

## Tema 5. Colecciones e iteradores

Dpto. Lenguajes y Ciencias de la Computación. E.T.S.I. Informática.  
Universidad de Málaga

Programación Orientada a Objetos

## Tema 5. Colecciones e iteradores

- Clases genéricas.
- Clases ordenables, orden natural y órdenes alternativos.
- Colecciones y correspondencias (asociaciones).
  - Colecciones.
  - Iteradores.
  - Conjuntos.
  - Listas e Iteradores sobre listas.
  - Colas.
  - Correspondencias (asociaciones).
- Creación de Colecciones y Correspondencias constantes.
- Algoritmos sobre Listas y Colecciones. La clase Collections.
- Algoritmos sobre arrays. La clase Arrays.

Esta obra se encuentra bajo una licencia Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) de Creative Commons.



# Clases genéricas

- Las **clases genéricas** permiten definir clases e interfaces que no dependan de clases concretas, sino que sus comportamientos puedan ser aplicados a cualquier clase o interfaz en general (**tipos genéricos parametrizados**).
- Las **clases genéricas** permiten expresar en **una única definición** comportamientos comunes para objetos pertenecientes a **distintas clases**.
- Un ejemplo de clase genérica puede ser una clase **contenedora** que contiene una *colección de elementos*, donde los elementos pueden ser de *cualquier clase*.
- Una clase o interfaz puede especificar **tipos genéricos parametrizados** en su definición, que siempre representan clases o interfaces.
  - A la hora de **instanciar** la clase genérica, se debe especificar el **tipo concreto** de los **parámetros**. Éste debe ser una **clase o interfaz**, nunca un tipo primitivo.
  - Una clase genérica puede instanciarse tantas veces como sea necesario, y los tipos concretos de los parametros utilizados pueden variar en cada instanciación.
  - En la definición, se pueden especificar **restricciones** sobre los parámetros formales, que se deberán satisfacer por los tipos concretos en la instanciación.

# Clases genéricas. Un ejemplo simple (I)

- Supongamos que queremos definir una clase (genérica) que esté compuesta por dos elementos de *cualquier* otra clase (tipo genérico parametrizado).
  - Denominaremos T al tipo genérico parametrizado, donde T representa a cualquier clase o interfaz. Se debe especificar <T> en la definición de la clase.

```
public class Dos<T> {  
    private T primero, segundo;  
    public Dos(T p, T s) {  
        primero = p;  
        segundo = s;  
    }  
    public void setPrimero(T p) {  
        primero = p;  
    }  
    public void setSegundo(T s) {  
        segundo = s;  
    }  
    public T getPrimero() {  
        return primero;  
    }  
    public T getSegundo() {  
        return segundo;  
    }  
    @Override  
    public String toString() {  
        return "(" + primero.toString() + ", " + segundo.toString() + ")";  
    }  
}
```

Las **clases genéricas** permiten definir clases e interfaces que no dependan de clases concretas, sino que sus comportamientos puedan ser aplicados a cualquier clase o interfaz en general (**tipos genéricos parametrizados**).

## Nombres habituales de tipos genéricos parametrizados:

- T cualquier tipo.
- E cualquier tipo de elemento.
- N cualquier tipo numérico.
- K cualquier tipo de clave (*key*).
- V cualquier tipo de valor.

# Clases genéricas. Un ejemplo simple (II)

- Para crear objetos de una determinada clase genérica, debemos **instanciar** los tipos genéricos parametrizados, proporcionando los tipos concretos.

```
public class Programa {  
    public static void main(String[] args) {  
        Dos<String> conC = new Dos<String>("hola", "adios");  
        Dos<Integer> conI = new Dos<Integer>(4, 9);  
        Dos<Integer> conI2 = new Dos<>(4, 9);           // Inferencia de Tipos  
        // ...  
        String v1 = conC.getPrimero();                 // v1 = "hola"  
        String v2 = conC.getSegundo();                 // v2 = "adios"  
        int v3 = conI.getPrimero();                   // v3 = 4  
        int v4 = conI.getSegundo();                   // v4 = 9  
        conI.setPrimero(5);  
        conI.setSegundo(7);  
        System.out.println(conC.toString());           // "(hola, adios)"  
    }  
}
```

# Clases genéricas y herencia

- Una clase puede definirse genérica relacionando sus tipos genéricos parametrizados con los que tuviese su superclase o alguna interfaz que implemente.
- Por ejemplo, la clase `DosPeso` tiene el mismo parámetro genérico que la clase `Dos` de la que hereda.

```
public class DosPeso<T> extends Dos<T> {  
    int pesoPrimero;  
    int pesoSegundo;  
    public DosPeso(T p, int pp, T s, int ps) {  
        super(p, s);  
        pesoPrimero = pp;  
        pesoSegundo = ps;  
    }  
    // ...  
}  
  
DosPeso<String> dosp = new DosPeso<String>("hola", 112, "adios", 65);
```

- Aunque la clase `D` sea subclase de `B`, la clase `F<D>` **no** es subclase de `F<B>`.
- Ejemplo: La clase `String` es subclase de `Object` pero la clase `Dos<String>` **no** es subclase de `Dos<Object>`.

# Restricciones en las clases genéricas

- Podemos imponer **restricciones** a los tipos concretos que puede tomar un tipo genérico parametrizado:

- Que sean de una **subclase** de una clase dada (**extends**).
- Que **implementen** una o varias interfaces (**extends**).

```
public class DosNumeros<T extends Number> {  
    private T primero, segundo;  
    // ...  
    public double getValorPrimero() {  
        return primero.doubleValue(); // Invocación a método de Number  
    }  
}  
  
// Number es una clase de la que heredan los envoltorios numéricos  
DosNumeros<Integer> p1 = new DosNumeros<Integer>(10, 15);  
DosNumeros<Double> p2 = new DosNumeros<Double>(10.5, 15.3);  
DosNumeros<String> q = new DosNumeros<String>("Pepe", "Ana"); // ERROR
```

- Si un tipo genérico parametrizado está restringido, entonces se pueden **invocar a los métodos** definidos por la restricción sobre los objetos del tipo genérico.
- La forma general de definir una restricción sobre el parámetro de una clase genérica es (se debe especificar la clase primero):

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

# Clases con más de un tipo genérico parametrizado

- Se pueden especificar más un tipo genérico parametrizado en la definición de una clase o interfaz genérica.

```
public class Par<A, B> {  
    private A primero;  
    private B segundo;  
    public Par(A a, B b) {  
        primero = a; segundo = b;  
    }  
    public void setPrimero(A a) {  
        primero = a;  
    }  
    public void setSegundo(B b) {  
        segundo = b;  
    }  
    public A getPrimero() {  
        return primero;  
    }  
    public B getSegundo() {  
        return segundo;  
    }  
}  
  
// Par<String, Integer> p = new Par<String,Integer>("hola", 10);  
// Par<String, Integer> p = new Par<>("hola", 10);    // Inferencia de Tipos
```



# Métodos genéricos

- También se pueden especificar **tipos genéricos parametrizados** en la definición de un determinado **método**.
  - Los tipos genéricos parametrizados se especifican **antes** del tipo del método.
  - Supongamos una clase no genérica con el siguiente método genérico:

```
public class Programa {  
    public static <A,B> String aCadena(Par<A, B> par) {  
        return "(" + par.getPrimero() + "," + par.getSegundo() + ")";  
    }  
}  
  
// Par<String, Integer> p = new Par<String,Integer>("hola", 10);  
// System.out.println(Programa.<String,Integer>aCadena(p));  
// System.out.println(Programa.aCadena(p));           // Inferencia de Tipos
```

## Tipos parametrizados anónimos

- Cuando un tipo genérico parametrizado no se utiliza en el cuerpo del método, entonces se puede utilizar el símbolo ?.

```
public class Programa {  
    public static String aCadena(Par<?, ?> par) {  
        return "(" + par.getPrimero() + "," + par.getSegundo() + ")";  
    }  
}
```

# Restricciones sobre los tipos parametrizados anónimos (I)

- Problema: Supongamos la siguiente clase con un método genérico:

```
public class Coche { /* ... */ }
public class CocheImportado extends Coche { /* ... */ }
public class Programa {
    public static <T> void copiaPrimero(Dos<T> orig, Dos<T> dest) {
        dest.setPrimero(orig.getPrimero());
    }
    public static void main(String[] args) {
        CocheImportado ip = new CocheImportado("Porsche", 50000, 3000);
        CocheImportado is = new CocheImportado("Mazda", 40000, 2500);
        Dos<CocheImportado> dosI = new Dos<CocheImportado>(ip, is);
        Coche cp = new Coche("Seat", 14000);
        Coche cs = new Coche("Renault", 18000);
        Dos<Coche> dosC = new Dos<Coche>(cp, cs);
        Programa.copiaPrimero(dosI, dosC);
    }
}
```

- El método `copiaPrimero(Dos<T>, Dos<T>)` no es aplicable en la forma `copiaPrimero(Dos<CocheImportado>, Dos<Coche>)`.
- El código no compila.

# Restricciones sobre los tipos parametrizados anónimos (II)

- Sobre un parámetro anónimo de un método se pueden especificar restricciones:
  - La clase anónima debe ser **superclase** de una clase dada (**super**).
  - La clase anónima debe ser **subclase** de una clase dada (**extends**).

```
public class Coche { /* ... */ }
public class CocheImportado extends Coche { /* ... */ }
public class Programa {
    public static <T> void copiaPrimero(Dos<? extends T> orig,
                                       Dos<? super T> dest) {
        dest.setPrimero(orig.getPrimero());
    }
    public static void main(String[] args) {
        CocheImportado ip = new CocheImportado("Porche", 50000, 3000);
        CocheImportado is = new CocheImportado("Mazda", 40000, 2500);
        Dos<CocheImportado> dosI = new Dos<CocheImportado>(ip, is);
        Coche cp = new Coche("Seat", 14000);
        Coche cs = new Coche("Renault", 18000);
        Dos<Coche> dosC = new Dos<Coche>(cp, cs);
        Programa.copiaPrimero(dosI, dosC);
    }
}
```

# Limitaciones sobre los tipos genéricos parametrizados

- 1 No se pueden instanciar tipos genéricos parametrizados con tipos primitivos.  
`Par<char, int> p1 = new Par<char, int>('a', 10);` *// ERROR*  
`Par<Character, Integer> p2 = new Par<Character,Integer>('a', 10);` *// CORRECTO*
- 2 No se pueden crear instancias (objetos) de tipos genéricos parametrizados.
- 3 No se pueden crear *arrays* de tipos genéricos parametrizados.
- 4 No se pueden definir variables de clase (estáticas) de tipos genéricos parametrizados.
- 5 No se puede utilizar `instanceof` de tipos genéricos parametrizados (sí con ?).
- 6 No se pueden realizar *castings* de tipos genéricos parametrizados (sí con ?).
- 7 No se pueden definir, ni capturar, ni lanzar excepciones de tipos genéricos parametrizados.

```
public class Dato<T> {  
    private static T x;           // ERROR variable de clase (static) de tipo genérico  
    private T valor;  
    private T[] array;  
    public Dato() {  
        valor = new T();          // ERROR creación de objeto de tipo genérico  
        array = new T[16];        // ERROR creación de array de tipo genérico  
    }  
    public boolean equals(Object o) {  
        boolean ok = false;  
        if (o instanceof Dato<T>) {           // ERROR instanceof de tipo genérico  
            Dato<T> otro = (Dato<T>)o;        // ERROR casting de tipo genérico  
            ok = this.valor.equals(otro.valor);  
        }  
        return ok;  
    }  
}
```

# Clases ordenables. Orden natural y órdenes alternativos

El **orden natural** define el *mecanismo por defecto* para ordenar objetos.

- Hace que los objetos, en sí mismos, sean comparables y ordenables.
- La interfaz `Comparable<T>` permite definir el **orden natural**.
- El orden natural se define en la propia clase ordenable (comparable).

```
public class Persona implements Comparable<Persona> {  
    public int compareTo(Persona otra) { /* ... */ }           // @Override  
}
```

Los **órdenes alternativos** definen *mecanismos alternativos* para ordenar objetos.

- Utiliza objetos auxiliares para comparar otros objetos.
- La interfaz `Comparator<T>` permite definir un **orden alternativo**.
- Cada uno de los órdenes alternativos debe implementarse en una clase aparte diferente (*clase "satélite"*), que proporcione esa funcionalidad.

```
public class OrdenAlt1 implements Comparator<Persona> {  
    public int compare(Persona p1, Persona p2) { /* ... */ }    // @Override  
}  
  
public class OrdenAlt2 implements Comparator<Persona> {  
    public int compare(Persona p1, Persona p2) { /* ... */ }    // @Override  
}
```

Sólo es posible definir un **único** orden natural para una determinada clase, aunque se pueden definir **varios** órdenes alternativos.

# Orden natural. La interfaz java.lang.Comparable<T>

La interfaz `Comparable<T>` permite definir el **orden natural** para una clase.

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

Cuando una clase `T` proporciona el **Orden Natural**, entonces la clase `T` debe implementar la interfaz `Comparable<T>` y definir el método `compareTo`.

- El método `o1.compareTo(o2)` debe devolver:
  - **negativo** si `o1` es menor que `o2`.
  - **cero** si `o1` es igual a `o2` (Si `o1.equals(o2)` es `true`, entonces `o1.compareTo(o2)` devuelve `cero`).
  - **positivo** si `o1` es mayor que `o2`.
  - Atención a las comparaciones *sin diferenciar mayúsculas de minúsculas* (`IgnoreCase`).
  - Atención al orden (ascendente o descendente) en las comparaciones de los componentes.

```
public class Persona implements Comparable<Persona> {  
    // Compara la edad ascendente, la nota descendente, el nombre ascendente-IgnoreCase  
    public int compareTo(Persona other) {    // El tipo del parámetro es Persona  
        int resultado = Integer.compare(this.edad, other.edad); // Ascendente  
        if (resultado == 0) {  
            resultado = Double.compare(other.nota, this.nota); // Descendente  
            if (resultado == 0) {  
                resultado = this.nombre.compareToIgnoreCase(other.nombre); // Ascendente  
            }  
        }  
        return resultado;  
    }  
}
```

# La interfaz `java.lang.Comparable<T>` en la API de Java

La clase `String` implementa la interfaz `Comparable<T>`, y proporciona los *métodos de instancia* `compareTo(String)` (orden *natural lexicográfico*) y `compareToIgnoreCase(String)`.

Las clases *envoltorios* (`Character`, `Boolean`, `Integer`, `Double`, etc) implementan la interfaz `Comparable<T>`, y proporcionan el *método de instancia* `compareTo(T)`.

Además, las clases *envoltorios* también proporcionan el *método de clase* `compare()`, que permiten comparar los tipos primitivos:

- ▶ `int Character.compare(char a, char b);`
- ▶ `int Boolean.compare(char a, char b);`
- ▶ `int Integer.compare(int a, int b);`
- ▶ `int Double.compare(double a, double b);`

Las enumeraciones (`enum`) también implementan (proporcionada automáticamente por el lenguaje Java) la interfaz `Comparable<T>`, y proporcionan el *método de instancia* `compareTo(T)`.

# Ejemplo 1: 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; }
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona)o;
            ok = (this.edad == other.edad)&&(this.nombre.equals(other.nombre));
        }
        return ok;
    }
    public int hashCode() {
        return Integer.hashCode(this.edad) + this.nombre.hashCode();
    }
    public int compareTo(Persona other) {
        // Comparación por edad ascendente, y a igualdad de edad, por nombre ascendente
        int resultado = Integer.compare(this.edad, other.edad);
        if (resultado == 0) {
            resultado = this.nombre.compareTo(other.nombre);
        }
        return resultado;
    }
}
```

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



## Ejemplo 2: clase Persona (*IgnoreCase*)

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String nombre() { return nombre; }
    public int edad() { return edad; }
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona)o;
            ok = (this.edad == other.edad) && (this.nombre.equalsIgnoreCase(other.nombre));
        }
        return ok;
    }

    public int hashCode() {
        return Integer.hashCode(this.edad) + this.nombre.toLowerCase().hashCode();
    }

    public int compareTo(Persona other) {
        // Comparación por edad ascendente, y a igualdad de edad, por nombre ascendente-IgnoreCase
        int resultado = Integer.compare(this.edad, other.edad);
        if (resultado == 0) {
            resultado = this.nombre.compareToIgnoreCase(other.nombre);
        }
        return resultado;
    }
}
```

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

## Ejemplo 3: clase Persona (descendente)

```
public class Persona implements Comparable<Persona> {
    private String nombre;
    private int edad;
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
    public String nombre() { return nombre; }
    public int edad() { return edad; }
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona)o;
            ok = (this.edad == other.edad) && (this.nombre.equals(other.nombre);
        }
        return ok;
    }
    public int hashCode() {
        return Integer.hashCode(this.edad) + this.nombre.hashCode();
    }
    public int compareTo(Persona other) {
        // Comparación por edad descendente, y a igualdad de edad, por nombre descendente
        // El orden en el que se comparan los componentes es importante
        int resultado = Integer.compare(other.edad, this.edad);
        if (resultado == 0) {
            resultado = other.nombre.compareTo(this.nombre);
        }
        return resultado;
    }
}
```

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

## Orden alternativo. La interfaz `java.util.Comparator<T>`

La interfaz `Comparator<T>` permite definir un **orden alternativo** para una clase.

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

Cuando una **clase satélite** proporciona un **Orden Alternativo** sobre otra clase `T`, entonces la clase satélite debe implementar la interfaz `Comparator<T>`, y definir el método `compare`.

- El método `sat.compare(o1, o2)` debe devolver:
  - **negativo** si `o1` es menor que `o2`.
  - **cero** si `o1` es igual a `o2`.
  - **positivo** si `o1` es mayor que `o2`.

Es deseable que el método `compare(o1, o2)` sea **consistente** con `o1.equals(o2)`. En caso de que no lo sea, tanto el método `add()` sobre un conjunto ordenado, como el método `put()` sobre una correspondencia ordenada, utilizan el método `compare()` en vez de `equals()` para comprobar la igualdad de elementos o claves.

# Ejemplo de uso de java.util.Comparator<T>

```
import java.util.*;
public class OrdenPersona implements Comparator<Persona> {
    // Comparación 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());
        }
        return resultado;
    }
}
```

```
import java.util.*;
public class MainPersona3 {
    public static void main(String[] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);
        Comparator<Persona> op = new OrdenPersona();
        System.out.println(op.compare(p1, p2));
    }
}
```

# Composición de órdenes alternativos

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

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

import java.util.*;

public class MainPersona4 {
    public static void main(String[] args) {
        Persona p1 = new Persona("Juan", 35);
        Persona p2 = new Persona("Pedro", 22);

        Comparator<Persona> op1 = new OrdenEdad().thenComparing(new OrdenNombre());
        System.out.println(op1.compare(p1, p2));

        Comparator<Persona> op2 = new OrdenNombre().reversed().thenComparing(new OrdenEdad());
        System.out.println(op2.compare(p1, p2));

        Comparator<Persona> op3 = Comparator.naturalOrder(); // Inferencia de Tipos
        System.out.println(op3.compare(p1, p2));
    }
}
```

# Uso de Comparable<Persona> y Comparator<Persona>

- Podemos definir una Asamblea como 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  
    }  
}
```

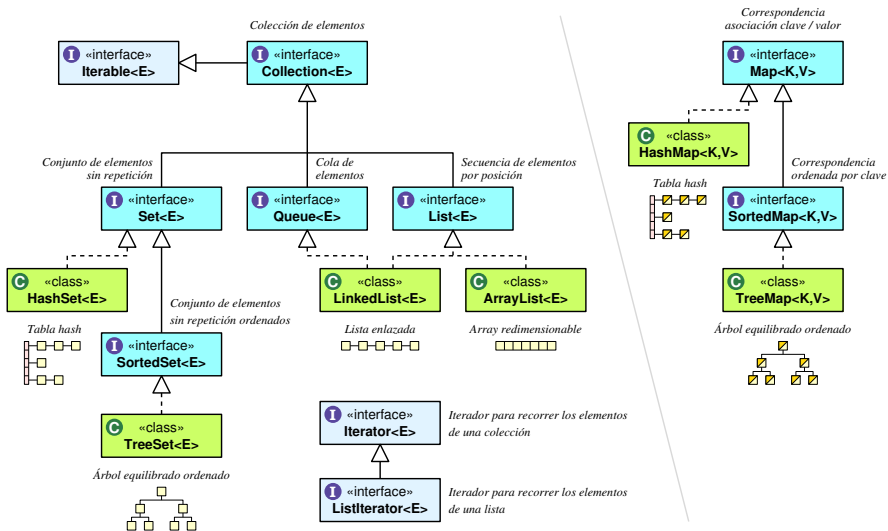
- Podemos utilizarla donde sea necesaria.

```
Asamblea a = new Asamblea();  
Asamblea a = new Asamblea(new OrdenPersona());  
// ...
```

# Colecciones y Correspondencias (Asociaciones)




- El marco de **colecciones** y **correspondencias** de Java proporciona (en el paquete `java.util`) varias clases e interfaces adecuadas para el manejo de colecciones y asociaciones de datos.
- Proporciona:
  - **Interfaces** que especifican el comportamiento para manipular colecciones y asociaciones de datos de forma independiente de la implementación.
  - **Clases** que proporcionan una implementación del comportamiento especificado por las interfaces.
    - Las implementaciones pueden proporcionar propiedades y características diferentes.
  - **Algoritmos** adicionales para realizar determinadas operaciones sobre colecciones, como ordenaciones, búsquedas, etc.
- Beneficios de usar el marco de colecciones y correspondencias:
  - Reduce los esfuerzos de programación y los errores.
  - Incrementa la eficiencia y la calidad del software.
  - Ayuda a la interoperabilidad y reemplazabilidad.

# Colecciones y Correspondencias (Asociaciones)





# Cuadro resumen de las colecciones y correspondencias

Interfaz	Collection<E>	Set<E>	SortedSet<E>	Queue<E>	List<E>	Map<K,V>	SortedMap<K,V>	Iterator<E>	ListIterator<E>
Implement		HashSet	TreeSet	LinkedList	ArrayList LinkedList	HashMap	TreeMap		
Consultas	size() isEmpty() contains(o) containsAll(c)		first() last() comparator()	element() <sup>e</sup> peek()	get(i) indexOf(o) lastIndexOf(o)	size() isEmpty() get(k) containsKey(k) containsValue(v)	firstKey() lastKey() comparator()	hasNext() next()	hasPrevious() previous() nextIndex() previousIndex()
Añadir	add(e) addAll(c)			add(e) <sup>e</sup> offer(e)	add(i, e) addAll(i, c)	put(k, v) putAll(m)			add(e)
Eliminar	clear() remove(o) removeAll(c) retainAll(c)			remove() <sup>e</sup> poll()	remove(i)	clear() remove(k)		remove()	
Modificar					set(i, e) sort(cp)				set(e)
Iterar	iterator()				listIterator() listIterator(i)				
Vistas	toArray() toArray(a)		headSet(t) tailSet(f) subSet(f, t)		subList(f, t)	entrySet() keySet() values()	headMap(t) tailMap(f) subMap(f, t)		

**Map.Entry<K,V>**  
 getKey()  
 getValue()  
 setValue(v)

Leyenda: [ o : Object ] [ e : Element ] [ c : Collection ] [ i : Index (int) ] [ f : From ] [ t : To ] [ a : Array ] [ k : Key ] [ v : Value ] [ m : Map ]

# Colecciones y Correspondencias ordenadas

- Una clase puede especificar una relación de orden por medio de:
  - La interfaz `Comparable<T>` (orden natural)
  - La interfaz `Comparator<T>` (orden alternativo)
- Estas relaciones se utilizan tanto en conjuntos y correspondencias ordenadas, como en algoritmos de ordenación.
  - Los objetos que implementan un orden natural o alternativo pueden ser utilizados:
    - Como **elementos** de un conjunto ordenado (`SortedSet<E>`).
    - Como **claves** en una correspondencia ordenada (`SortedMap<K,V>`).
    - En listas ordenables con el **método** `sort(...)`, etc.

# Implementaciones de las interfaces

- No hay implementación directa de la interfaz `Collection<E>`. Ésta se utiliza sólo para mejorar la interoperación de las distintas colecciones.
- Por convención, las clases que implementan colecciones proporcionan **constructores** para crear nuevas colecciones con los elementos de un objeto del tipo `Collection<E>` que se le pasa como argumento.
- Lo mismo sucede con las implementaciones de `Map<K,V>`.
- Colecciones y correspondencias (asociaciones) **no** son intercambiables.
- Todas las implementaciones descritas son “**modificables**” (implementan los métodos etiquetados como opcionales que son los que modifican la estructura).

# Convenciones sobre excepciones

- Las clases e interfaces de colecciones siguen las siguientes convenciones:
  - Si el tamaño especificado para construir la colección no es adecuado, entonces los constructores y algunos otros métodos lanzan la excepción:
    - `IllegalArgumentException`
  - Si se accede a un elemento que no existe, entonces los métodos lanzarán la excepción:
    - `NoSuchElementException`
  - Si se pasa una referencia `null`, entonces los métodos y constructores suelen lanzar una excepción:
    - `NullPointerException`
  - Si los elementos que van a ser insertados en la colección no son del tipo apropiado, entonces los métodos y constructores lanzan la excepción:
    - `ClassCastException`
  - Los métodos opcionales “**no implementados**” lanzan la excepción:
    - `UnsupportedOperationException`

# La interfaz java.util.Collection<E>

```
public interface Collection<E> extends Iterable<E> {  
    // Consultar  
    int size(); // Dev. el número de elementos  
    boolean isEmpty(); // Dev. true si está vacío  
    boolean contains(Object elemento); // Dev. true si está el elemento  
    boolean containsAll(Collection<?> c); // Dev. true si contiene todos elms de c  
  
    // Añadir (devuelven TRUE si cambian los elementos de la colección)  
    boolean add(E elemento); // Añade el elemento  
    boolean addAll(Collection<? extends E> c); // Añade todos los elementos de c  
  
    // Eliminar (devuelven TRUE si cambian los elementos de la colección)  
    void clear(); // Elimina todos los elementos  
    boolean remove(Object elemento); // Elimina el elemento  
    boolean removeAll(Collection<?> c); // Elimina todos los elementos de c  
    boolean retainAll(Collection<?> c); // Elimina todos los elms que no están en c  
  
    // Iteración  
    Iterator<E> iterator(); // Dev. iterador al comienzo de la colección  
  
    // Operaciones con arrays  
    Object[] toArray(); // Dev. array con todos los objetos de la colección  
    <T> T[] toArray(T[] a); // Dev. array con todos los elementos de la colección  
}  
// toString(): [elemento, elemento, elemento, ..., elemento]
```

# Las interfaces `Iterable<E>` e `Iterator<E>`

- La interfaz `Iterable<E>` especifica un método que devuelve una instancia de alguna clase que implemente la interfaz `Iterator<E>`.

```
public interface Iterable<E> {  
    Iterator<E> iterator(); // dev. iterador al comienzo de la colección  
}
```

- Un objeto `Iterator<E>` permite el acceso secuencial a los elementos de una colección y realizar recorridos sobre la colección.

```
public interface Iterator<E> {  
    boolean hasNext(); // Comprueba si hay siguiente elemento  
    E next();           // Devuelve el siguiente elemento y mueve el iterador  
    void remove();      // Se invoca después de next() para eliminar elemento  
}
```

- Si no hay siguiente, `next()` lanza una excepción `NoSuchElementException`.
- El método `remove()` permite eliminar elementos de la colección.
  - Ésta es la **única forma adecuada** para eliminar elementos durante la iteración. En otro caso, se lanza una excepción `ConcurrentModificationException`.
  - Sólo puede haber una invocación a `remove()` por cada invocación a `next()`. Si no se cumple, se lanza una excepción `IllegalStateException`.
- Si se modifica la colección, todos los iteradores quedan **invalidados**, excepto el caso del iterador cuando se le aplica el método `remove()`.
- No se puede modificar la colección dentro del bucle **for-each**.

# Ejemplo del uso de iteradores

- Mostrar una colección en pantalla utilizando **for-each**:

```
static <E> void mostrar(Collection<E> c) {  
    System.out.print("[");  
    for (E e : c) {  
        System.out.print(" " + e.toString());  
    }  
    System.out.println(" ]");  
}
```

- Mostrar una colección en pantalla utilizando **iteradores**:

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

- **No se puede modificar** la colección mientras se utiliza *for-each* o iteradores, salvo el método `remove()`.

# Ejemplo del uso de iteradores

- Mostrar una colección en pantalla utilizando **iteradores**, separando los elementos con ',':

```
static <E> void mostrar(Collection<E> c) {  
    System.out.print("[");  
    Iterator<E> iter = c.iterator();  
    if (iter.hasNext()) {  
        E e = iter.next();  
        System.out.print(" " + e.toString());  
        while (iter.hasNext()) {  
            E e = iter.next();  
            System.out.print(", " + e.toString());  
        }  
    }  
    System.out.println(" ]");  
}
```

- Eliminar las cadenas largas de una colección de cadenas.

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

- No se puede modificar** la colección mientras se utiliza *for-each* o iteradores, salvo el método `remove()`.



# Ejemplo del uso de iteradores

- Buscar un elemento en una colección:

```
static <E> E buscar(Collection<E> c, E e) {  
    // if ( ! c.contains(e) ) { throw new NoSuchElementException("No encontrado"); }  
    E a = null;  
    Iterator<E> it = c.iterator();  
    while ((a == null) && it.hasNext()) {  
        E x = it.next();  
        if (x.equals(e)) {  
            a = x;  
        }  
    }  
    // if (a == null) { throw new NoSuchElementException("No encontrado"); }  
    return a;  
}
```

- Buscar un elemento en una colección (solución alternativa):

```
static <E> E buscar(Collection<E> c, E e) {  
    // if ( ! c.contains(e) ) { throw new NoSuchElementException("No encontrado"); }  
    E a = null;  
    boolean ok = false;  
    Iterator<E> it = c.iterator();  
    while ( ! ok && it.hasNext()) {  
        a = it.next();  
        ok = a.equals(e);  
    }  
    // if (! ok) { throw new NoSuchElementException("No encontrado"); }  
    return ok ? a : null;  
}
```

- No se puede modificar la colección mientras se utiliza *for-each* o iteradores, salvo el método `remove()`.

# La interfaz java.util.Set<E>

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

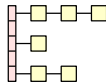
```
public interface Set<E> extends Collection<E> { /* ... */ }
```

- No permite elementos repetidos (control con `equals()`, `hashCode()`, `compareTo()`, `compare()`).
- Los elementos no están asociados a ninguna posición concreta.
- Los métodos definidos permiten realizar la lógica de conjuntos:

Método	Operación
<code>a.contains(e)</code>	$\dot{?} e \in a$
<code>a.add(e)</code>	$a = a \cup \{e\}$
<code>a.remove(e)</code>	$a = a - \{e\}$
<code>a.containsAll(b)</code>	$\dot{?} b \subseteq a$
<code>a.addAll(b)</code>	$a = a \cup b$
<code>a.removeAll(b)</code>	$a = a - b$
<code>a.retainAll(b)</code>	$a = a \cap b$
<code>a.clear()</code>	$a = \emptyset$
<code>a.isEmpty()</code>	$\dot{?} a == \emptyset$
<code>a.size()</code>	$ a $

# Implementación de Set<E>: java.util.HashSet<E>

- `java.util.HashSet<E>` proporciona una implementación de `Set<E>`:
  - Almacena los datos en una tabla hash (control con `equals()` y `hashCode()`).
  - Búsqueda, inserción y eliminación en tiempo (casi) constante.
  - Constructores:
    - ▶ `HashSet()`; *// Conjunto vacío*
    - ▶ `HashSet(Collection<? extends E> c)`; *// Copia elementos de c*
    - ▶ `HashSet(int c, float fc)`; *// capacidad inicial y factor carga*



- Ejemplo, diferenciar palabras con repetición y sin repetición:

```
import java.util.*;
public class Repetidos {
    public static void main(String[] args) {
        Set<String> palabras = new HashSet<String>();
        Set<String> palabrasNoRepetidas = new HashSet<String>();
        for (String palabra : args) {
            if ( palabras.contains(palabra) ) {
                palabrasNoRepetidas.remove(palabra);
            } else {
                palabrasNoRepetidas.add(palabra);
            }
            palabras.add(palabra); // añade sin repetición
        }
        System.out.println("Palabras: " + palabras.toString());
        System.out.println("Palabras no repetidas: " + palabrasNoRepetidas.toString());
    }
}
```

# La interfaz `java.util.SortedSet<E>`

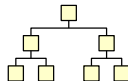
- La interfaz `java.util.SortedSet<E>` extiende `Set<E>` para proporcionar la funcionalidad para conjuntos con elementos ordenados. El orden utilizado es:
  - El orden natural (control con `compareTo()`).
  - El orden alternativo proporcionado como parámetro `Comparator<E>` en el constructor de la clase que implemente esta interfaz (control con `compare()`).

```
public interface SortedSet<E> extends Set<E> {  
    // Vistas de rangos  
    SortedSet<E> headSet(E toElement);           // elementos menores  
    SortedSet<E> tailSet(E fromElement);         // elementos mayores o iguales  
    SortedSet<E> subSet(E fromElement, E toElement); // elementos desde-hasta  
  
    E first();                                     // elemento mínimo  
    E last();                                      // elemento máximo  
  
    Comparator<? super E> comparator();          // acceso al comparador  
}
```

# Impl. de SortedSet<E>: java.util.TreeSet<E>

- `java.util.TreeSet<E>` proporciona una implementación de `SortedSet<E>`:

- Almacena los datos en un árbol binario equilibrado ordenado.
- Búsqueda y modificación menos eficiente que en `HashSet<E>`.
- Constructores:



- ▶ `TreeSet()`; *// Orden natural*
- ▶ `TreeSet(Comparator<? super E> o)`; *// Orden alternativo*
- ▶ `TreeSet(Collection<? extends E> c)`; *// Copia y Orden natural*
- ▶ `TreeSet(SortedSet<E> s)`; *// Copia y Mismo orden que s*

- Ejemplo de la clase Asamblea:

```
import java.util.*;
public class Asamblea { // grupo de personas (ordenadas)
    private SortedSet<Persona> personas; // conjunto ordenado almacén
    public Asamblea() {
        personas = new TreeSet<Persona>(); // orden natural
    }
    public Asamblea(Comparator<Persona> comp) {
        personas = new TreeSet<Persona>(comp); // orden alternativo
    }
}
```

# La interfaz java.util.List<E>

- Colección de elementos que forman una **secuencia**, donde cada elemento tiene una **posición** en la colección. Además de los métodos especificados por **Collection<E>**.
  - En caso de acceso incorrecto, lanza **IndexOutOfBoundsException**.

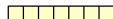
```
public interface List<E> extends Collection<E> {  
    // Consultar (acceso posicional)  
    E get(int index);           // Devuelve el elemento de la posición  
    // Buscar (devuelve -1 si no encontrado)  
    int indexOf(Object o);      // Devuelve la primera posición del objeto  
    int lastIndexOf(Object o); // Devuelve la última posición del objeto  
    // Añadir  
    boolean add(E element);     // Añade elemento al final  
    void add(int index, E element); // Añade elemento en la posición  
    boolean addAll(int index, Collection<? extends E> c); //Añade elms en pos  
    // Eliminar  
    E remove(int index);       // Elimina elemento de la posición  
    // Modificar (acceso posicional)  
    E set(int index, E element); // Modifica elemento de la posición  
    void sort(Comparator<? super E> cp); // Ordena la lista (si null orden nat)  
    // Iteración  
    ListIterator<E> listIterator(); // Iterador al comienzo de la lista  
    ListIterator<E> listIterator(int index); // Iterador a la pos de la lista  
    // Vista de subrango  
    List<E> subList(int desde, int hasta); // Sublista desde sin incluir hasta  
}
```

# Implementaciones de `java.util.List<E>`

- `java.util.ArrayList<E>` proporciona una implementación de `List<E>`:

- Implementación basada en **array redimensionable** dinámicamente.
- Inserción y eliminación *ineficientes* (excepto al final).
- Consultas por posición *eficientes* (acceso directo).
- Constructores:

```
▶ ArrayList(); // Lista vacía
▶ ArrayList(Collection<? extends E> c); // Copia elementos de c
▶ ArrayList(int c); // capacidad inicial
```



- `java.util.LinkedList<E>` proporciona una implementación de `List<E>`:

- Implementación basada en **lista doblemente enlazada**.
- Inserción y eliminación *eficientes*.
- Consultas por posición *ineficientes* (acceso secuencial).
- Constructores:

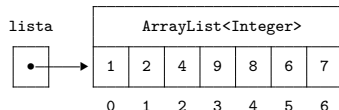
```
▶ LinkedList(); // Lista vacía
▶ LinkedList(Collection<? extends E> c); // Copia elementos de c
```



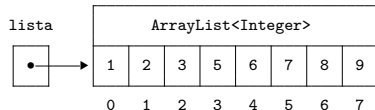
# La interfaz java.util.List<E>

## • Ejemplo:

```
Integer[] datos = { 1, 2, 4, 9, 8, 6, 7 };  
List<Integer> lista = new ArrayList<Integer>(Arrays.asList(datos));
```



```
► int a = lista.size();           // a=7 [ 1, 2, 4, 9, 8, 6, 7 ]  
► int b = lista.get(0);          // b=1 [ 1, 2, 4, 9, 8, 6, 7 ]  
► int c = lista.get(6);          // c=7 [ 1, 2, 4, 9, 8, 6, 7 ]  
► int d = lista.set(3, 5);        // d=9 [ 1, 2, 4, 5, 8, 6, 7 ]  
► int e = lista.remove(4);        // e=8 [ 1, 2, 4, 5, # 6, 7 ]  
► lista.remove(Integer.valueOf(4)); // [ 1, 2, # 5, 6, 7 ]  
► lista.add(2, 3);                // [ 1, 2, 3, 5, 6, 7 ]  
► lista.add(6, 8);                // [ 1, 2, 3, 5, 6, 7, 8 ]  
► lista.add(9);                   // [ 1, 2, 3, 5, 6, 7, 8, 9 ]
```





# Ordenando Listas

- La interfaz `java.util.List<E>` incluye un método por defecto que permite ordenar listas (si `c` es `null` se usa el orden natural):

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

- Por ejemplo:

```
import java.util.*;
public class PalabrasOrdenadas {
    public static void main(String[] args) {
        // creamos la lista con los argumentos de main
        List<String> lista = new LinkedList<String>();
        for (String arg : args) {
            lista.add(arg);
        }
        lista.sort(null); // ordena según el orden natural
        System.out.println("Palabras ordenadas: " + lista);
    }
}
// args: pepe juan maria lola pepe ana maria pepe
// Palabras ordenadas: [ana, juan, lola, maria, maria, pepe, pepe, pepe]
```

# La interfaz java.util.ListIterator<E>

- La interfaz `List<E>` posee iteradores especializados (`ListIterator<E>`)
- Un objeto `ListIterator<E>` permite el acceso secuencial bidireccional a los elementos de una lista y realizar recorridos sobre la lista.

```
public interface ListIterator<E> extends Iterator<E> {  
    boolean hasNext();           // Comprueba si hay siguiente elemento  
    E next();                    // Devuelve el siguiente elemento y mueve el iterador  
    boolean hasPrevious();       // Comprueba si hay elemento previo  
    E previous();                // Devuelve el elemento previo y mueve el iterador  
  
    int nextIndex();             // Devuelve el índice del next()  
    int previousIndex();         // Devuelve el índice del previous()  
  
    void remove();              // Elimina el next() o previous() actual  
    void set(E o);              // Actualiza el next() o previous() actual  
    void add(E o);              // Añade antes del next() y después del previous()  
}
```

# Ejemplo de uso de java.util.List<E>

```
import java.util.*;

public class Palindromo {
    public static void main(String[] args) {
        // creamos la lista con los argumentos de main
        List<String> lista = new LinkedList<String>();
        for (String arg : args) {
            lista.add(arg);
        }
        if (lista.isEmpty()) {
            System.out.println("La lista está vacía");
        } else {
            // iterador al principio de la lista
            ListIterator<String> iterInicio = lista.listIterator();
            // iterador al final de la lista
            ListIterator<String> iterFinal = lista.listIterator(lista.size());
            boolean ok = true;
            while ( ok && iterInicio.hasNext() && iterFinal.hasPrevious() ) {
                String e1 = iterInicio.next();
                String e2 = iterFinal.previous();
                if ( ! e1.equals(e2) ) {
                    ok = false;
                }
            }
            if ( ok ) {
                System.out.println("La lista es palíndromo: " + lista);
            } else {
                System.out.println("La lista no es palíndromo: " + lista);
            }
        }
    }
}
```



# Implementaciones de java.util.Queue<E>

- java.util.LinkedList<E> proporciona una implementación de Queue<E>:

- Implementación basada en **lista doblemente enlazada**.



- Constructores:

- ▶ `LinkedList()`; *// Cola vacía*
- ▶ `LinkedList(Collection<? extends E> c)`; *// Copia elementos de c*

- Por ejemplo:

```
import java.util.*;
public class ColaPalabras {
    public static void main(String[] args) {
        // creamos una cola con los argumentos de main
        Queue<String> cola = new LinkedList<String>();
        for (String arg : args) {
            cola.add(arg);
        }
        while ( ! cola.isEmpty() ) {
            String palabra = cola.remove();
            System.out.print(" " + palabra);
        }
        System.out.println();
    }
}
// args: pepe juan maria lola pepe ana maria pepe
// salida: pepe juan maria lola pepe ana maria pepe
```

# La interfaz `java.util.Map<K,V>`

- La interfaz `Map<K,V>` define **correspondencias** (asociaciones) de claves a valores asociados.
  - Las claves son únicas, sin repetición (control con `equals()`, `hashCode()`, `compareTo()`, `compare()`).
  - Cada clave se asocia con un único valor.
- Una correspondencia no es una colección, y por esto la interfaz `Map<K,V>` **no** hereda de `Collection<E>`. Sin embargo, una correspondencia puede ser **vista** como una colección de varias formas:
  - Como un conjunto de claves.
  - Como una colección de valores.
  - Como un conjunto de pares *<clave, valor>*.
- Al igual que en `Collection<E>`, algunas de las operaciones son opcionales, y si se invoca una operación no implementada, entonces lanza la excepción `UnsupportedOperationException`.
  - Las implementaciones del paquete `java.util` implementan todas las operaciones.

# La interfaz java.util.Map<K,V>

```
public interface Map<K,V> {  
    // Consultar  
    int size();                // Dev. el número de elementos  
    boolean isEmpty();         // Dev. true si está vacío  
    V get(Object key);         // Dev. el valor asociado a la clave (null si no está)  
    V getOrDefault(Object key, V defVal); // Dev. el valor asociado a la clave  
    boolean containsKey(Object key); // Dev. true está la clave  
    boolean containsValue(Object value); // Dev. true está el valor  
    // Añadir  
    V put(K key, V value);      // Añade la clave y el valor asociado  
    V putIfAbsent(K key, V value); // Añade la clave y el valor (si clave no está)  
    void putAll(Map<? extends K, ? extends V> m); // Añade la correspondencia  
    // Eliminar  
    void clear();              // Elimina todos los elementos  
    V remove(Object key);      // Elimina la clave y su valor asociado  
  
    // Vistas como colecciones  
    Set<K> keySet();           // Dev. el conjunto con todas las claves  
    Collection<V> values();    // Dev. la colección con todos los valores  
    Set<Map.Entry<K,V>> entrySet(); // Dev. el conjunto con todas las entradas  
  
    // Interfaz para las entradas de la correspondencia  
    static interface Entry<K,V> {  
        K getKey();           // Devuelve la clave de la entrada  
        V getValue();         // Devuelve el valor de la entrada  
        V setValue(V value);  // Modifica el valor de la entrada  
    }  
}  
  
// toString(): {clave=valor, clave=valor, clave=valor, ..., clave=valor}
```

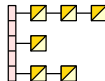
# La interfaz `java.util.Map<K,V>`

- El método por defecto `getOrDefault()` accede al valor correspondiente a la clave asociada. Si no existe, entonces devuelve `defaultValue`.
- El método por defecto `putIfAbsent()`, si no existe la clave previamente, entonces añade la correspondencia entre la clave y su valor asociado y devuelve `null`, en otro caso, devuelve el valor previamente asociado con la clave.

```
default V getOrDefault(Object key, V defVal) {  
    V v = get(key);  
    if (v == null) {  
        v = defVal;  
    }  
    return v;  
}
```

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

- La clase `java.util.HashMap<K,V>` proporciona una implementación de la interfaz `Map<K,V>` basada en **tabla hash**.
  - Almacena los datos en una tabla hash (control con `equals()` y `hashCode()`).
  - Búsqueda, inserción y eliminación en tiempo (casi) constante.
  - Constructores:
    - `HashMap()`; *// correspondencia vacía*
    - `HashMap(Map<? extends K, ? extends V> m)`; *// copia elementos de m*
    - `HashMap(int c, float fc)`; *// cap. inicial y factor carga*





# Ejemplo 1 de uso de java.util.Map<K,V> (keySet)

```
import java.util.*;
public class Frecuencias {
    // args: pepe juan maria lola pepe ana maria pepe
    public static void main(String[] args) {
        Map<String, Integer> frecs = new HashMap<String, Integer>();
        for (String clave : args) {
            // Incrementa la cuenta de cada clave
            Integer cnt = frecs.get(clave);
            if (cnt == null) {
                frecs.put(clave, 1);
            } else {
                frecs.put(clave, cnt + 1);
            }
            // int cnt = frecs.getOrDefault(clave, 0);
            // frecs.put(clave, cnt + 1);
        }
        // Mostramos frecuencias iterando sobre el conjunto de claves
        for (String clave : frecs.keySet()) {
            int cnt = frecs.get(clave);
            System.out.println(clave + ":\t" + cnt);
        }
    }
}
```

este esquema es muy habitual

ana:	1
juan:	1
lola:	1
maria:	2
pepe:	3

## Ejemplo 2 de uso de `java.util.Map<K,V>` (`entrySet`)

```
import java.util.*;
public class Frecuencias {
    // args: pepe juan maria lola pepe ana maria pepe
    public static void main(String[] args) {
        Map<String, Integer> frecs = new HashMap<String, Integer>();
        for (String clave : args) {
            // Incrementa la cuenta de cada clave
            Integer cnt = frecs.get(clave);
            if (cnt == null) {
                cnt = 0;
            }
            frecs.put(clave, cnt + 1);
            // int cnt = frecs.getOrDefault(clave, 0);
            // frecs.put(clave, cnt + 1);
        }
        // Mostramos frecuencias iterando sobre el conjunto de entradas
        for (Map.Entry<String, Integer> entry : frecs.entrySet()) {
            String clave = entry.getKey();
            int cnt = entry.getValue();
            System.out.println(clave + ":\t" + cnt);
        }
    }
}
```

este esquema es muy habitual

## Ejemplo 3 de uso de Map<K,V> (clave y colección)

```
import java.util.*;
public class Posiciones {
    // args: pepe juan maria lola pepe ana maria pepe
    public static void main(String[] args) {
        Map<String,List<Integer>> mPos = new HashMap<String,List<Integer>>();
        for (int i = 0; i < args.length; i++) {
            String clave = args[i];
            // Buscamos la lista de posiciones asociada a clave
            List<Integer> lPos = mPos.get(clave);
            if (lPos == null) {
                lPos = new LinkedList<Integer>();
                lPos.add(i);
                mPos.put(clave, lPos);
            } else {
                lPos.add(i);
            }
        }
        // Mostramos posiciones iterando sobre el conjunto de entradas
        for (Map.Entry<String,List<Integer>> entrada : mPos.entrySet()) {
            String clave = entrada.getKey();           // ana:      [5]
            List<Integer> lPos = entrada.getValue();    // juan:     [1]
            System.out.println(clave + ":\t" + lPos);    // lola:     [3]
                                                        // maria:    [2, 6]
                                                        // pepe:     [0, 4, 7]
        }
    }
}
```

este esquema es muy habitual

## Ejemplo 4 de uso de Map<K,V> (clave y colección)

```
import java.util.*;
public class Posiciones {
    // args: pepe juan maria lola pepe ana maria pepe
    public static void main(String[] args) {
        Map<String,List<Integer>> mPos = new HashMap<String,List<Integer>>();
        for (int i = 0; i < args.length; i++) {
            String clave = args[i];
            // Buscamos la lista de posiciones asociada a clave
            List<Integer> lPos = mPos.get(clave);
            if (lPos == null) {
                lPos = new LinkedList<Integer>();
                mPos.put(clave, lPos);
            }
            lPos.add(i);
        }
        // Mostramos posiciones iterando sobre el conjunto de entradas
        for (Map.Entry<String,List<Integer>> entrada : mPos.entrySet()) {
            String clave = entrada.getKey();           // ana:      [5]
            List<Integer> lPos = entrada.getValue();    // juan:     [1]
            System.out.println(clave + ":\t" + lPos);    // lola:     [3]
        }                                               // maria:    [2, 6]
    }                                                  // pepe:    [0, 4, 7]
}
```

este esquema es muy habitual

# La interfaz `java.util.SortedMap<K,V>`

- La interfaz `java.util.SortedMap<K,V>` extiende `Map<K,V>` para proporcionar la funcionalidad para correspondencias con claves ordenadas. El orden utilizado es:
  - El orden natural (control con `compareTo()`).
  - El orden alternativo proporcionado como parámetro `Comparator<K>` en el constructor de la clase que implemente esta interfaz (control con `compare()`).

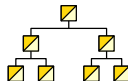
```
public interface SortedMap<K,V> extends Map<K,V> {  
    // Vistas de rangos  
    SortedMap<K,V> headMap(K toKey);           // elementos menores  
    SortedMap<K,V> tailMap(K fromKey);          // elementos mayores o iguales  
    SortedMap<K,V> subMap(K fromKey, K toKey);  // elementos desde-hasta  
  
    K firstKey();                               // clave mínima  
    K lastKey();                                // clave máxima  
  
    Comparator<? super K> comparator();         // acceso al comparador  
}
```

# Impl. de SortedMap<K,V>: java.util.TreeMap<K,V>

- `java.util.TreeMap<K,V>` proporciona una implementación de `SortedMap<K,V>`:

- Almacena los datos en un árbol binario equilibrado ordenado.
- Búsqueda y modificación menos eficiente que en `HashMap<K,V>`.
- Constructores:

```
▶ TreeMap(); // Orden natural
▶ TreeMap(Comparator<? super K> o); // Orden alternativo
▶ TreeMap(Map<? extends K, ? extends V> m); // Copia y Orden natural
▶ TreeMap(SortedMap<K, ? extends V> m); // Copia y Mismo orden que m
```



# Ejemplo de uso de java.util.SortedMap<K,V>

```
import java.util.*;
public class Posiciones {
    public static void main(String[] args) {
        SortedMap<String,List<Integer>> mPos = new TreeMap<String,List<Integer>>();
        for (int i = 0; i < args.length; i++) {
            String clave = args[i];
            // Buscamos la lista asociada a clave en mPos
            List<Integer> lPos = mPos.get(clave);
            if (lPos == null) {
                lPos = new LinkedList<Integer>();
                mPos.put(clave, lPos);
            }
            lPos.add(i);
        }
        // Mostramos posiciones iterando sobre el conjunto de entradas
        for (Map.Entry<String,List<Integer>> entrada : mPos.entrySet()) {
            String clave = entrada.getKey();
            List<Integer> lPos = entrada.getValue();
            System.out.println(clave + ":\t" + lPos);
        }
    }
}
```

este esquema es muy habitual

# Modificación de los componentes de los elementos

- Existen algunas restricciones respecto a la modificación de los componentes de los elementos alojados en colecciones y correspondencias:
- Set y SortedSet: no se deben modificar los componentes que afecten al acceso del elemento (`hashCode`, `equals`, `compareTo` y `compare`). En caso de ser necesario, se debe eliminar del conjunto el elemento con los valores antiguos y añadir posteriormente el elemento con los nuevos valores.
- List y Queue: sí se pueden modificar los componentes de los elementos, ya que el acceso a los elementos no depende de los valores propios de sus componentes, sino de la posición que ocupan.
- Map y SortedMap: sí se puede modificar el valor asociado a una determinada clave, pero no se deben modificar los componentes que afecten al acceso de la clave (`hashCode`, `equals`, `compareTo` y `compare`). En caso de ser necesario, se debe eliminar de la correspondencia la clave con los valores antiguos y añadir posteriormente la clave con los nuevos valores, asociada al valor correspondiente.



# Creación de Set, List y Map Constantes: método of(...)

- El método de clase (estático) `of(...)` proporcionado por las interfaces (`Set`, `List` y `Map`) permite crear instancias **constantes** (*inmutables*) con los elementos especificados (a partir de la *versión 9 de java*).

```
Set<String> c1 = Set.of("hola", "que", "tal");  
List<Integer> l1 = List.of(1, 2, 3, 4, 5, 6);  
Map<String, Integer> m1 = Map.of("hola", 1, "que", 2, "tal", 3);
```

- El método `of()` de `Map` tiene un máximo de **10** pares de argumentos. Se puede utilizar el método `Map.ofEntries(...)` sin esa limitación, utilizando el método `Map.entry(c,v)` (a partir de la *versión 9 de java*):

```
Map<String, Integer> m2 = Map.ofEntries(Map.entry("hola", 1),  
                                         Map.entry("adios", 2));
```

- También es posible crear una lista **constante** (*immutable*) con el método de clase (estático) `asList(...)` proporcionado por la clase `Arrays`.

```
List<String> l2 = Arrays.asList("hola", "que", "tal");
```

- Los métodos especificados se pueden aplicar a **argumentos individuales** o a un **array** con los elementos.
- Se pueden usar en constructores o para añadir elementos (`addAll()`, `putAll()`).

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

```
▶ static <E> void copy(List<? super E> dest, List<? extends E> src); // copiar elms
▶ static <E> void fill(List<? super E> list, E o); // rellenar elementos
▶ static void reverse(List<?> list); // invertir elementos
▶ static void shuffle(List<?> list); // barajar elementos

▶ static <E extends Comparable<? super E>> void sort(List<E> list);
▶ static <E> int binarySearch(List<? extends Comparable<? super E>> list, E key);
▶ static <E extends Object & Comparable<? super E>> E min(Collection<E> coll);
▶ static <E extends Object & Comparable<? super E>> E max(Collection<E> coll);

▶ static <E> void sort(List<E> list, Comparator<? super E> c);
▶ static <E> int binarySearch(List<? extends E> list, E key, Comparator<? super E> c);
▶ static <E> E min(Collection<? extends E> coll, Comparator<? super E> c)
▶ static <E> E max(Collection<? extends E> coll, Comparator<? super E> c)
```

# Creación de vistas de colecciones y correspondencias

- La clase `Collections` proporciona métodos factoría para crear **vistas no-modificables** de colecciones y correspondencias:

```
▶ static <E>    Collection<E>    unmodifiableCollection(Collection<? extends E> c);
▶ static <E>    Set<E>           unmodifiableSet(Set<? extends E> s);
▶ static <E>    SortedSet<E>     unmodifiableSortedSet(SortedSet<E> s);
▶ static <E>    List<E>          unmodifiableList(List<? extends E> l);
▶ static <K,V>  Map<K,V>         unmodifiableMap(Map<? extends K,? extends V> m);
▶ static <K,V>  SortedMap<K,V>   unmodifiableSortedMap(SortedMap<K,? extends V> m);
```

- La clase `Collections` proporciona métodos factoría para crear **vistas seguras en entornos multi-hebras** de colecciones y correspondencias:

```
▶ static <E>    Collection<E>    synchronizedCollection(Collection<E> c);
▶ static <E>    Set<E>           synchronizedSet(Set<E> s);
▶ static <E>    SortedSet<E>     synchronizedSortedSet(SortedSet<E> s);
▶ static <E>    List<E>          synchronizedList(List<E> l);
▶ static <K,V>  Map<K,V>         synchronizedMap(Map<K,V> m);
▶ static <K,V>  SortedMap<K,V>   synchronizedSortedMap(SortedMap<K,V> m);
```

# La clase java.util.Arrays

- La clase `java.util.Arrays` proporciona métodos estáticos que implementan algoritmos sobre arrays de elementos de tipos primitivos u `Object`.

► `static int binarySearch(T[] a, T e);` *// T es un tipo primitivo u Object*

- Devuelve el índice de la posición del elemento `e` en `a`.
- Devuelve `-p-1` si `e` no está (`p` posición para insertar elemento `e` ordenado).

- Otros métodos estáticos:

```
► static String toString(T[] a);                // [a1, a2, ..., an]
► static String deepToString(Object[] a);        // [a1, a2, ..., an]
► static boolean equals(T[] a1, T[] a2);         // ¿ arrays iguales ?
► static boolean deepEquals(Object[] a1, Object[] a2); // ¿ arrays iguales ?
► static int hashCode(T[] a);                   // hashCode del array
► static int deepHashCode(Object[] a);           // hashCode del array
► static T[] copyOf(T[] a, int l);               // duplica y copia array
► static T[] copyOfRange(T[] a, int d, int h);    // duplica y copia array
► static void fill(T[] a, T e);                  // rellena array
► static void sort(T[] a);                       // ordena array
► static <T> void sort(T[] a, Comparator<? super T> c); // ordena array
► static <T> int binarySearch(T[] a, T e, Comparator<? super T> c);
► static <T> List<T> asList(T[] a);              // Dev. lista con elms de array
► static <T> List<T> asList(T... a);             // Dev. lista con elms de array
```

# Ejemplo de uso de `java.util.Arrays.asList()`

- El método de clase (estático) `asList(...)` proporcionado por la clase `Arrays` permite crear una lista **constante** (*inmutable*) a partir de los elementos de un array, o de los múltiples valores que se le pasen como parámetros.

```
String[] palabras = { "hola", "que", "tal" };  
List<String> l1 = Arrays.asList(palabras);  
List<String> l2 = Arrays.asList("hola", "que", "tal");
```

- Ejemplo, seleccionar las palabras con menos de 5 letras, sin repeticiones:

```
import java.util.*;  
public class Filtro {  
    private static final int LIMITE = 5;  
    public static void main(String[] args) {  
        // Set<String> palabras = new HashSet<String>(List.of(args));  
        Set<String> palabras = new HashSet<String>(Arrays.asList(args));  
        Iterator<String> iter = palabras.iterator();  
        while (iter.hasNext()) {  
            String e = iter.next();  
            if (e.length() >= LIMITE) {  
                iter.remove();  
            }  
        }  
        System.out.println("Palabras seleccionadas: " + palabras.toString());  
    }  
}
```