    public static <T> void añadir(Coleccion<T> orig, Coleccion<T> dest) {  
        ...  
    }  
  
    public static void main(String [] args) {  
        Coleccion<CocheImportado> cci = new Coleccion<>();  
        ...  
        Coleccion<Coche> cc = new Coleccion<>();  
        ...  
        añadir(cci, cc);  
    }  
}
```

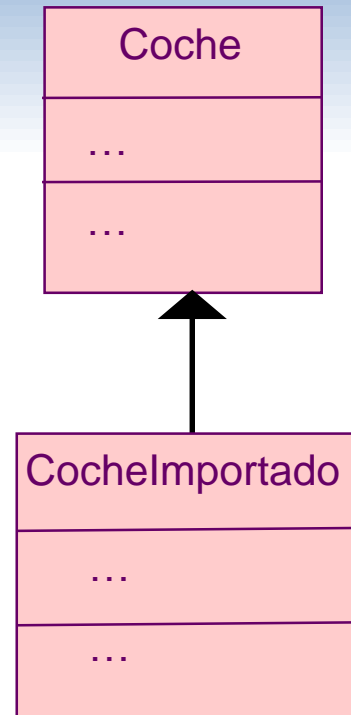


El código no compila, ¿por qué?

Restricciones sobre los parámetros

- Supongamos la siguiente clase con un método genérico:

```
public class Programa {  
    public static <T> void añadir(Coleccion<T> orig, Coleccion<T> dest) {  
        ...  
    }  
  
    public static void main(String [] args) {  
        Coleccion<CocheImportado> cci = new Coleccion<>();  
        ...  
        Coleccion<Coche> cc = new Coleccion<>();  
        ...  
        añadir(cci, cc);  
    }  
}
```



El método `añadir(Coleccion<T>, Coleccion<T>)` **no es aplicable** en la forma `añadir(Coleccion<CocheImportado>, Coleccion<Coche>)`

El código no compila, ¿por qué?

Restricciones sobre los parámetros

- Solución: Sobre un parámetro anónimo se pueden especificar restricciones

Alternativa 1

```
public class Programa {  
    public static <T> void añadir(Coleccion<T> orig, Coleccion<? super T> dest) {  
        ...  
    }  
    ...  
}
```

El método `añadir(Coleccion<T>, Coleccion<? super T>)` **sí es aplicable en la forma** `añadir(Coleccion<CocheImportado>, Coleccion<Coche>)`

? super T indica que los elementos de la colección `dest` son de tipo T o de alguna superclase de T

Restricciones sobre los parámetros

- Solución: Sobre un parámetro anónimo se pueden especificar restricciones

Alternativa 2

```
public class Programa {  
    public static <T> void añadir(Coleccion<? extends T> orig, Coleccion<T> dest) {  
        ...  
    }  
    ...  
}
```

El método añadir(Coleccion<? extends T>, Coleccion<T>) **sí es aplicable** en la forma añadir(Coleccion<CocheImportado>, Coleccion<Coche>)

? extends T indica que los elementos de la colección **orig** son de tipo T o de alguna subclase de T

Restricciones sobre los parámetros


- Solución: Sobre un parámetro anónimo se pueden especificar restricciones

Alternativa 3

```
public class Programa {  
    public static <T> void añadir(Coleccion<? extends T> orig, Coleccion<? super T> dest) {  
        ...  
    }  
    ...  
}
```

El método añadir(Coleccion<? extends T>, Coleccion<? super T>) sí es aplicable en la forma añadir(Coleccion<CocheImportado>, Coleccion<Coche>)

Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables 
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias

Interfaces para objetos ordenables

- Una clase puede especificar una relación de orden para los objetos creados a partir de la misma mediante:
 - 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 OtraClase1 implements Comparator<Persona> {  
    ...  
}  
public class OtraClase2 implements Comparator<Persona> {  
    ...  
}
```
- Si se intentan comparar dos objetos no comparables se lanza una excepción **ClassCastException**.

La interfaz Comparable<T>

(java.lang)

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

- *Orden natural* para una clase.
- **compareTo()** *no debe* entrar en contradicción con **equals()**.
 - | | | | | |
|---|----------|--------------------|-----------|---------------|
| { | negativo | si receptor (this) | menor que | parámetro (o) |
| | cero | si receptor (this) | igual que | parámetro (o) |
| | positivo | si receptor (this) | mayor que | parámetro (o) |
- 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
...	

Ejemplo: clase Persona

```
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;  
    }  
    @Override  
    public boolean equals(Object o) {  
        return (o instanceof Persona) &&  
            (edad == ((Persona) o).edad) &&  
            ((Persona) o).nombre.equals(nombre);  
    }  
    @Override  
    public int hashCode() {  
        return Objects.hash(nombre, edad);  
    }  
    ...  
}
```

Si no importa mayúsculas o minúsculas, se usa `equalsIgnoreCase` para los `String`

Persona implementa Comparable<Persona>

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

Si no importa mayúsculas o minúsculas, se usa `compareToIgnoreCase` para los `String`

Ejemplo: uso de *compareTo*

```
public class Prueba {  
    public static void main(String [] args) {  
        Persona p1 = new Persona("Juan", 35);  
        Persona p2 = new Persona("Pedro", 22);  
        int comp = p1.compareTo(p2);  
        if (comp < 0) {  
            System.out.println("Juan es menor que Pedro");  
        } else if (comp > 0) {  
            System.out.println("Juan es mayor que Pedro");  
        } else {  
            System.out.println("Juan y Pedro son iguales");  
        }  
    }  
}
```

La interfaz `Comparator<T>`

(`java.util`)

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

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- Orden Alternativo* para una clase.
- `compare()` no debe entrar en contradicción con `equals()`.

{	negativo	si o1 menor que o2
	cero	si o1 igual que o2
	positivo	si o1 mayor que o2

Ejemplo: OrdenAlternativoPersona implementa Comparator<Persona>

```
import java.util.*;

public class OrdenAlternativoPersona implements Comparator<Persona> {
    // Se comparan por nombres, y a igualdad de nombres, por edad
    @Override
    public int compare(Persona p1, Persona p2) {
        int resultado = p1.nombre().compareTo(p2.nombre());
        if (resultado == 0) {
            resultado = Integer.compare(p1.edad(), p2.edad());
            // resultado = p1.edad() - p2.edad();
        }
        return resultado;
    }
}
```

Ejemplo: uso de *compare*

```
import java.util.*;

public class Prueba {
    public static void main(String [] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        Comparator<Persona> ordAlt = new OrdenAlternativoPersona();
        int comp = ordAlt.compare(p1,p2);
        if (comp < 0) {
            System.out.println("Juan es menor que Pedro");
        } else if (comp > 0) {
            System.out.println("Juan es mayor que Pedro");
        } else {
            System.out.println("Juan y Pedro son iguales");
        }
    }
}
```

La interfaz `Comparator<T>`

(`java.util`)

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

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

```
public class Prueba {  
    public static void main(String [] args) {  
        ...  
        Comparator<Persona> ordAltRev = new OrdenAlternativoPersona().reversed();  
        ...  
        Comparator<Persona> ordAltNat = Comparator.naturalOrder();  
    }  
}
```

Mismo ejemplo de otra forma:

Ordenes alternativos simples que implementan Comparator<Persona>

```
import java.util.*;

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

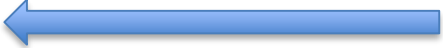
public class OrdenEdad implements Comparator<Persona> {
    // Se comparan por edad
    public int compare(Persona p1, Persona p2) {
        return Integer.compare(p1.edad(), p2.edad());
    }
}
```

Mismo ejemplo de otra forma: uso composición Comparator<T>

```
import java.util.*;

public class Prueba {
    public static void main(String [] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        Comparator<Persona> ordAlt =
            new OrdenNombre().thenComparing(new OrdenEdad());
        int comp = ordAlt.compare(p1,p2);
        if (comp < 0) {
            System.out.println("Juan es menor que Pedro");
        } else if (comp > 0) {
            System.out.println("Juan es mayor que Pedro");
        } else {
            System.out.println("Juan y Pedro son iguales");
        }
    }
}
```

Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables
- Colecciones y Correspondencias 
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias

Colecciones y Correspondencias

- Java proporciona en su paquete `java.util` herramientas para manejar Colecciones y Correspondencias (Asociaciones) de la forma más apropiada:
 - **Interfaces**. Para manipularlas de forma independiente de la implementación.
 - **Implementaciones**. Implementan su funcionalidad de una manera concreta.
 - **Algoritmos**. Para realizar determinadas operaciones sobre ellas, como ordenaciones, búsquedas, etc.
- **Beneficios** de usar el marco de Colecciones y Correspondencias:
 - Reduce los esfuerzos de aprendizaje, diseño y programación.
 - Incrementa calidad.
 - Aumenta la velocidad de la obtención de las soluciones.
 - Ayuda a la interoperabilidad.

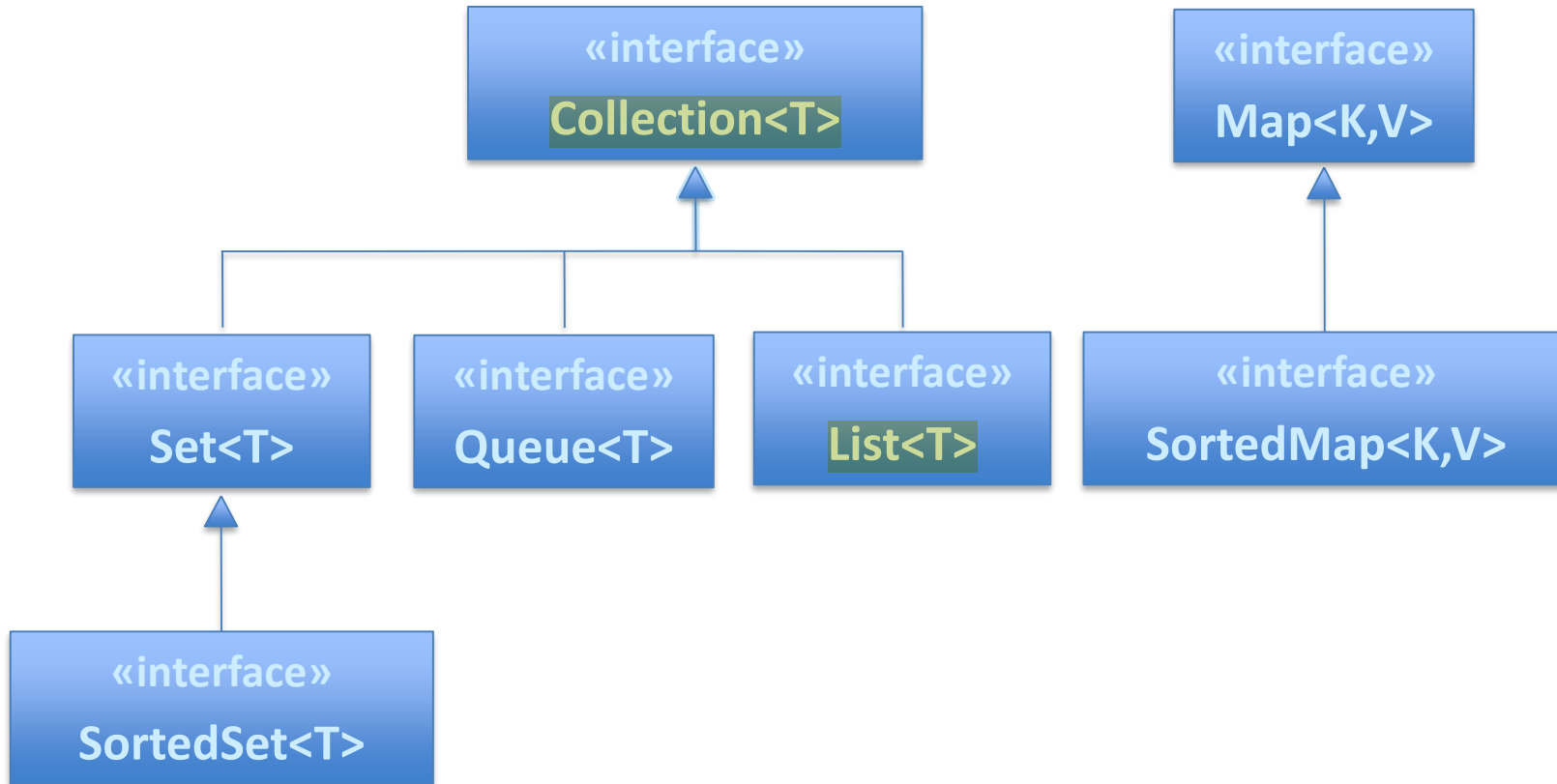
Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias



Interfaces básicas

(java.util)



Interfaces básicas

Interfaz que define las operaciones esenciales que deben ofrecer las clases que representan colecciones de objetos.

Interfaz que define las operaciones esenciales que deben ofrecer las clases que representan correspondencias (o asociaciones) de claves a valores.

Extiende `Collection<T>` 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>`

Extiende `Collection<T>` para colas de elementos

Extiende `Collection<T>` para secuencias de elementos, a los que se puede acceder atendiendo a su posición dentro de éstas.

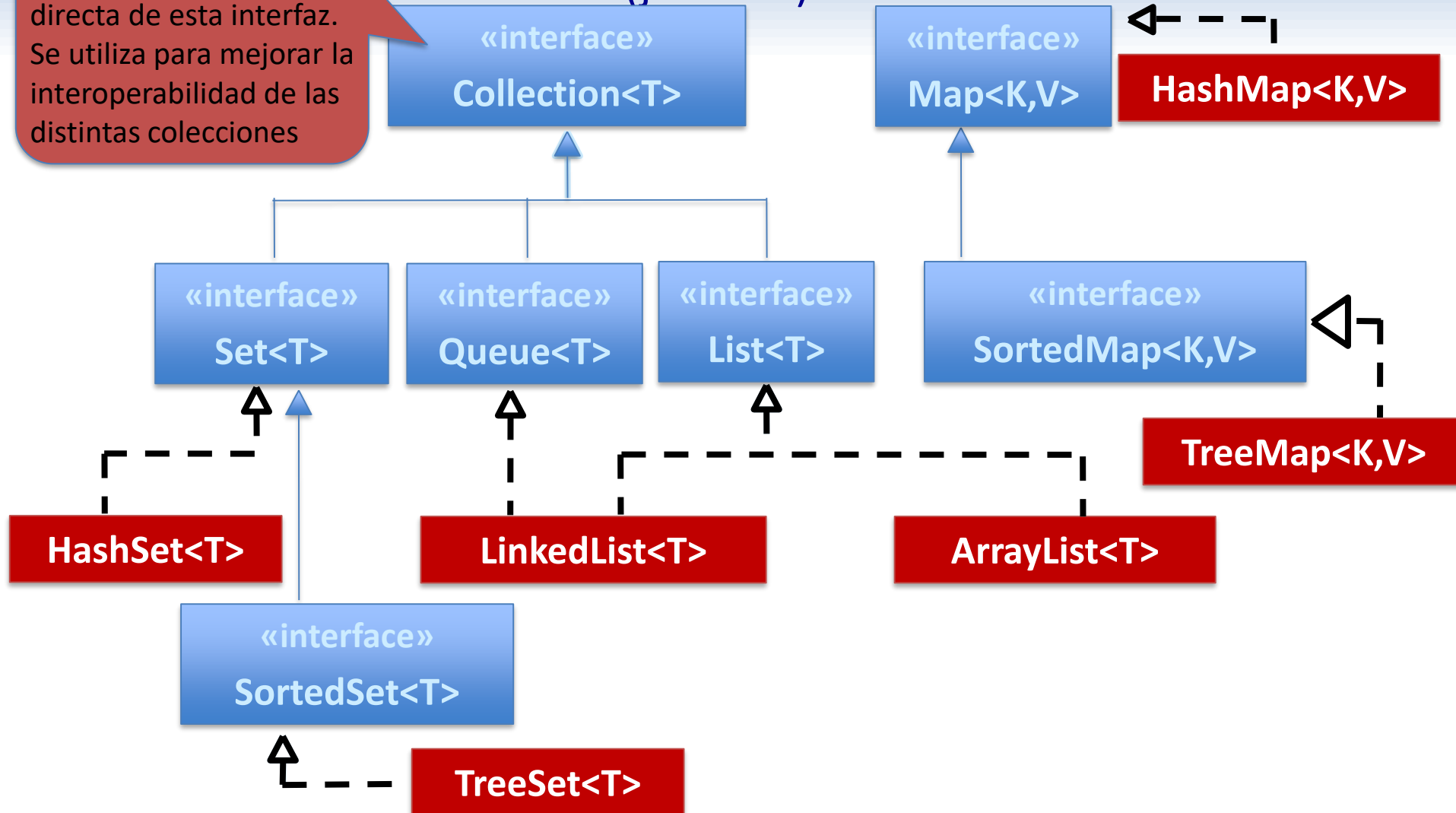
Extiende `Map<K,V>` para correspondencias que mantienen sus relaciones ordenadas por las claves.

Extiende `Set<T>` para conjuntos que mantienen sus elementos ordenados.

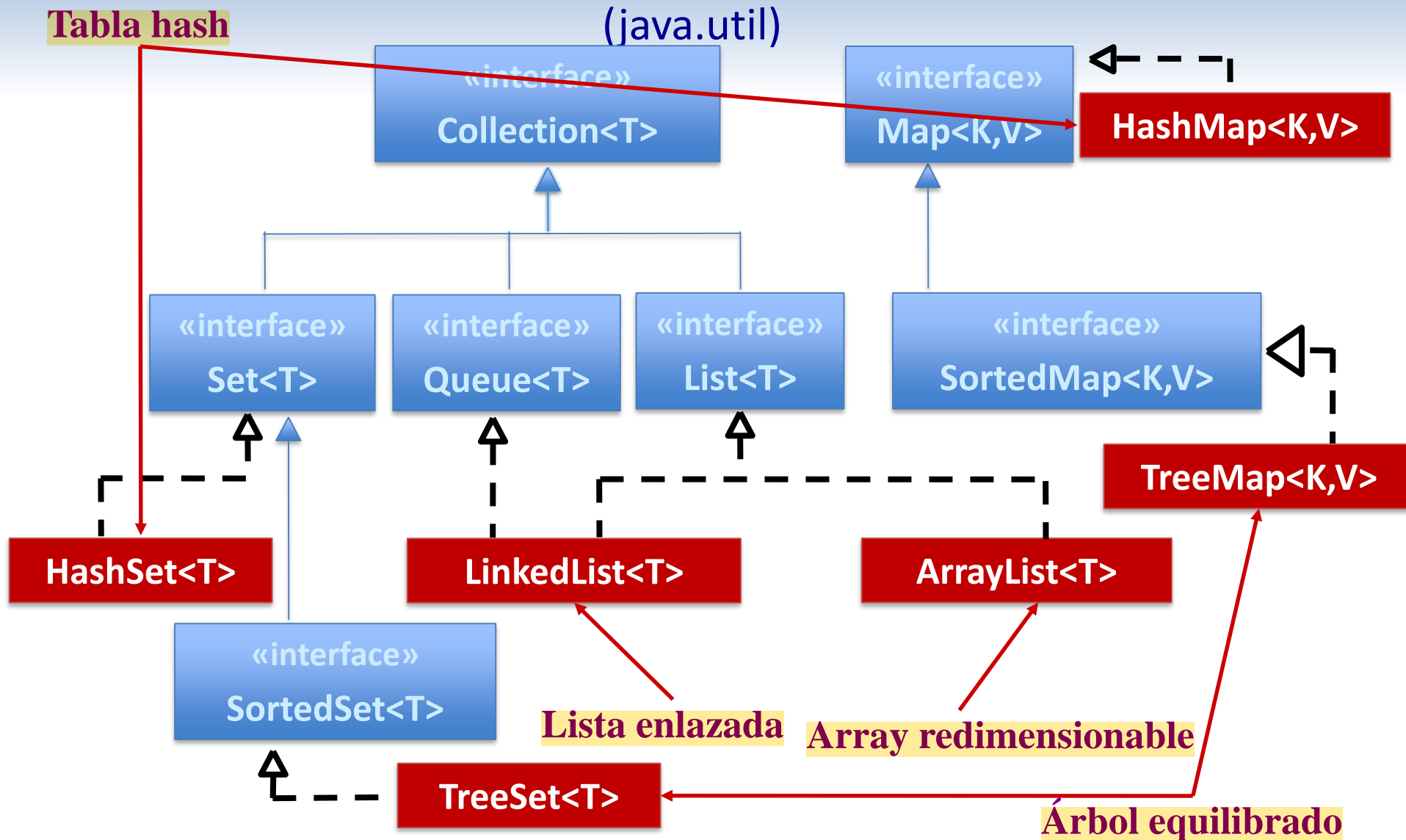
Interfaces básicas y sus implementaciones

No hay implementación directa de esta interfaz. Se utiliza para mejorar la interoperabilidad de las distintas colecciones

(java.util)



Interfaces básicas y sus implementaciones



Resumen

¿Qué?

¿Cómo?

`Interfaz<...> co = new Implementación<>();`

Si Interfaz es:

La implementación puede ser

Collection

HashSet, TreeSet, LinkedList, ArrayList

Set

HashSet, TreeSet

SortedSet

TreeSet

Queue

LinkedList

List

LinkedList, ArrayList

Map

HashMap, TreeMap

SortedMap

TreeMap

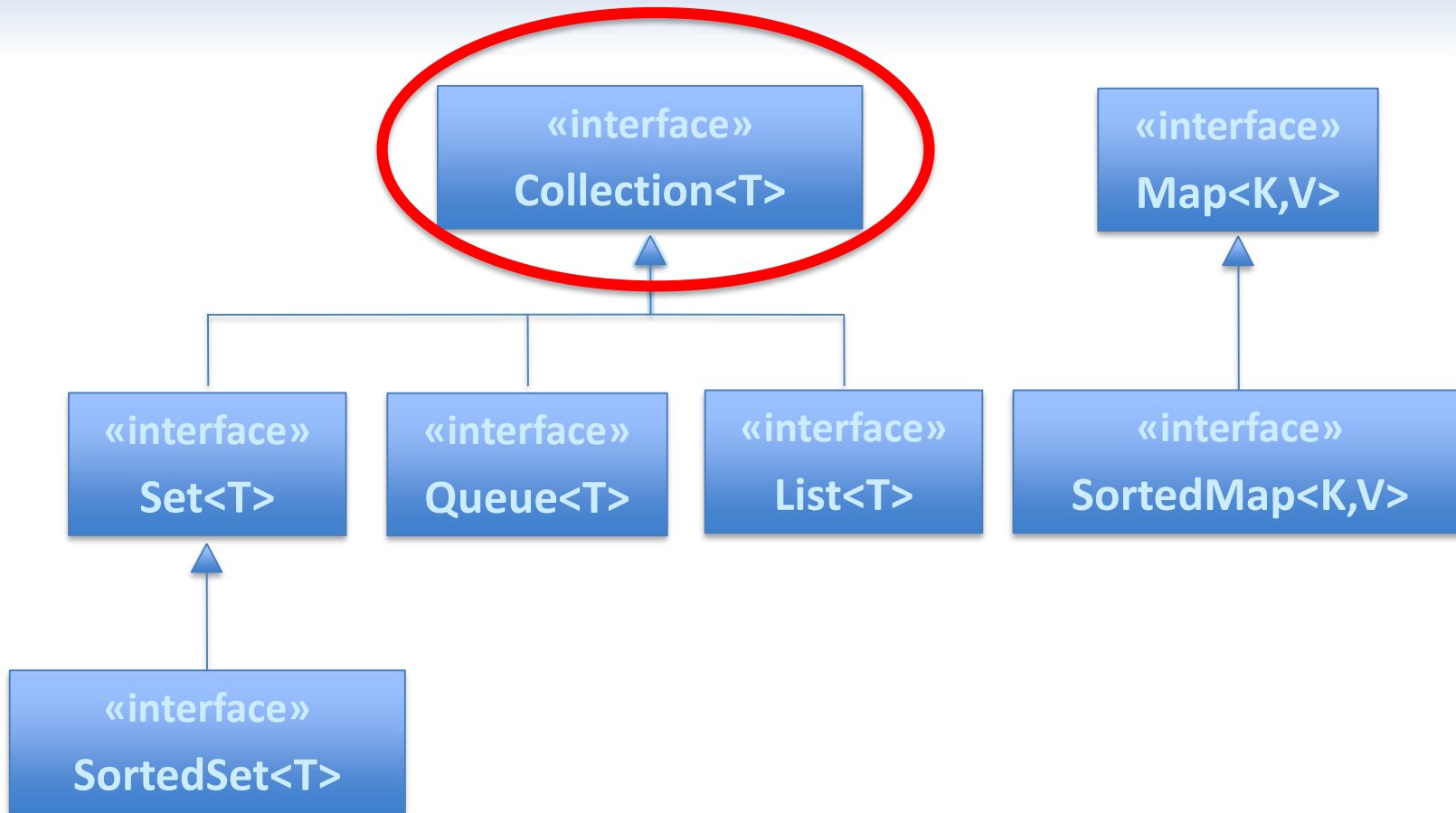
Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias



Interfaces básicas

(java.util)



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);
```

```
boolean remove(Object element);
```

```
// Operaciones con grupos de elementos
```

```
boolean containsAll(Collection<?> c);
```

```
boolean addAll(Collection<? extends T> c);
```

```
boolean removeAll(Collection<?> c);
```

```
boolean retainAll(Collection<?> c);
```

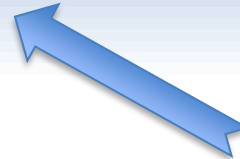
```
void clear();
```

```
// Operaciones con arrays
```

```
Object[] toArray();
```

```
<S> S[] toArray(S[] a);
```

```
}
```



La interfaz `Iterable<T>`

`(java.lang)`

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

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

```
Collection<String> c = ...;
```

```
c.add(...);
```

```
...
```

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

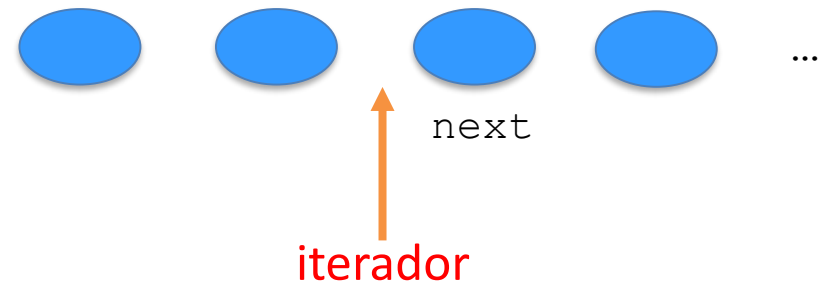
```
...
```

La interfaz `Iterator<T>`

`(java.util)`

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

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



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

Ejemplos: uso de iteradores

Con estos ejemplos se pone de manifiesto cómo podemos utilizar la interfaz `Collection<T>` para manejar colecciones sin conocer realmente el tipo concreto de colección (conjunto, lista, ...)

Recorrido completo para consultar los elementos

- Mostrar una colección de cadenas en pantalla.

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

Ejemplos: uso de iteradores

No hace falta recorrer la colección si es para mostrarla con el formato [e1, e2, ... , en]

- Mostrar una colección de cadenas en pantalla.

```
public static void mostrar(Collection<String> c) {  
    System.out.println(c);  
}
```

Ejemplos: uso de iteradores

Búsqueda

- Comprobar si en una colección de cadenas hay alguna cuya longitud sea mayor que un valor dado.

```
public static boolean hay(Collection<String> c, int umbral) {  
    Iterator<String> iter = c.iterator();  
    boolean encontrada = false;  
    while (!encontrada && iter.hasNext()) {  
        if ((iter.next()).length() > umbral) {  
            encontrada = true;  
        }  
    }  
    return encontrada;  
}
```

Ejemplos: uso de iteradores

Recorrido completo (o parcial) con (posible) eliminación de elementos de la colección

- Eliminar de una colección de cadenas aquellas cuya longitud sea mayor que un valor dado.

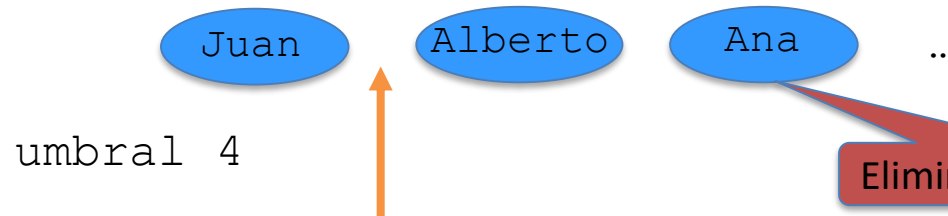
```
public static void eliminar(Collection<String> c, int umbral) {  
    Iterator<String> iter = c.iterator();  
    while (iter.hasNext()) {  
        if ((iter.next()).length() > umbral) {  
            iter.remove();  
        }  
    }  
}
```


Ejemplos: uso de iteradores

Recorrido completo (o parcial) con (posible) eliminación de elementos de la colección (otro ejemplo)

- Eliminar de una colección de cadenas aquellas cuya longitud sea mayor que un valor dado y comiencen por letra mayúscula.

```
public static void eliminar(Collection<String> c, int umbral) {  
    Iterator<String> iter = c.iterator();  
    while (iter.hasNext()) {  
        if ((iter.next()).length() > umbral &&  
            Character.isUpperCase(iter.next().charAt(0))) {  
            iter.remove();  
        }  
    }  
}
```

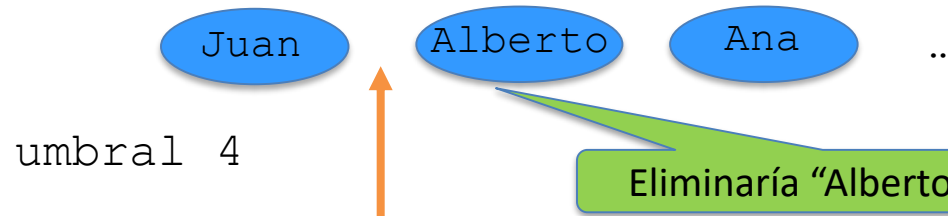


Ejemplos: uso de iteradores

Recorrido completo (o parcial) con (posible) eliminación de elementos de la colección (otro ejemplo)

- Eliminar de una colección de cadenas aquellas cuya longitud sea mayor que un valor dado y comiencen por letra mayúscula.

```
public static void eliminar(Collection<String> c, int umbral) {  
    Iterator<String> iter = c.iterator();  
    String str;  
    while (iter.hasNext()) {  
        str = iter.next();  
        if (str.length() > umbral  
            && Character.isUpperCase(str.charAt(0))) {  
            iter.remove();  
        }  
    }  
}
```



Ejemplos: uso de iteradores

Recorrido completo (o parcial) con modificación de los elementos (todos o algunos) de la colección

- Modificar la edad de cada una de las personas almacenadas en la colección.

```
public static void modificarEdad(Collection<Persona> c, int inc) {  
    Iterator<Persona> iter = c.iterator();  
    Persona p;  
    while (iter.hasNext()) {  
        p = iter.next();  
        p.setEdad(p.getEdad()+inc);  
    }  
}
```

Uso de for-each

- Al igual que se ha utilizado con Arrays, el for-each permite recorrer una colección por completo desde el primero al último de los elementos (tanto para consultarlos como para modificarlos)

Así, por ejemplo, el código:

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

Puede escribirse alternativamente como (mejor, más sencillo):

```
public static void mostrar(Collection<String> c) {  
    for (String cad : c) {  
        System.out.println(cad);  
    }  
}
```

Uso de for-each

- Al igual que se ha utilizado con Arrays, el for-each permite recorrer una colección por completo desde el primero al último de los elementos (tanto para consultarlos como para modificarlos)

Y el código:

```
public static void modificarEdad(Collection<Persona> c, int inc) {  
    Iterator<Persona> iter = c.iterator();  
    Persona p;  
    while (iter.hasNext()) {  
        p = iter.next();  
        p.setEdad(p.getEdad()+inc);  
    }  
}
```

Puede escribirse alternativamente como (mejor, más sencillo):

```
public static void modificarEdad(Collection<Persona> c, int inc) {  
    for (Persona p : c) {  
        p.setEdad(p.getEdad()+inc);  
    }  
}
```

Uso de for-each

- No es recomendable su uso para las búsquedas (ineficiente)

Por ejemplo el siguiente código debe evitarse:

```
public static boolean hay(Collection<String> c, int umbral) {  
    boolean encontrada = false;  
    for (String cad : c) {  
        if (cad.length() > umbral) {  
            encontrada = true;  
        }  
    }  
    return encontrada;  
}
```

Y utilizar en su lugar iteradores (mejor, más eficiente):

```
public static boolean hay(Collection<String> c, int umbral) {  
    Iterator<String> iter = c.iterator();  
    boolean encontrada = false;  
    while (!encontrada && iter.hasNext()) {  
        if ((iter.next()).length() > umbral) {  
            encontrada = true;  
        }  
    }  
    return encontrada;  
}
```

Uso de for-each

- La sentencia for-each **no permite eliminar elementos** de la colección mientras se está recorriendo. Esto habría que hacerlo con iteradores como se ha visto antes.
- Ejemplo:

```
public static void eliminar(Collection<String> c, int umbral) {  
    for(String cad : c) {  
        if (cad.length() > umbral) {  
            c.remove(cad);  
        }  
    }  
}
```

Exception in thread "main" [java.util.ConcurrentModificationException](#)
at java.util.ArrayList\$Itr.checkForComodification([ArrayList.java:901](#))
at java.util.ArrayList\$Itr.next([ArrayList.java:851](#))
at prTema5Extra.OperacionesConColecciones.eliminar([OperacionesConColecciones.java:68](#))
at prTema5Extra.OperacionesConColecciones.main([OperacionesConColecciones.java:17](#))

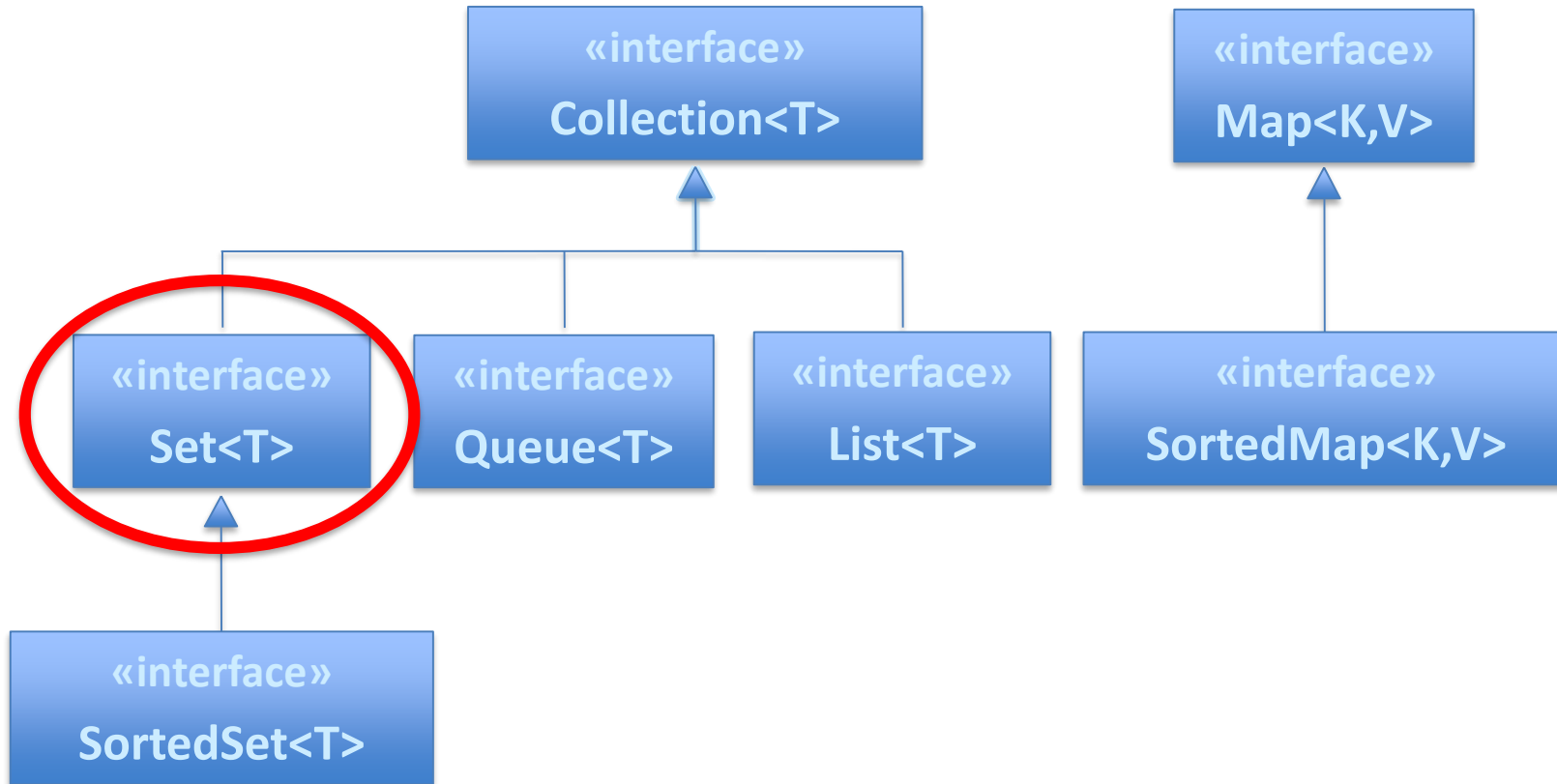
Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - **Conjuntos**, Listas y Colas
 - Correspondencias



Interfaces básicas

(java.util)

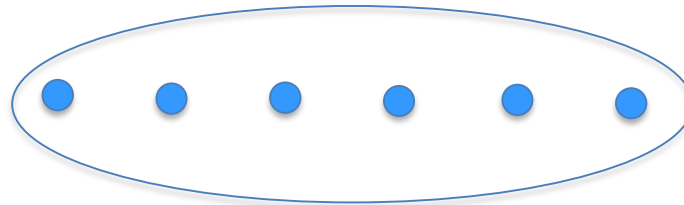


La interfaz `Set<T>`

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

```
public interface Set<T> extends Collection<T>
```

- Pensada para manipular conjuntos. Un conjunto es una colección sin 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()`

$$a = \emptyset$$

Crear un Conjunto constante

- El método `static of` de la interfaz `Set` permite crear un conjunto constante con los argumentos del método (a partir de JDK 1.9)
 - Puede tener un número variable de argumentos.
 - El conjunto así creado no puede modificarse (añadir o eliminar elementos).

```
Set<String> set = Set.of("Antonio", "Juan", "Luis");
```

```
set.add("Pedro"); // error
```

```
...
```

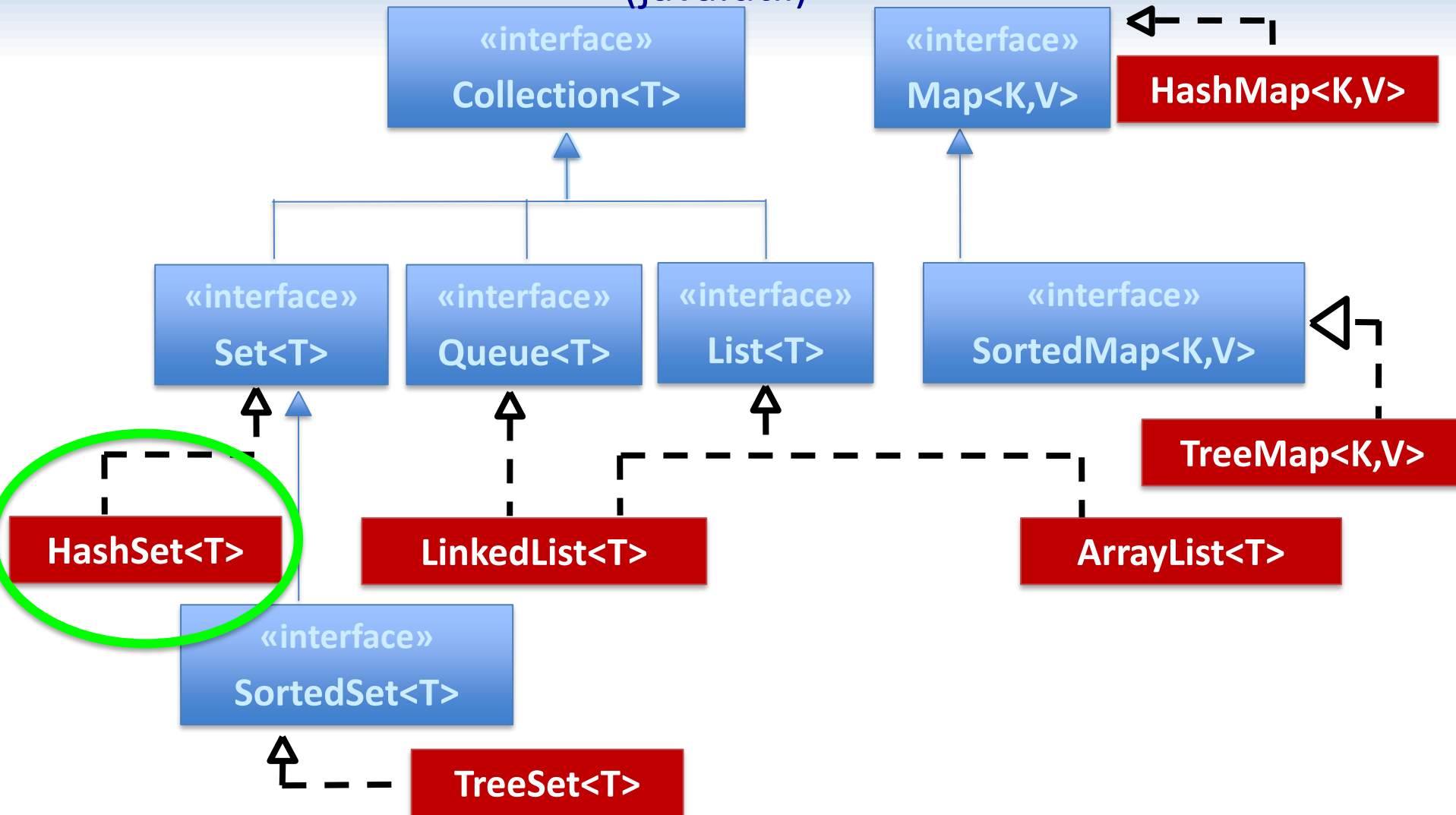
Formas de recorrer un Conjunto

Como cualquier Colección (`Collection<T>`), un Conjunto puede recorrerse (ver ejemplos del apartado Colecciones e Iteradores):

- Con iteradores (`Iterator<T>`)
 - recorridos completos para consultar elementos
 - búsquedas
 - recorridos completos (o parciales) con (posible) eliminación de elementos
 - recorridos completos (o parciales) con modificación de elementos
- Con un `for-each`
 - recorridos completos para consultar elementos
 - recorridos completos con modificación de elementos

Interfaces básicas y sus implementaciones

(java.util)

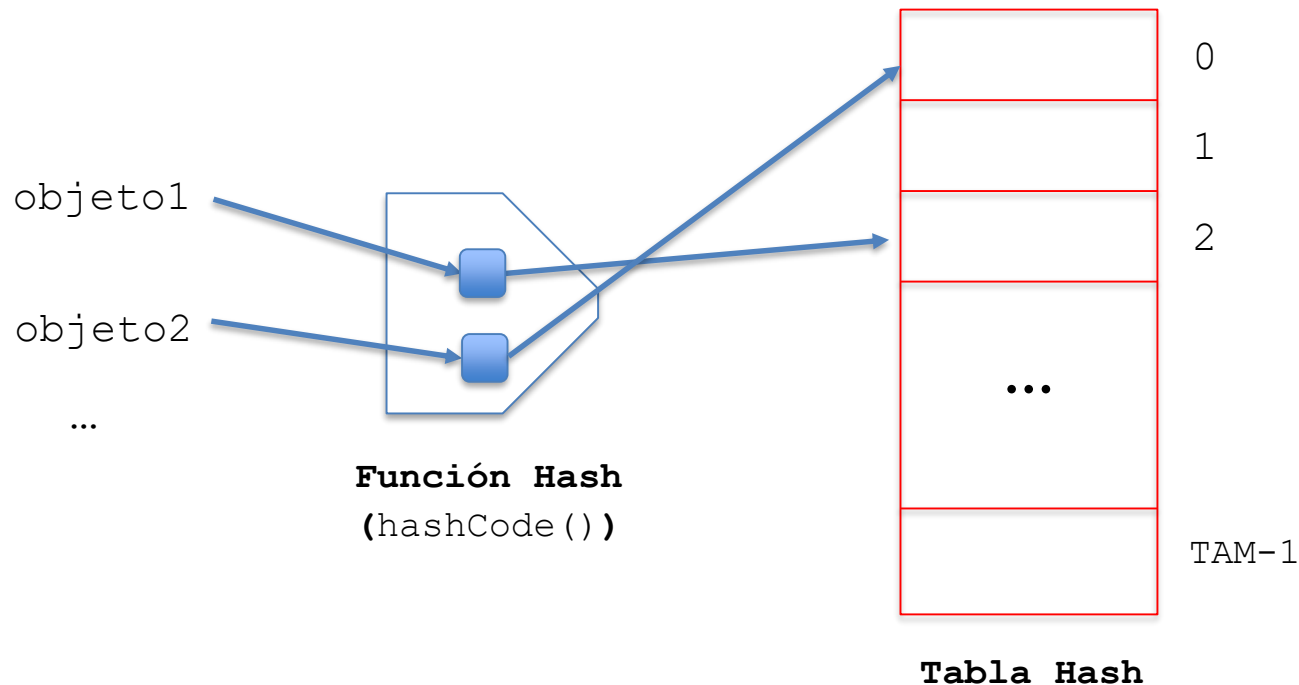


Implementación de `Set<T>`

`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.



Implementación 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 hash.
- Representación como Cadena de Caracteres (`toString()`):
 - Igual que Arrays
[e1, e2, ..., en]

Ejemplo: uso de HashSet<T>

Dado un array de cadenas, detectar cadenas repetidas y mostrar cadenas diferentes

```
import java.util.*;

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

ARGUMENTOS:

SALIDA:

uno dos cuatro dos tres cuatro cinco

repetida: dos

repetida: cuatro

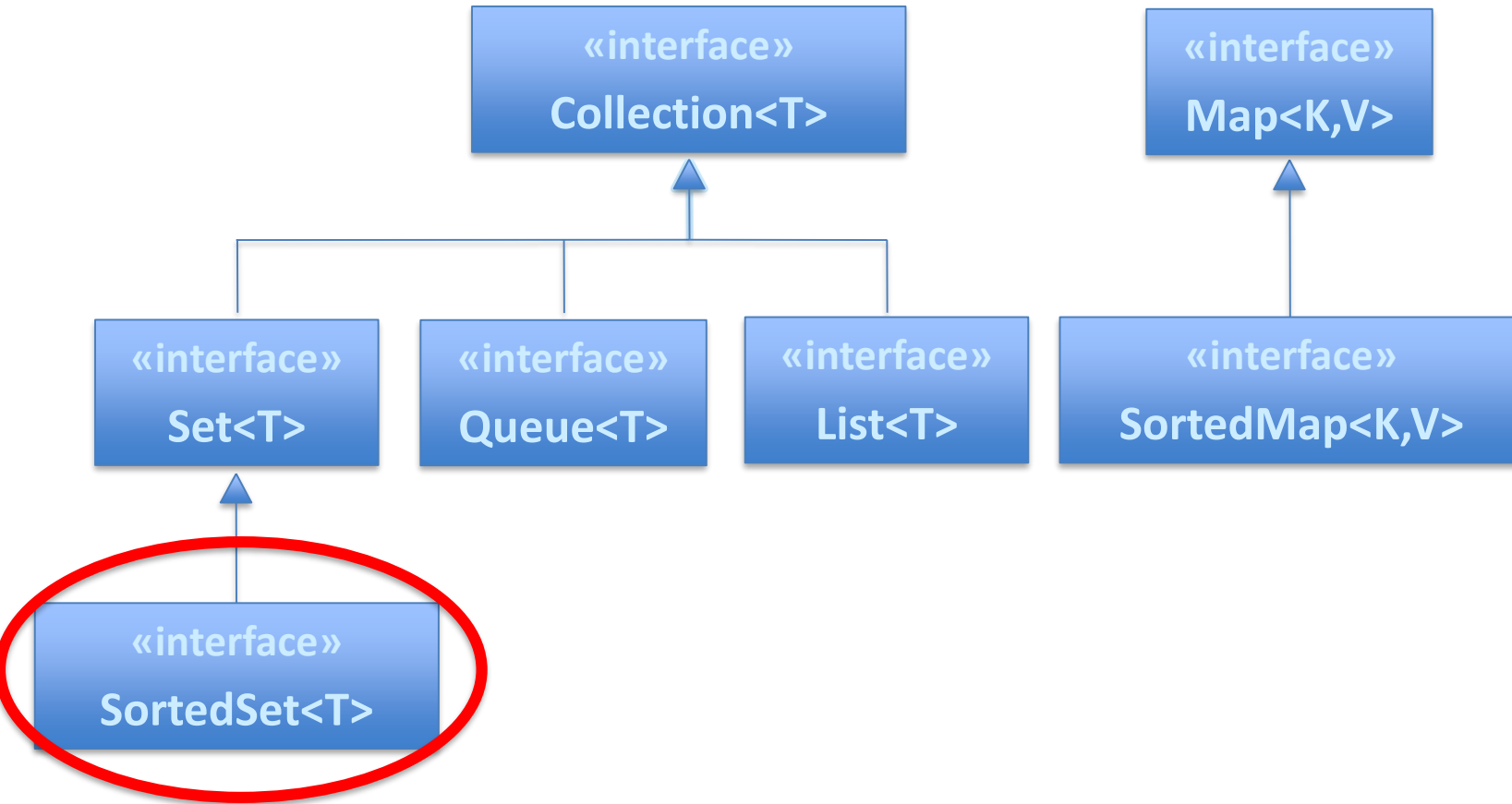
5 cadenas diferentes: [tres, dos, uno, cinco, cuatro]

¿orden?

Tabla hash

Interfaces básicas

(java.util)



La interfaz `SortedSet<T>`

- Extiende `Set<T>` para proporcionar la funcionalidad de conjuntos con elementos ordenados.
- El orden utilizado es:
 - Por defecto el *orden natural* (`Comparable<T>`)
 - El *orden alternativo* especificado por un `Comparator<T>` en el constructor de la clase que implemente esta interfaz

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 primero y último
```

```
T first();
```

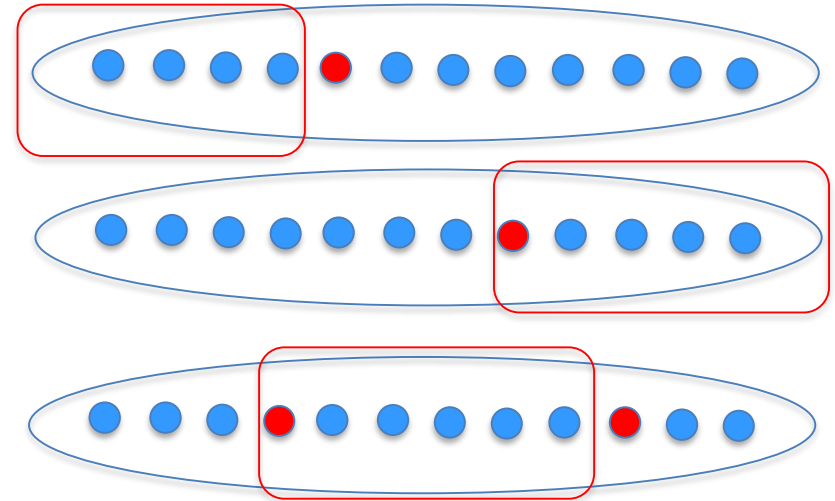
```
T last();
```

```
// acceso al comparador
```

```
Comparator<? super T> comparator(); // devuelve null si se usa el orden natural
```

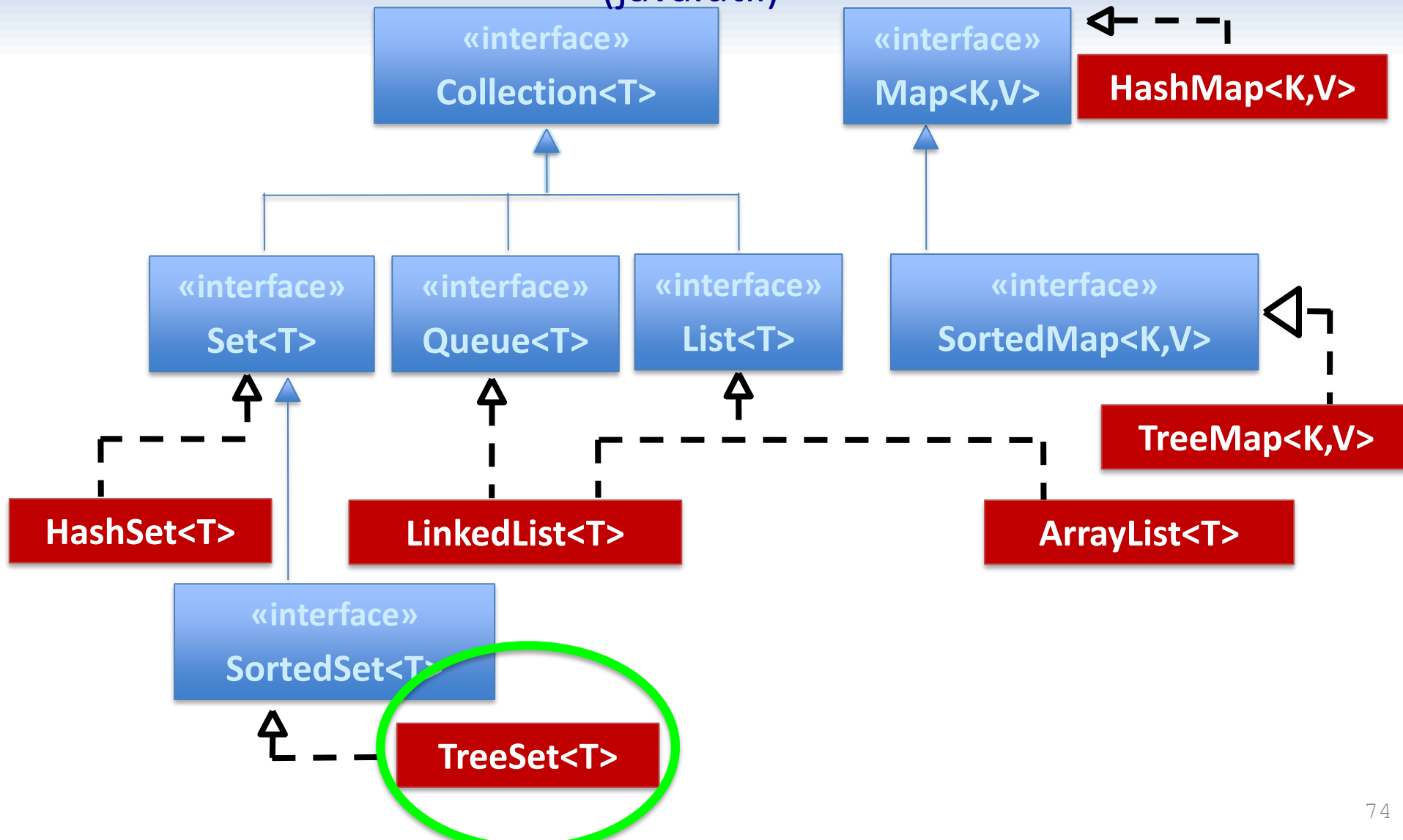
```
...
```

```
}
```



Interfaces básicas y sus implementaciones

(java.util)

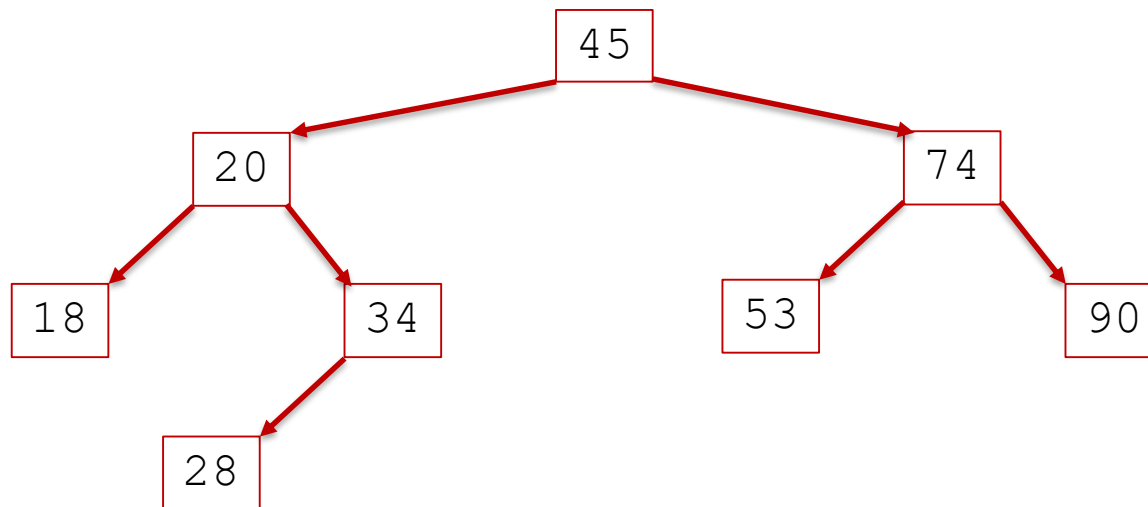


Implementación de SortedSet<T>

`java.util` proporciona la siguiente implementación directa de SortedSet<T> (al mismo tiempo es una implementación indirecta de Set<T>):

TreeSet<T>

- Guarda los datos en un árbol binario equilibrado.
- Búsqueda y modificación más lenta que en HashSet<T>.



Implementación de SortedSet<T>

`java.util` proporciona la siguiente implementación directa de `SortedSet<T>` (al mismo tiempo es una implementación indirecta de `Set<T>`):

TreeSet<T>

- Guarda los datos en un árbol binario equilibrado.
- Búsqueda y modificación más lenta que en `HashSet<T>`.
- Constructores:
 - `TreeSet()` // Orden natural
 - `TreeSet(Comparator<? super T> o)` // Orden alternativo
 - `TreeSet(Collection<? extends T> c)` // Orden natural
 - `TreeSet(SortedSet<T> s)` // Mismo orden que s
- Representación como Cadena de Caracteres (`toString()`):
 - Igual que Arrays
[e1, e2, ..., en]

Ejemplo: uso de TreeSet<T>

Dado un array de cadenas, detectar cadenas repetidas y mostrar cadenas diferentes **ordenadas**

```
import java.util.*;

public class Duplicados {
    public static void main(String[] args) {
        Set<String> s = new TreeSet<>();
        // SortedSet<String> s = new TreeSet<>();
        for (String arg : args) {
            if (!s.add(arg)) {
                System.out.println("repetida: " + arg);
            }
        }
        System.out.println(s.size() + " cadenas diferentes: " + s);
    }
}
```

ARGUMENTOS:

SALIDA:

uno dos cuatro dos tres cuatro cinco

repetida: dos

repetida: cuatro

5 cadenas diferentes: [cinco, cuatro, dos, tres, uno]

¿orden?

Lexicográfico

Ejemplo: uso de TreeSet<T>

Dado un array de cadenas, detectar cadenas repetidas y mostrar **subconjunto** de cadenas diferentes **ordenadas**

```
import java.util.*;

public class Duplicados {
    public static void main(String[] args) {
        SortedSet<String> s = new TreeSet<>();
        for (String arg : args) {
            if (!s.add(arg)) {
                System.out.println("repetida: " + arg);
            }
        }
        System.out.println(s.headSet("dos" ));
    }
}
```

ARGUMENTOS:	uno dos cuatro dos tres cuatro cinco
SALIDA:	repetida: dos repetida: cuatro [cinco, cuatro]

Ejemplo: Recordemos la clase Persona con su Orden Natural definido

```
public class Persona implements Comparable<Persona> {  
    private String nombre;  
    private int edad;  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
    ...  
    @Override  
    public int compareTo(Persona p) {  
        int resultado = Integer.compare(edad,p.edad);  
        // int resultado = edad - p.edad;  
        if (resultado == 0) {  
            resultado = nombre.compareTo(p.nombre);  
        }  
        return resultado;  
    }  
}
```

Ejemplo: Recordemos el Orden Alternativo para Persona

```
import java.util.*;

public class OrdenAlternativoPersona implements Comparator<Persona> {
    // Se comparan por nombres, y a igualdad de nombres, por edad
    @Override
    public int compare(Persona p1, Persona p2) {
        int resultado = p1.nombre().compareTo(p2.nombre());
        if (resultado == 0) {
            resultado = Integer.compare(p1.edad(), p2.edad());
            // resultado = p1.edad() - p2.edad();
        }
        return resultado;
    }
}
```

Ejemplo: Queremos diseñar una clase Asamblea para almacenar un grupo de personas ordenadas

```
public class Asamblea { // grupo de personas (ordenadas)
    ...
    public Asamblea() {
        ... // Se crea una asamblea que utilizará el orden natural de Persona para ordenar
    }
    ...
}
```

```
Asamblea a = new Asamblea();
```

Ejemplo: Queremos diseñar una clase Asamblea para almacenar un grupo de personas ordenadas

```
public class Asamblea { // grupo de personas (ordenadas)

    private SortedSet<Persona> personas; // conjunto ordenado para almacenar las personas

    public Asamblea() {
        personas = new TreeSet<>(); // orden natural para ordenar
    }

    ...
}
```

Ejemplo: Queremos diseñar una clase Asamblea para almacenar un grupo de personas ordenadas

```
public class Asamblea { // grupo de personas (ordenadas)
    ...
    public Asamblea() {
        ... // Se crea una asamblea que utilizará el orden natural de Persona para ordenar
    }
    public Asamblea(Comparator<Persona> comp) {
        ... // Se crea una asamblea que utilizará el orden alternativo de Persona para ordenar
    }
    ...
}
```

```
Asamblea a1 = new Asamblea();
```

```
Asamblea a2 = new Asamblea(new OrdenAlternativoPersona());
```

Ejemplo: Queremos diseñar una clase Asamblea para almacenar un grupo de personas ordenadas

```
public class Asamblea { // grupo de personas (ordenadas)

    private SortedSet<Persona> personas; // conjunto ordenado para almacenar las personas

    public Asamblea() {
        personas = new TreeSet<>(); // orden natural para ordenar
    }

    public Asamblea(Comparator<Persona> comp) {
        personas = new TreeSet<>(comp); // orden alternativo para ordenar
    }

    ...
}
```

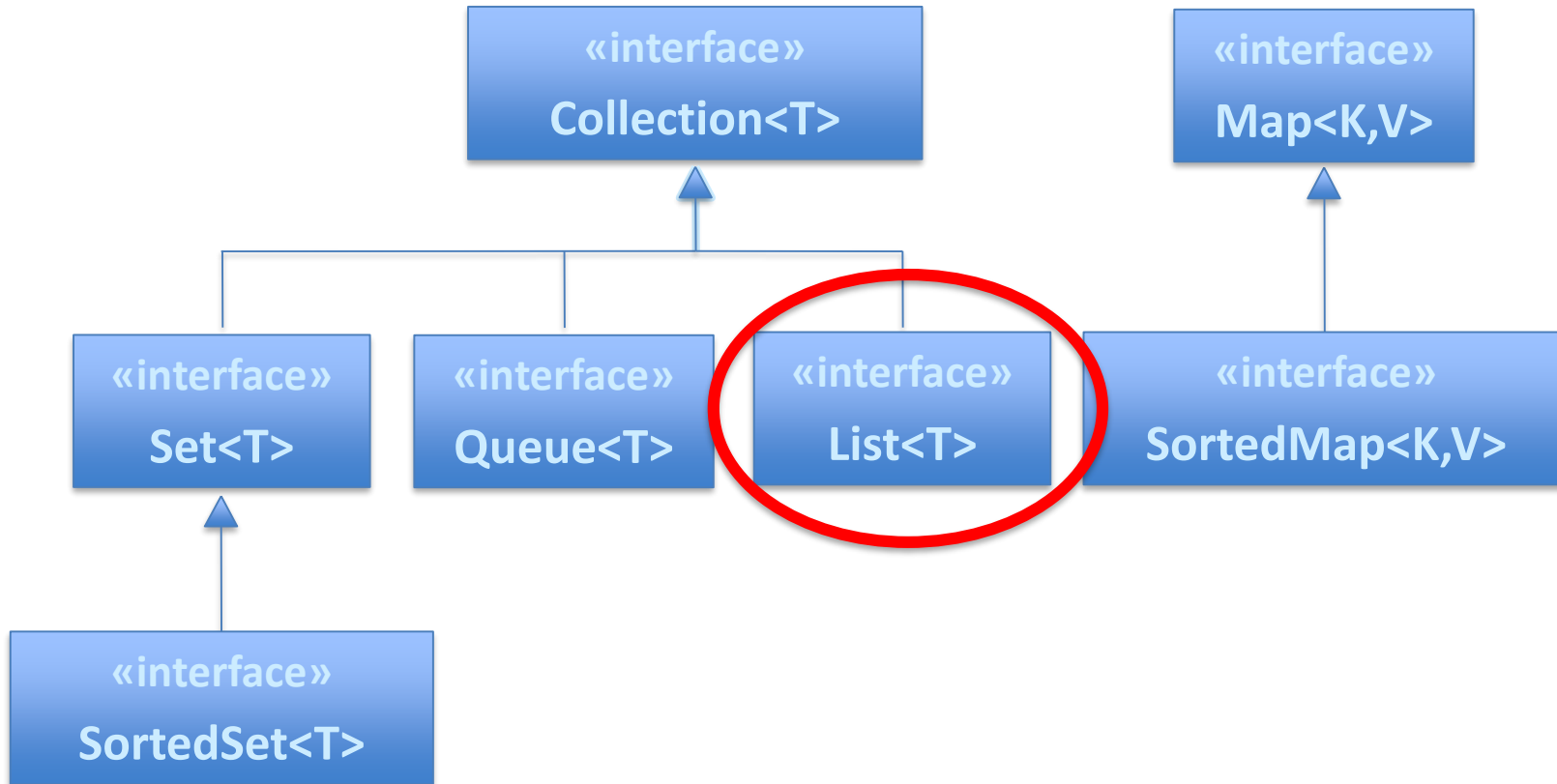
Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias



Interfaces básicas

(java.util)

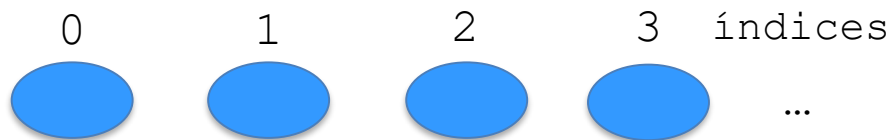


La interfaz `List<T>`

- La interfaz `List<T>` hereda de la interfaz `Collection<T>`

```
public interface List<T> extends Collection<T>
```

- Pensada para manipular listas. Una **Lista** es una colección de elementos que forman una **secuencia** (cada elemento tiene una **posición (índice)**).



- Permite acceso por posición ($0 \dots \text{size}() - 1$)
 - Un índice ilegal produce el lanzamiento de una excepción `IndexOutOfBoundsException`.

La interfaz List<T>

```
public interface List<T> extends Collection<T> {
```

```
// Acceso posicional
```

```
T get(int index);
```

```
T set(int index, T element);
```

```
void add(int index, T element);
```

```
boolean addAll(int index, Collection<? extends T> c);
```

```
T remove(int index);
```

```
// Búsqueda
```

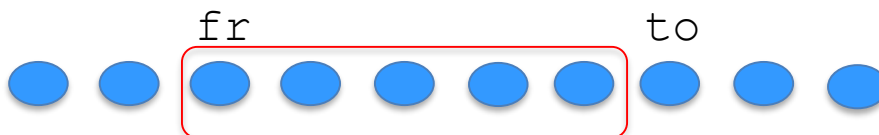
```
int indexOf(Object o);
```

```
int lastIndexOf(Object o);
```

```
// Vista de subrango
```

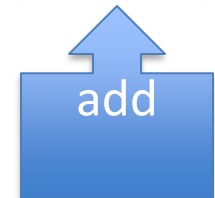
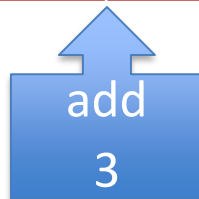
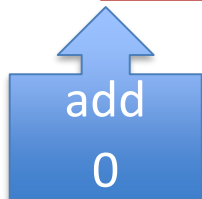
```
List<T> subList(int fromIndex, int toIndex);
```

```
...  
}
```





Hola	Que	Tal	Como	Estás	Bien	Aquí
------	-----	-----	------	-------	------	------



Crear una Lista constante

- El método `static of` de la interfaz *List* permite crear una lista constante con los argumentos del método (a partir de JDK 1.9)
 - Puede tener un número variable de argumentos.
 - La lista así creada no puede modificarse (añadir o eliminar elementos).

```
List<String> lista = List.of("Antonio", "Juan", "Luis");  
lista.add("Pedro"); // error  
...
```


Formas de recorrer una Lista

Como cualquier Colección (`Collection<T>`), una Lista puede recorrerse (ver ejemplos del apartado Colecciones e Iteradores):

- Con iteradores (`Iterator<T>`)
 - recorridos completos para consultar elementos
 - búsquedas
 - recorridos completos (o parciales) con (posible) eliminación de elementos
 - recorridos completos (o parciales) con modificación de elementos
- Con un `for-each`
 - recorridos completos para consultar elementos
 - recorridos completos con modificación de elementos

Formas de recorrer una Lista

Además, una Lista puede recorrerse también:

- Con iteradores de lista (`ListIterator<T>`)  No lo usaremos
- Mediante índices
 - recorridos completos para consultar elementos
 - búsquedas
 - recorridos completos (o parciales) con (posible) eliminación e inserción de elementos
 - recorridos completos (o parciales) con modificación de elementos

Formas de recorrer una Lista

Ejemplo: Recorrido completo para consultar elementos mediante índices

Contar cuántos números pares hay en una lista de enteros

```
public static int contarPares(List<Integer> lista) {  
    int suma = 0;  
    for (int i = 0; i < lista.size(); i++) {  
        if (lista.get(i) % 2 == 0) {  
            suma++;  
        }  
    }  
    return suma;  
}
```

Formas de recorrer una Lista

Ejemplo: Búsqueda mediante índices

Comprobar si hay algún número par en una lista de enteros

```
public static boolean hayPar(List<Integer> lista) {  
    int i = 0;  
    while (i < lista.size() && lista.get(i) % 2 != 0) {  
        i++;  
    }  
    return i < lista.size();  
}
```


Formas de recorrer una Lista

Ejemplo: Recorrido completo (o parcial) con (posible) eliminación o inserción de elementos mediante índices

Eliminar los números pares de una lista de enteros

```
public static void eliminarPares(List<Integer> lista) {  
    int i = 0;  
    while (i < lista.size()) {  
        if (lista.get(i) % 2 == 0) {  
            lista.remove(i);  
        } else {  
            i++;  
        }  
    }  
}
```

Formas de recorrer una Lista

Ejemplo: Recorrido completo (o parcial) con (posible) modificación de elementos mediante índices

Incrementar en 1 cada valor de una lista de enteros

```
public static void incrementarValores(List<Integer> lista) {  
    for (int i = 0; i < lista.size(); i++) {  
        lista.set(i, lista.get(i)+1);  
    }  
}
```

Ordenando Listas

- No existe una interfaz `SortedList<T>` (equivalente a la `SortedSet<T>` para conjuntos), pero la interfaz `List<T>` incluye desde java 1.8 un método por defecto que permite ordenar listas:

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

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

Ejemplo:

```
List<Persona> lista = ...;  
...  
lista.sort(null);  
...  
lista.sort(new OrdenAlternativoPersona());
```

- Además, en la clase `Collections` (diferente de la interfaz `Collection<T>`) están los métodos estáticos `sort` (ver siguiente diapositiva)

Operaciones predefinidas sobre Listas

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

```
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> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
```

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

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

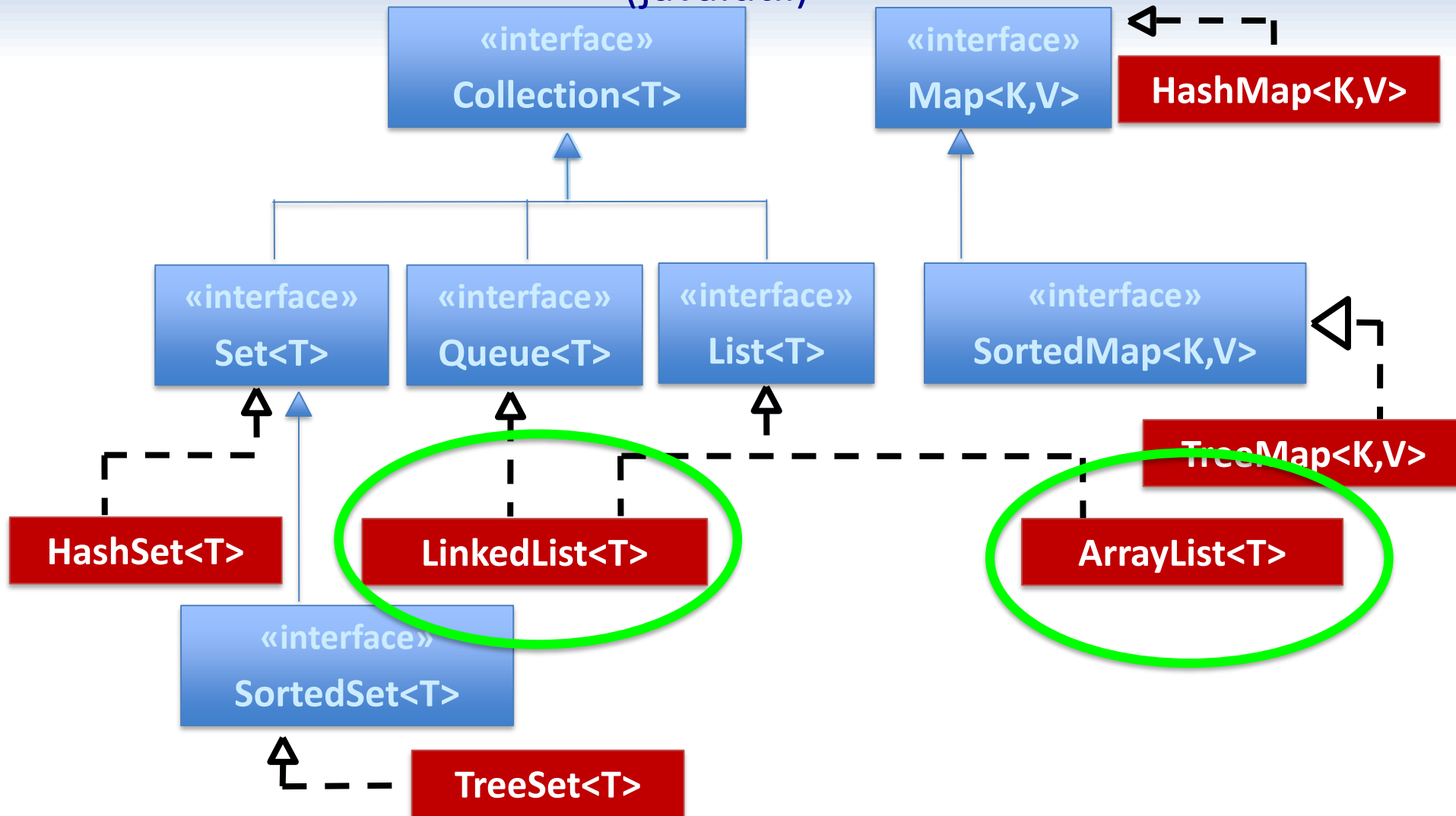
- La clase **java.util.Arrays** proporciona:

- Un método que devuelve una vista de un array como una lista (este método es un “puente” entre los arrays y las colecciones, al igual que toArray() de Collection<T>):

```
static <T> List<T> asList(T... a); // a es un array de T
```

Interfaces básicas y sus implementaciones

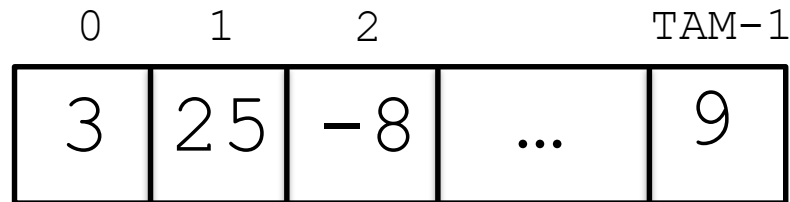
(java.util)



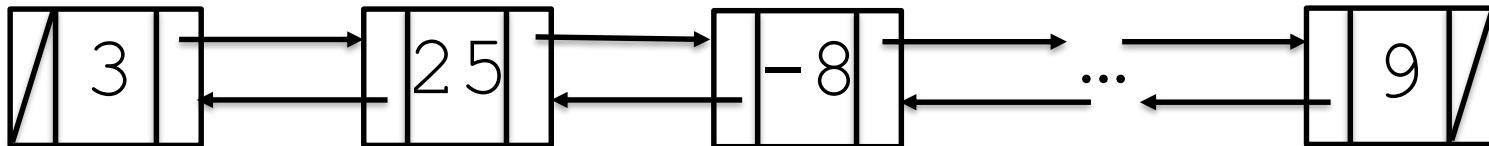
Implementaciones de `List<T>`

- `java.util` proporciona 3 implementaciones de `List<T>`, de las cuales veremos 2 (en realidad `ArrayList<T>` ya la hemos usado desde el Tema 2):

- `ArrayList<T>`
 - ✓ Array redimensionable dinámicamente.
 - ✓ Inserción y eliminación ineficientes.
 - ✓ Consultas rápidas.



- `LinkedList<T>`
 - ✓ Lista (doblemente) enlazada.
 - ✓ Inserción y eliminación eficientes,
 - ✓ Consultas ineficientes.



Implementaciones de List<T>

- `java.util` proporciona 3 implementaciones de List<T>, de las cuales veremos 2 (en realidad `ArrayList<T>` ya la hemos usado desde el Tema 2):
 - `ArrayList<T>`
 - ✓ Array redimensionable dinámicamente.
 - ✓ Inserción y eliminación ineficientes.
 - ✓ Consultas rápidas.
 - `LinkedList<T>`
 - ✓ Lista (doblemente) enlazada.
 - ✓ Inserción y eliminación eficientes,
 - ✓ Consultas ineficientes.
- Constructores:
 - Sin argumentos o con una colección como parámetro.
 - `ArrayList<T>` tiene un tercer constructor en el que se puede indicar la capacidad inicial.
- Representación como Cadena de Caracteres (`toString()`):
 - Igual que Arrays
[e1, e2, ..., en]

Ejemplo: uso de List<T>

Comparar 2 listas de cadenas y determinar cuántas cadenas coinciden en la misma posición

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

```
public class CompararListas {  
    public static void main(String args[]) {  
        // creamos la lista original a partir del array args  
        List<String> original = Arrays.asList(args);  
        // creamos una copia de la lista y la desordenamos  
        List<String> duplicado = new ArrayList<>(original);  
        Collections.shuffle(duplicado);  
        // comparamos las dos listas con sendos iteradores  
        Iterator<String> iterOriginal = original.iterator();  
        Iterator<String> iterDuplicado = duplicado.iterator();  
        int contMismoSitio = 0;  
        while (iterOriginal.hasNext()) {  
            if (iterOriginal.next().equals(iterDuplicado.next())) {  
                contMismoSitio++;  
            }  
        }  
        //mostramos el resultado en pantalla  
        System.out.println(duplicado + ": " + contMismoSitio + " en el mismo sitio.");  
    }  
}
```

iteradores

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

Ejemplo: uso de List<T>

Comparar 2 listas de cadenas y determinar cuántas cadenas coinciden en la misma posición

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

```
public class CompararListas {  
    public static void main(String args[]) {  
        // creamos la lista original a partir del array args  
        List<String> original = Arrays.asList(args);  
        // creamos una copia de la lista y la desordenamos  
        List<String> duplicado = new ArrayList<>(original);  
        Collections.shuffle(duplicado);  
        // comparamos las dos listas con índices  
        int contMismoSitio = 0;  
        for (int i = 0; i < original.size(); i++) {  
            if (original.get(i).equals(duplicado.get(i))) {  
                contMismoSitio++;  
            }  
        }  
        //mostramos el resultado en pantalla  
        System.out.println(duplicado + ": " + contMismoSitio + " en el mismo sitio.");  
    }  
}
```

} índices

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

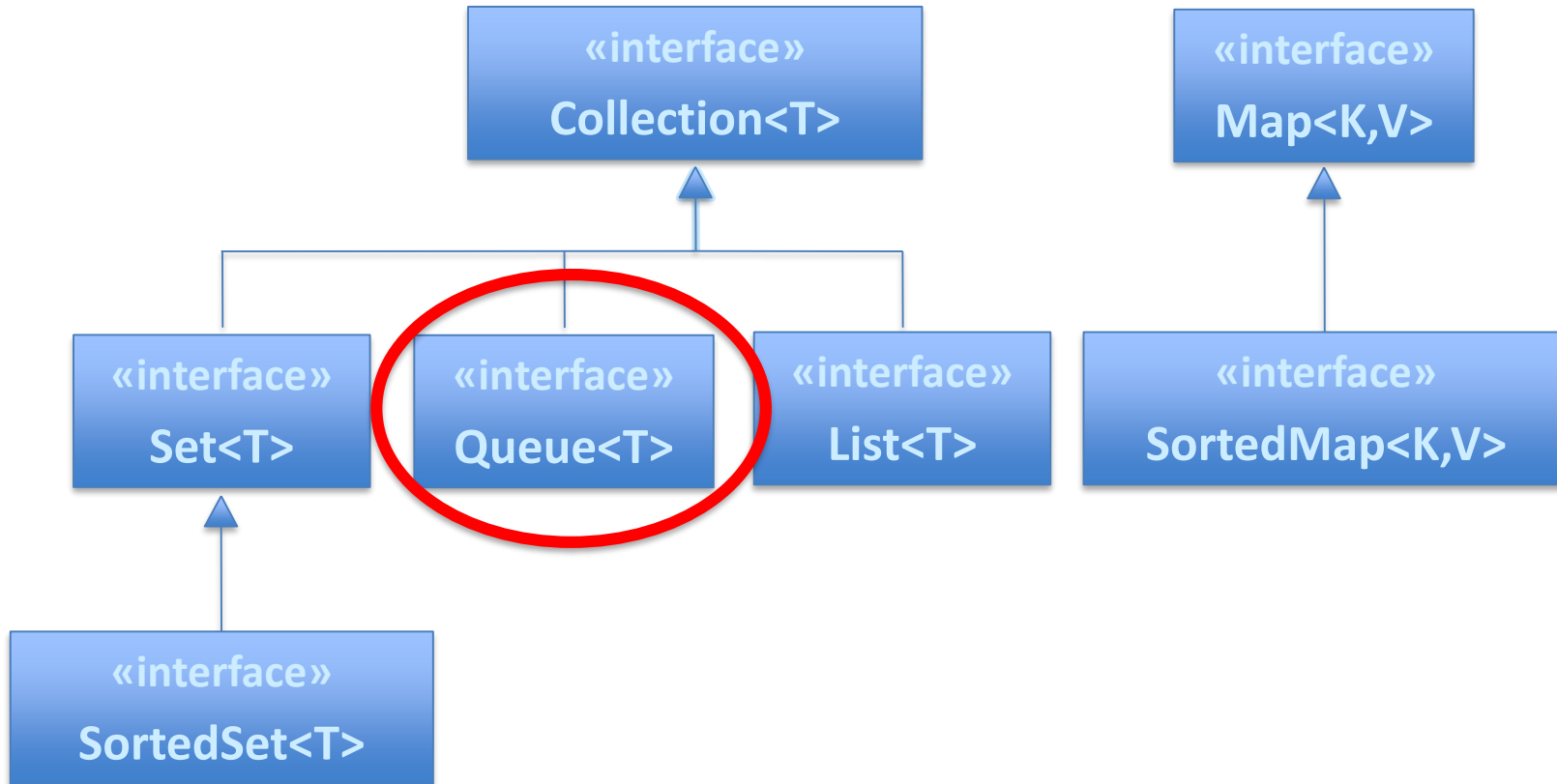
Contenido

- Clases e Interfaces genéricas
- Interfaces para objetos ordenables
- Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias



Interfaces básicas

(java.util)

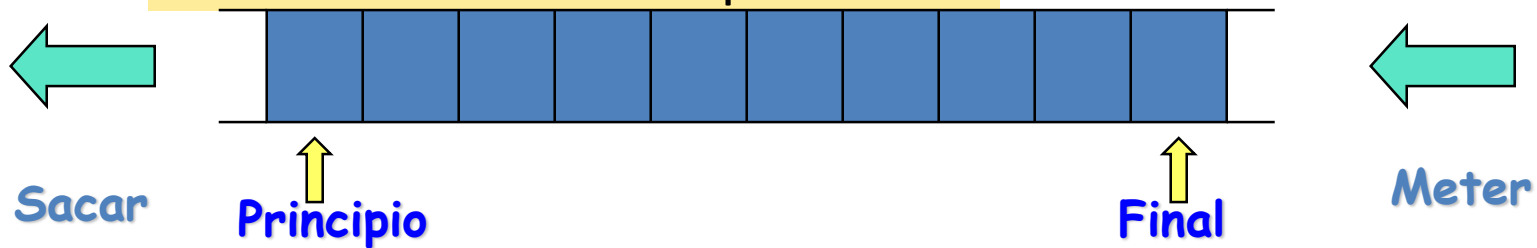


La interfaz Queue<T>

- La interfaz Queue<T> hereda de Collection<T>.

```
public interface Queue<T> extends Collection<T>
```

- Pensada para una colección de elementos que forman una secuencia a la que se puede acceder sólo por los dos extremos:
 - Se eliminan o consultan elementos por el principio
 - Se añaden elementos por el final



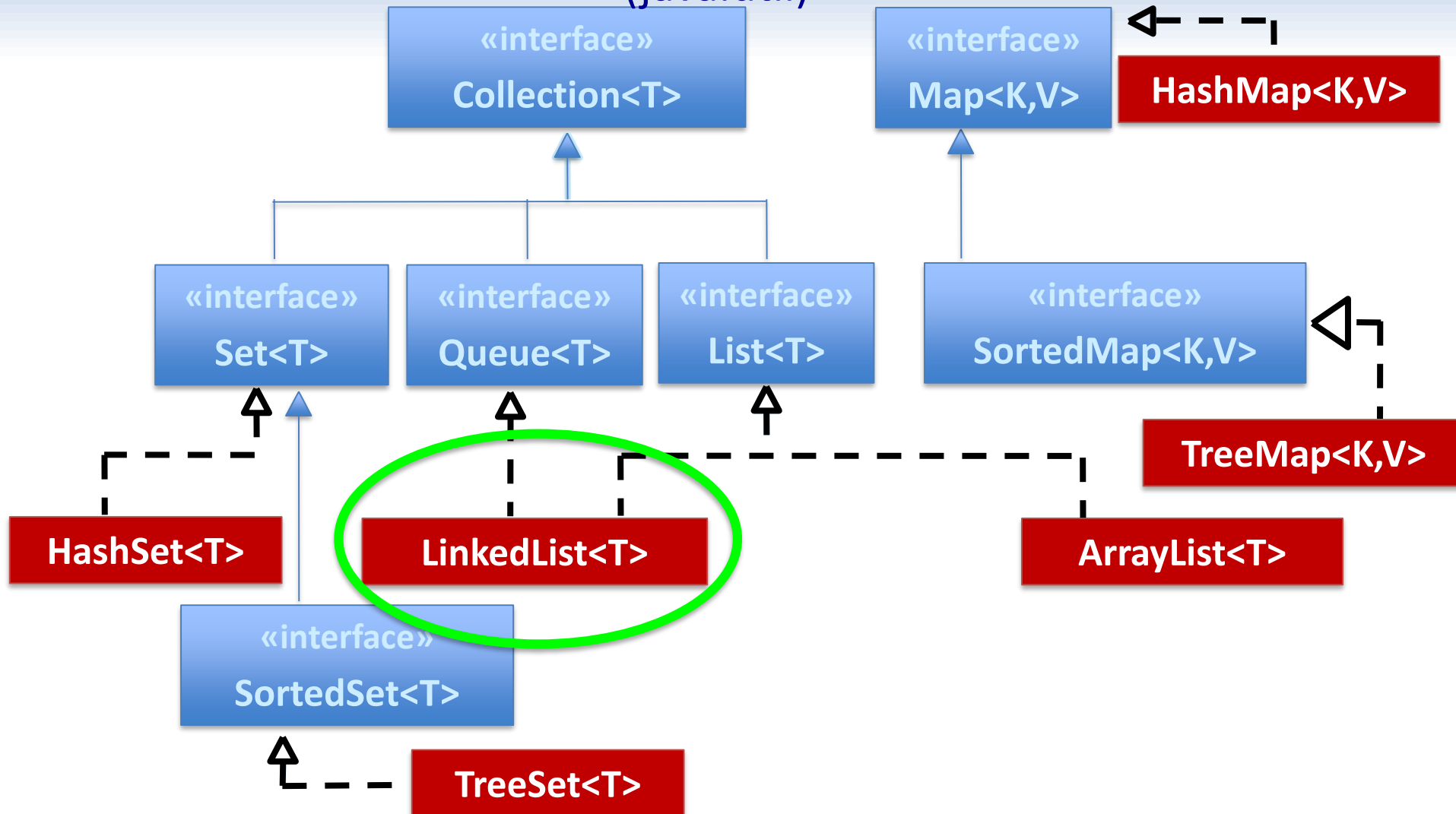
Estructura **FIFO** (First In First Out)

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  
    void add(T e);         // IllegalStateException si no cabe  
  
    boolean offer(T e);    // false si no cabe  
}
```

Interfaces básicas y sus implementaciones

(java.util)

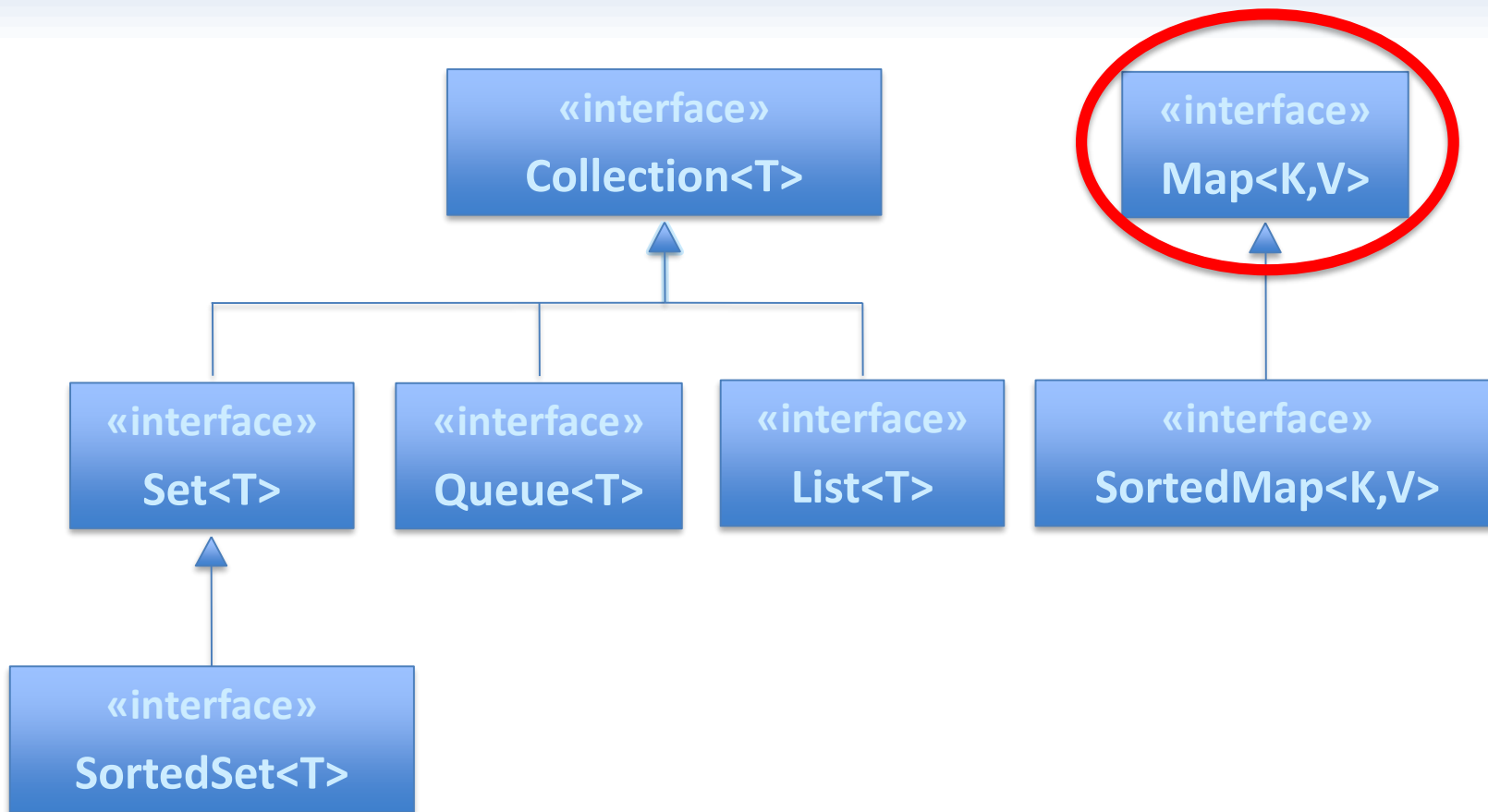


Contenido

- Clases e Interfaces genéricas
 - Interfaces para objetos ordenables
 - Colecciones y Correspondencias
 - Las interfaces básicas y sus implementaciones
 - Colecciones e Iteradores
 - Conjuntos, Listas y Colas
 - Correspondencias
- 

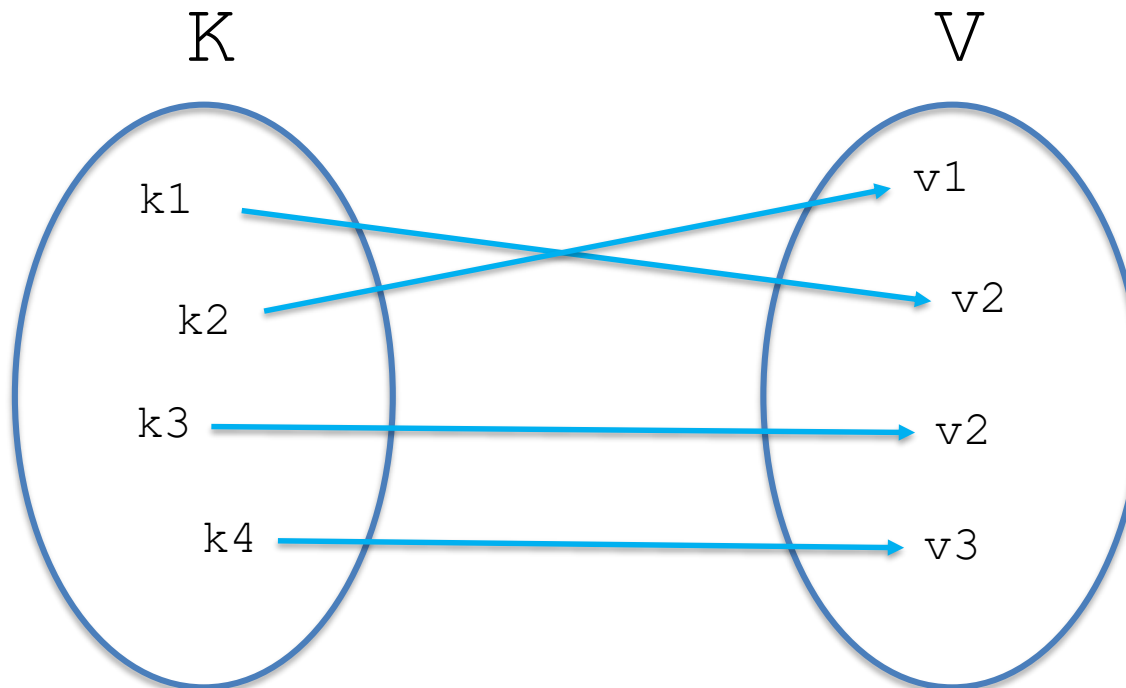
Interfaces básicas

(java.util)



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

- Map<K,V> define correspondencias (o asociaciones) de claves a valores.
 - Las claves son únicas (se controla con equals()).
 - Cada clave puede emparejarse con a lo sumo un valor.

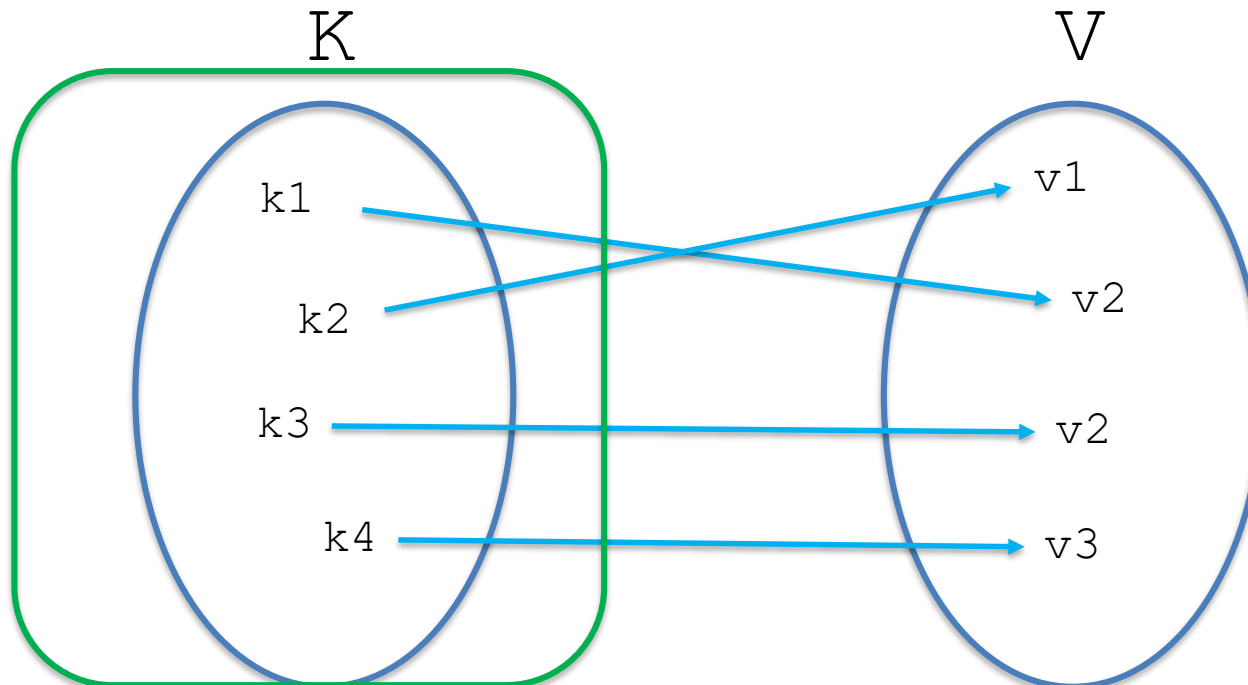


La interfaz `Map<K, V>`

- 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:

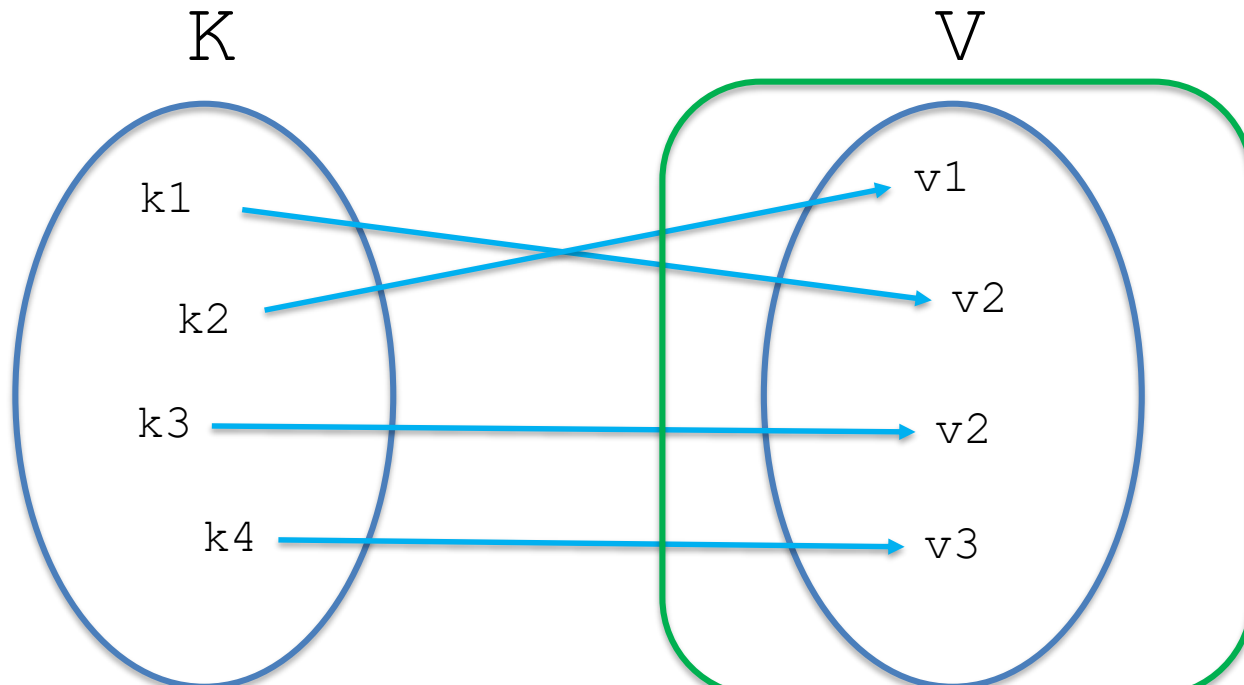
La interfaz $\text{Map}\langle K, V \rangle$

- Una correspondencia no es una colección, y por esto la interfaz $\text{Map}\langle K, V \rangle$ no hereda de $\text{Collection}\langle T \rangle$. Sin embargo, una correspondencia puede ser vista como una colección de varias formas:
 - un conjunto de claves
 - una colección de valores
 - un conjunto de pares <clave, valor>



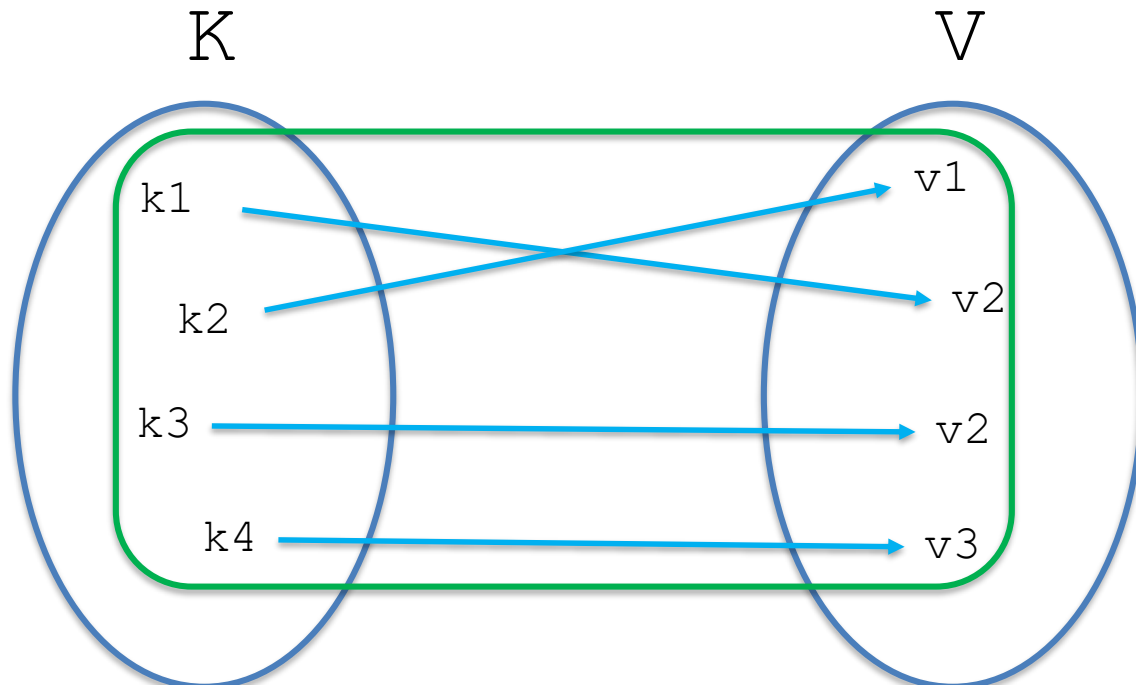
La interfaz $\text{Map}\langle K, V \rangle$

- Una correspondencia no es una colección, y por esto la interfaz $\text{Map}\langle K, V \rangle$ no hereda de $\text{Collection}\langle T \rangle$. Sin embargo, una correspondencia puede ser vista como una colección de varias formas:
 - un conjunto de claves
 - una colección de valores
 - un conjunto de pares $\langle \text{clave}, \text{valor} \rangle$



La interfaz $\text{Map}\langle K, V \rangle$

- Una correspondencia no es una colección, y por esto la interfaz $\text{Map}\langle K, V \rangle$ no hereda de $\text{Collection}\langle T \rangle$. Sin embargo, una correspondencia puede ser vista como una colección de varias formas:
 - un conjunto de claves
 - una colección de valores
 - un conjunto de pares <clave, valor>



La interfaz Map<K , V>

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

Crear una Correspondencia constante

El método `static 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 más de 10 parejas puede usarse el método ***ofEntries***:

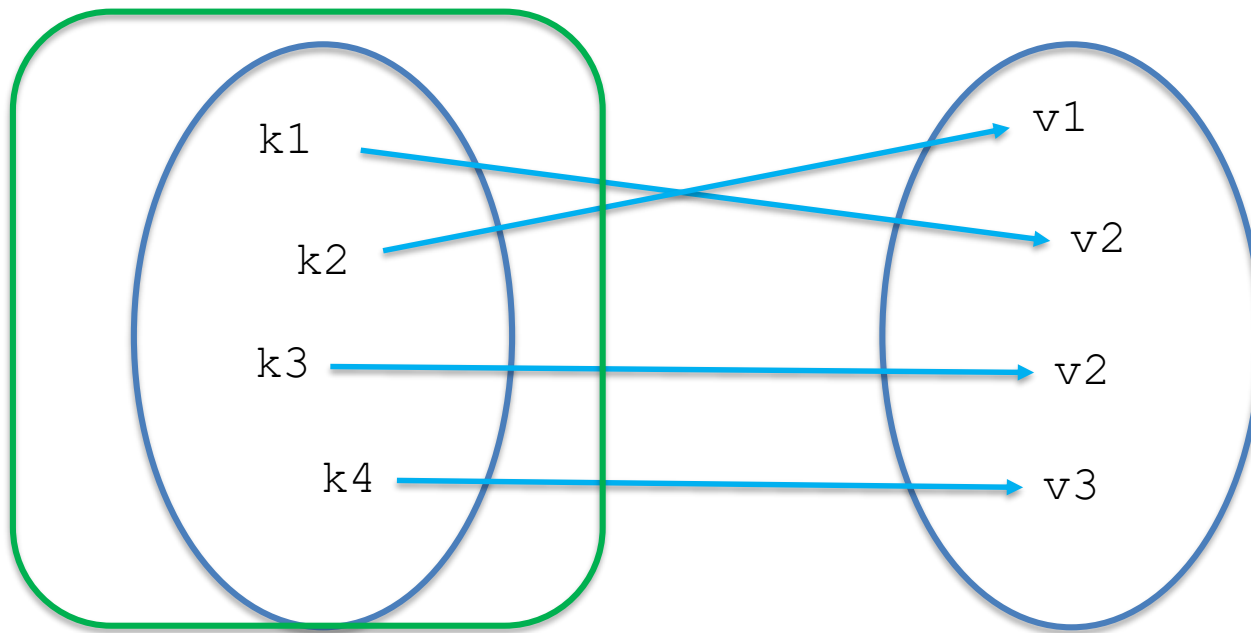
```
Map<String,Integer> map2 = Map.ofEntries( Map.entry("juan", 23),  
                                           Map.entry("Luis", 24),  
                                           Map.entry("maria",19));
```

Las correspondencias así creadas son inmutables.

Formas de recorrer una Correspondencia

– Recorrer el conjunto de claves

```
public Set<K> keySet();
```



Formas de recorrer una Correspondencia

Ejemplo: Recorrido del conjunto de claves

Calcular la media de las alturas de una serie de personas

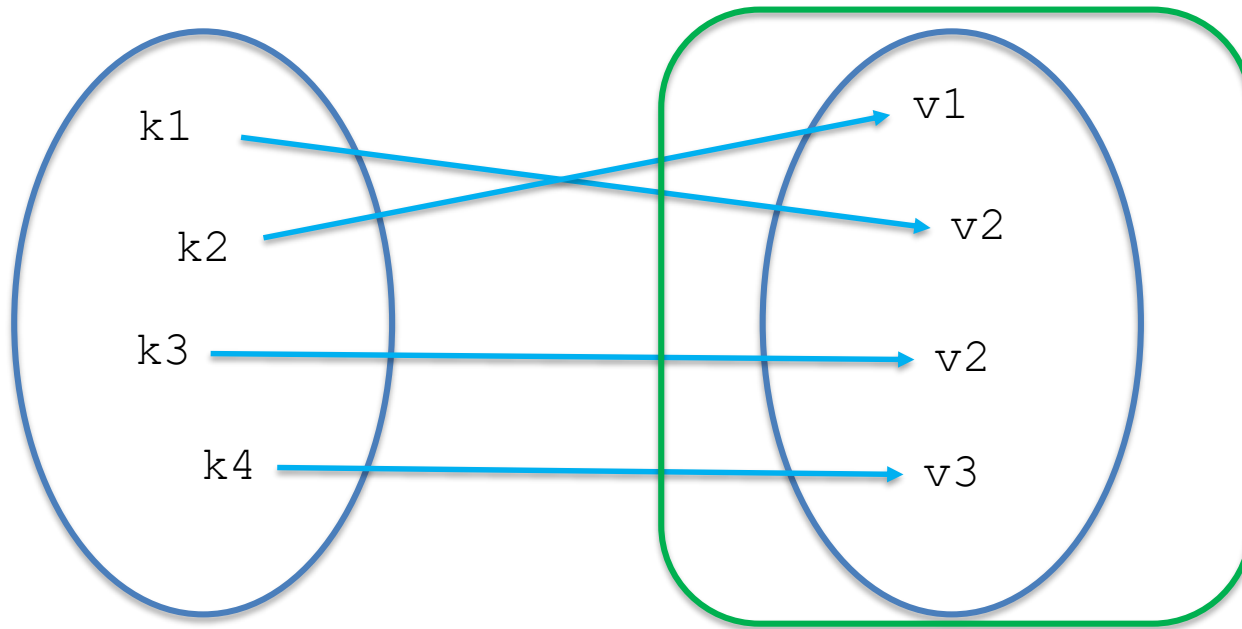
```
public static double mediaAltura(Map<String,Double>
                                alturaPersonas) {
    double res = 0, suma = 0;

    if (!alturaPersonas.isEmpty()) {
        for (String s : alturaPersonas.keySet()) {
            suma += alturaPersonas.get(s);
        }
        res = suma/alturaPersonas.size();
    }
    return res;
}
```

Formas de recorrer una Correspondencia

– Recorrer la colección de valores

```
public Collection<V> values();
```



Formas de recorrer una Correspondencia

Ejemplo: Recorrido de la colección de valores

Calcular la media de las alturas de una serie de personas

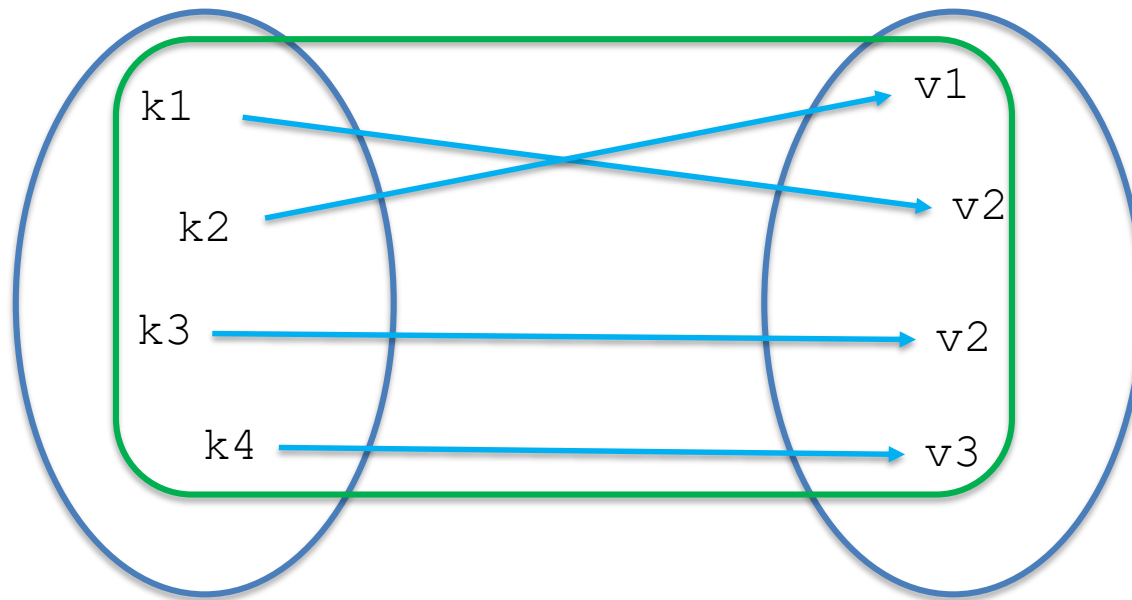
```
public static double mediaAltura(Map<String,Double>
                                alturaPersonas) {
    double res = 0, suma = 0;

    if (!alturaPersonas.isEmpty()) {
        for (Double d : alturaPersonas.values()) {
            suma += d;
        }
        res = suma/alturaPersonas.size();
    }
    return res;
}
```

Formas de recorrer una Correspondencia

- Recorrer el conjunto de pares <clave, valor>

```
public Set<Map.Entry<K,V>> entrySet();
```



Formas de recorrer una Correspondencia

Ejemplo: Recorrido del conjunto de pares <clave, valor>

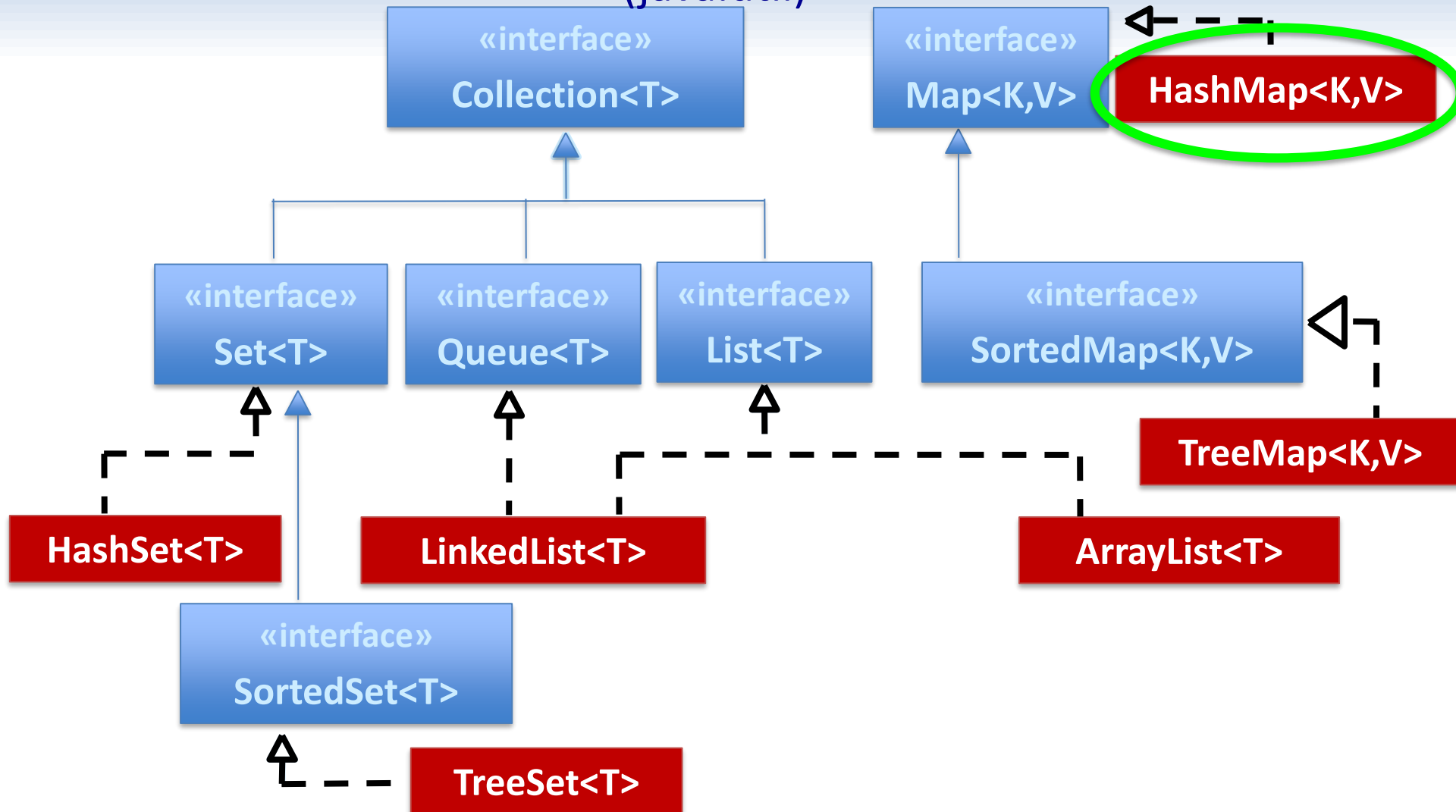
Calcular la media de las alturas de una serie de personas

```
public static double mediaAltura(Map<String,Double>
                                alturaPersonas) {
    double res = 0, suma = 0;

    if (!alturaPersonas.isEmpty()) {
        for (Map.Entry<String, Double> par : alturaPersonas.entrySet()) {
            suma += par.getValue();
        }
        res = suma/alturaPersonas.size();
    }
    return res;
}
```

Interfaces básicas y sus implementaciones

(java.util)



Implementación de Map<K, V>

`java.util` proporciona 2 implementaciones de Map<K,V>, de las cuales veremos 1:


- `HashMap<K,V>`
 - Utiliza una tabla hash
 - Constructores:
 - Sin argumentos
 - Con una asociación
 - Con capacidad y factor de carga.
 - Representación como Cadena de Caracteres (`toString()`):
`{k1=v1, k2=v2, ..., kn= vn}`

Ejemplo: frecuencias

HashMap<K, V>

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

```
public class Frecuencias {  
    public static void main(String[] args) {  
        Map<Integer, Integer> mFrecs = new HashMap<>();  
        int clave;  
        Integer frec;  
        for (String arg : args) {  
            // Incr. la frec. de cada número (clave)  
            clave = Integer.parseInt(arg);  
            frec = mFrecs.get(clave);  
            if (frec == null) {  
                mFrecs.put(clave, 1);  
            } else {  
                mFrecs.put(clave, frec + 1);  
            }  
        }  
        // Mostramos frecs. iterando sobre el conjunto de claves  
        for (Integer c : mFrecs.keySet()) {  
            frec = mFrecs.get(c);  
            System.out.print("\n" + c + ":\t");  
            for (int i = 0; i < frec; i++) {  
                System.out.print("*");  
            }  
        }  
    }  
}
```



```
frec = mFrecs.getOrDefault(clave, 0);  
mFrecs.put(clave, frec + 1);
```

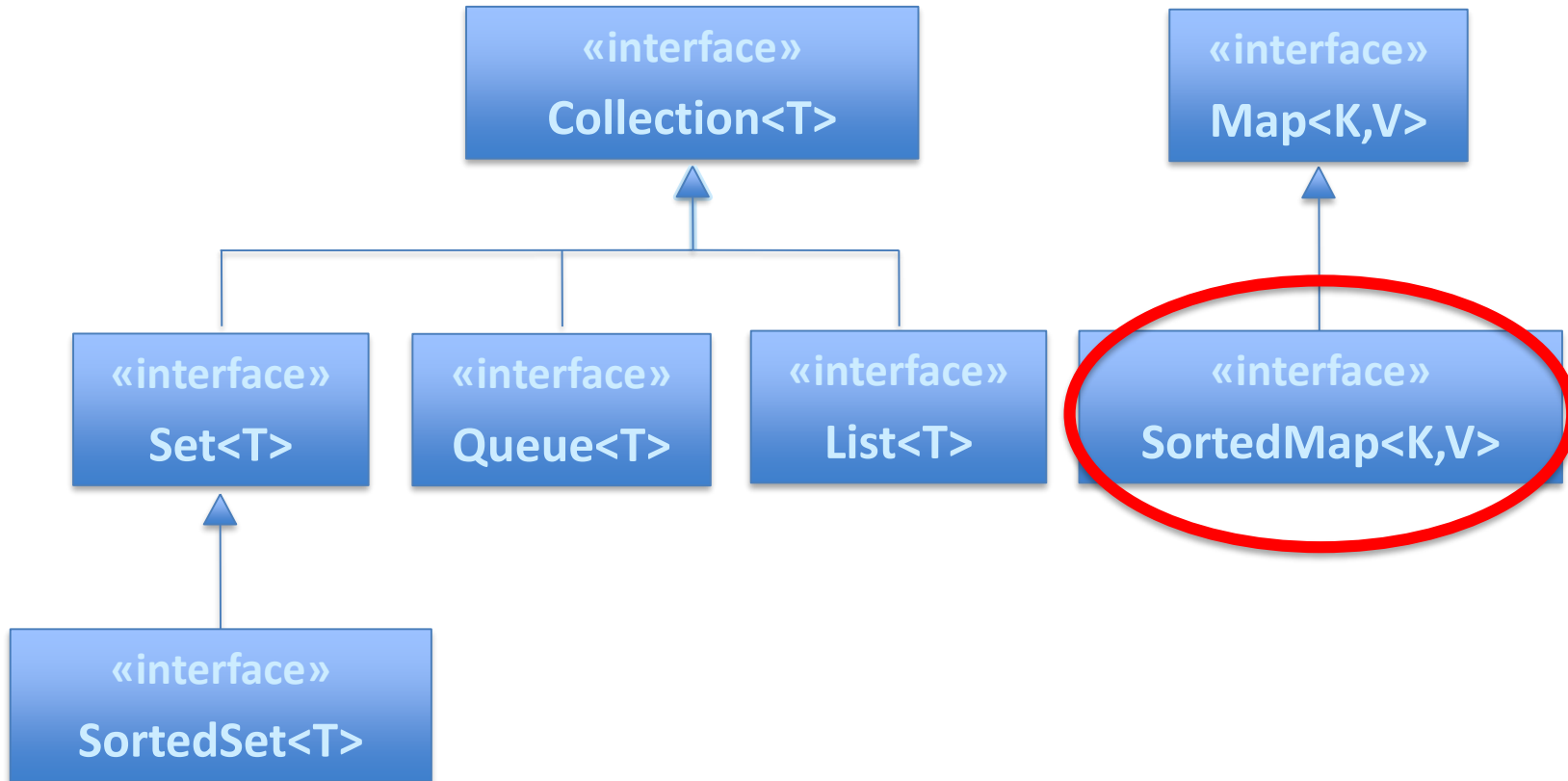
ARGUMENTOS: 5 4 32 3 4 3 2 3 4 2 5 2 3

SALIDA:

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


Interfaces básicas

(java.util)



La interfaz SortedMap<K, V>

- Extiende Map<K,V> para proporcionar la funcionalidad adecuada para correspondencias con claves ordenadas.
- El orden utilizado para ordenar las claves es:
 - Por defecto el *orden natural* (Comparable<K>)
 - El *orden alternativo* especificado por un Comparator<K> en el constructor de la clase que implemente esta interfaz

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 primera y última
```

```
    T firstKey();
```

```
    T lastKey();
```

```
    ...
```

```
}
```

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);
```

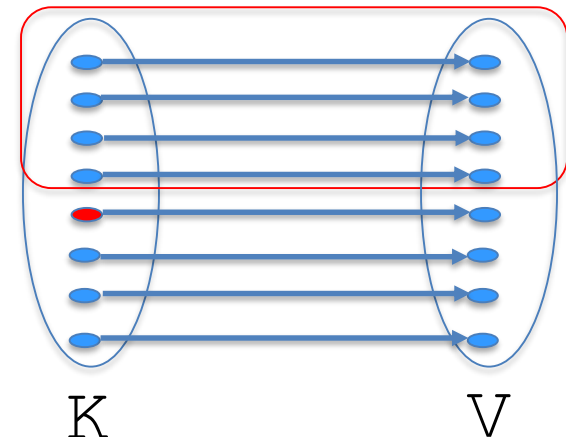
```
// claves primera y última
```

```
T firstKey();
```

```
T lastKey();
```

```
...
```

```
}
```



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);
```

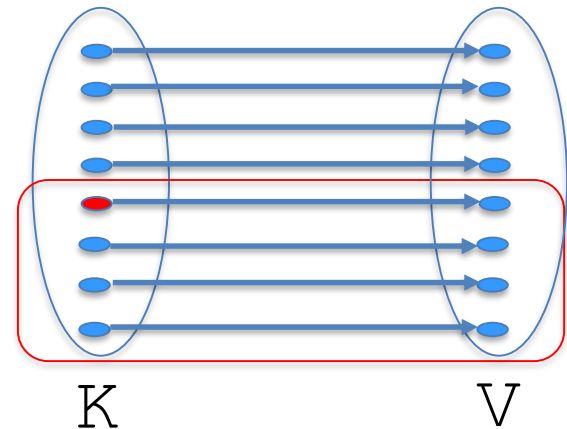
```
// claves primera y última
```

```
T firstKey();
```

```
T lastKey();
```

```
...
```

```
}
```



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);
```

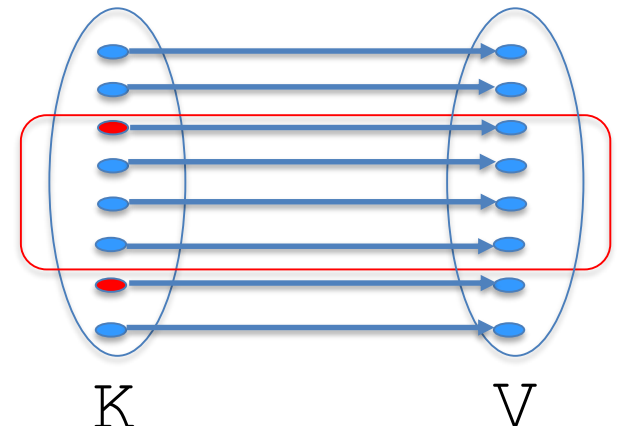
```
// claves primera y última
```

```
T firstKey();
```

```
T lastKey();
```

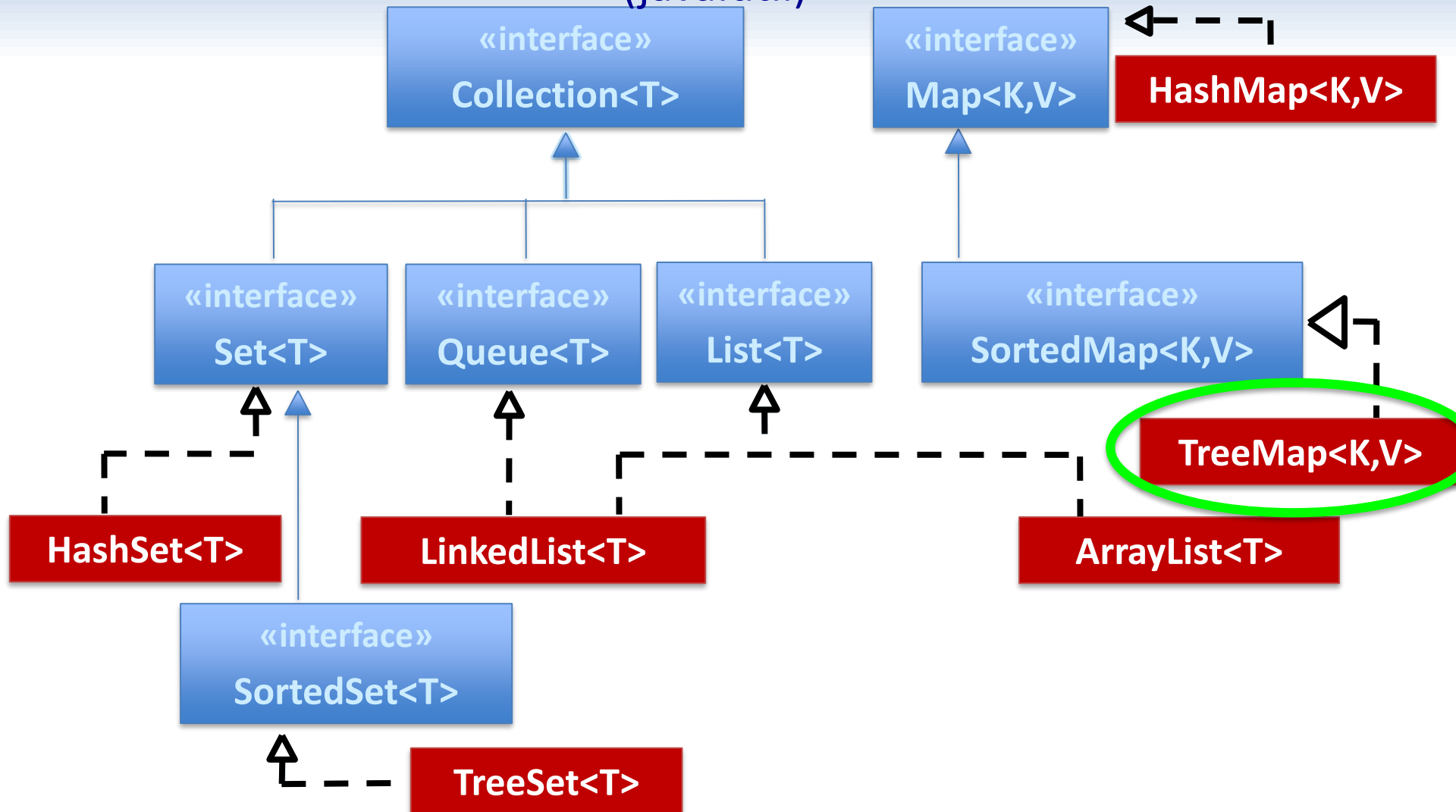
```
...
```

```
}
```



Interfaces básicas y sus implementaciones

(java.util)



Implementación de SortedMap<K, V>

- `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 alternativo
 - `TreeMap(Map<? extends K,? extends V> c)` // Orden natural
 - `TreeMap(SortedMap<K,? extends V> s)` // Mismo orden que s
- Representación como Cadena de Caracteres (`toString()`):
`{k1=v1, k2=v2, ..., kn= vn}`

Recordemos Ejemplo: frecuencias

HashMap<K, V>

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

```
public class Frecuencias {  
    public static void main(String[] args) {  
        Map<Integer, Integer> mFrecs = new HashMap<>();  
        int clave;  
        Integer frec;  
        for (String arg : args) {  
            // Incr. la frec. de cada número (clave)  
            clave = Integer.parseInt(arg);  
            frec = mFrecs.get(clave);  
            if (frec == null) {  
                mFrecs.put(clave, 1);  
            } else {  
                mFrecs.put(clave, frec + 1);  
            }  
        }  
        // Mostramos frecs. iterando sobre el conjunto de claves  
        for (Integer c : mFrecs.keySet()) {  
            frec = mFrecs.get(c);  
            System.out.print("\n" + c + ":\t");  
            for (int i = 0; i < frec; i++) {  
                System.out.print("*");  
            }  
        }  
    }  
}
```



```
frec = mFrecs.getDefault(clave, 0);  
mFrecs.put(clave, frec + 1);
```

ARGUMENTOS: 5 4 32 3 4 3 2 3 4 2 5 2 3

SALIDA:

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

Ahora Ejemplo: frecuencias

TreeMap<K, V>

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

```
public class Frecuencias {
```

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

```
        Map<Integer, Integer> mFrecs = new TreeMap<>();
```

```
        int clave;
```

```
        Integer frec;
```

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

```
            // Incr. la frec. de cada número (clave)
```

```
            clave = Integer.parseInt(arg);
```

```
            frec = mFrecs.get(clave);
```

```
            if (frec == null) {
```

```
                mFrecs.put(clave, 1);
```

```
            } else {
```

```
                mFrecs.put(clave, frec + 1);
```

```
            }
```

```
        }
```

```
        // Mostramos frecs. iterando sobre el conjunto de claves
```

```
        for (Integer c : mFrecs.keySet()) {
```

```
            frec = mFrecs.get(c);
```

```
            System.out.print("\n" + c + ":\t");
```

```
            for (int i = 0; i < frec; i++) {
```

```
                System.out.print("*");
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

frec = mFrecs.getDefault(clave, 0);

mFrecs.put(clave, frec + 1);

ARGUMENTOS: 5 4 32 3 4 3 2 3 4 2 5 2 3

SALIDA:

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

Ejemplo: frecuencias

TreeMap<K, V>

(subMap)

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

```
public class Frecuencias {
```

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

```
        SortedMap<Integer,Integer> mFrecs = new TreeMap<>();
```

```
        int clave;
```

```
        Integer frec
```

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

```
            // Incr. la frec. de cada número (clave)
```

```
            clave = Integer.parseInt(arg);
```

```
            frec = mFrecs.get(clave);
```

```
            if (frec == null) {
```

```
                mFrecs.put(clave, 1);
```

```
            } else {
```

```
                mFrecs.put(clave, frec + 1);
```

```
            }
```

```
        }
```

```
        // Muestra frecs. de subrango iterando sobre el conjunto de claves
```

```
        SortedMap<Integer,Integer> subFrecs = mFrecs.subMap(1, 5);
```

```
        for (Integer c : subFrecs.keySet()) {
```

```
            frec = subFrecs.get(c);
```

```
            System.out.print("\n" + c + ":\t");
```

```
            for (int i = 0; i < frec; i++) {
```

```
                System.out.print("*");
```

```
            }
```

```
        }
```

```
    }
```

frec = mFrecs.getDefault(clave,0);
mFrecs.put(clave, frec + 1);

ARGUMENTOS: 5 4 32 3 4 3 2 3 4 2 5 2 3

SALIDA:

```
2:   ***
3:   ****
4:   ***
```

Ejemplo: Mostrar posiciones

TreeMap<K, V>

```
import java.util.*;

public class Posiciones{
    public static void main(String[] args) {
        SortedMap<Integer,List<Integer>> mPos = new TreeMap<>();
        int num;
        List<Integer> lPos;
        for (int i = 0; i < args.length; i++) {
            num = Integer.parseInt(args[i]);
            // Buscamos la lista asociada a num en mPos
            lPos= mPos.get(num);
            if (lPos == null) {
                lPos = new ArrayList<>();          // 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);
        }
        // Muestra posiciones iterando sobre conj. ordenado de pares
        for (Map.Entry<Integer,List<Integer>> par : mPos.entrySet()) {
            System.out.println(par.getKey() + ":\t" + par.getValue());
        }
    }
}
```

ARGUMENTOS: 5 4 32 3 4 3 2 3 4 2 5 2 3

SALIDA:

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