

# Bloque 3

## Colecciones y Mapas

List, Set, SortedSet y Map

Fundamentos de Programación

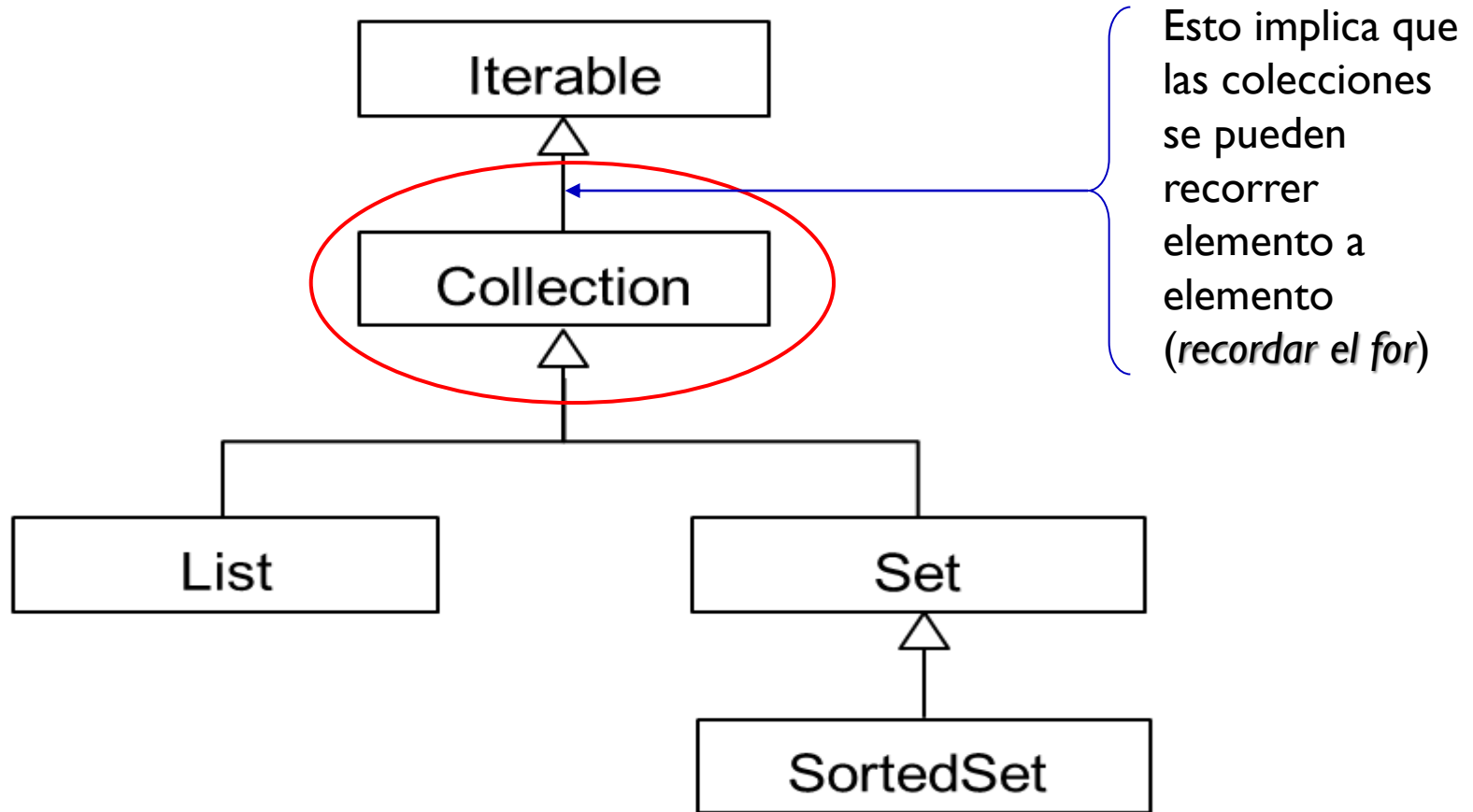
Departamento de Lenguajes y Sistemas Informáticos



2023/24

# Introducción a Collection

## Jerarquía de interfaces.





2023/24

# Introducción a Collection (métodos)

## Métodos de la Interfaz *Collection*:

**Nomenclaturas** que aparecen a continuación:

- **E**: Tipo de los objetos(por ejemplo: *Vuelo*, *Animal*, *Persona*, *Canción*, etc)
- **e**: Un objeto ya creado del tipo E (Por ejemplo: un tren, un evento, un libro, una canción, etc).
- **c**: una colección de objetos. Si aparece como tipo “?” (símbolo comodín) indica que los objetos son del mismo tipo de la colección que invoca al método (de tipo **E**) o un subtipo suyo.

Por ejemplo, a una *Collección* de objetos *Persona* le podemos añadir otra colección de tipo *Estudiante*, siempre que se haya definido *Estudiante* como subtipo de *Persona*.

```
public class Estudiante extends Persona { ..
```



2023/24

# Tipo Collection (recorrido)

## Recorrer una colección mediante for extendido:

Si se dispone de una Colección de objetos de tipo **E**, se recorre con la sentencia **for** con la siguiente sintaxis:

```
Collection<E> miColección = {  
    new ArrayList<E>();  
    new LinkedList<E>();  
    new HashSet<E>();  
    new TreeSet<E>();  
    ---se añaden objetos <E>---  
    for (E e: miColección){  
        tratamiento del objeto e;  
    }
```

**Nota.-** Donde dice *Collection<E>* puede ser, y normalmente es, *List <E>* / *Set <E>* / *SortedSet <E>* según el constructor con el que se cree la colección.



2023/24

# Introducción a Collection (métodos)

## Métodos de Collection

- boolean **add** (*E e*): Añade un objeto “e” de tipo “E” a la colección, devuelve *true* si se añade el objeto, ya que se ha modificado la colección.
- boolean **addAll** (*Collection<? extends E> c*): Añade todos los objetos de la colección “c” a la colección que invoca (*los elementos de c deben ser del mismo tipo o hijos de los elementos de la colección que invoca*). Devuelve *true* si la colección original se modifica.
- void **clear** (): Borra todos los objetos de la colección.
- boolean **contains** (*Object o*): Devuelve *true* si el objeto “o” está en la colección que invoca.



2023/24

# Introducción a Collection (métodos)

- boolean **containsAll** (*Collection*<E> c): Devuelve *true* si la colección que invoca al método contiene todos los objetos de la colección c (*los objetos de c y de la colección que invoca deben ser del mismo tipo*).
- boolean **isEmpty** (): Devuelve *true* si la colección que invoca a método no tiene objetos.
- boolean **remove** (*Object* o): Borra el objeto “o” de la colección que invoca; si no estuviera se devuelve *false* ya que la colección no se ha modificado.
- boolean **removeAll** (*Collection*<E> c): Borra todos los objetos de la colección que invoca que estén en la colección c (*el tipo de objetos de la colección c debe ser igual que la de la colección que invoca*). Devuelve *true* si la colección original se modifica.



2023/24

# Introducción a Collection (métodos)

- boolean **retainAll** (*Collection*<E> c): La colección que invoca se queda con los objetos que están en la colección c (*el tipo de objetos de la colección c debe ser igual que la de la colección que invoca*). Devuelve true si la colección original se modifica.
- int **size** ():Devuelve el número de elementos de la colección que invoca al método.

¡¡Los 10 métodos vistos anteriormente se pueden utilizar para manejar listas, conjuntos y conjuntos ordenados!!

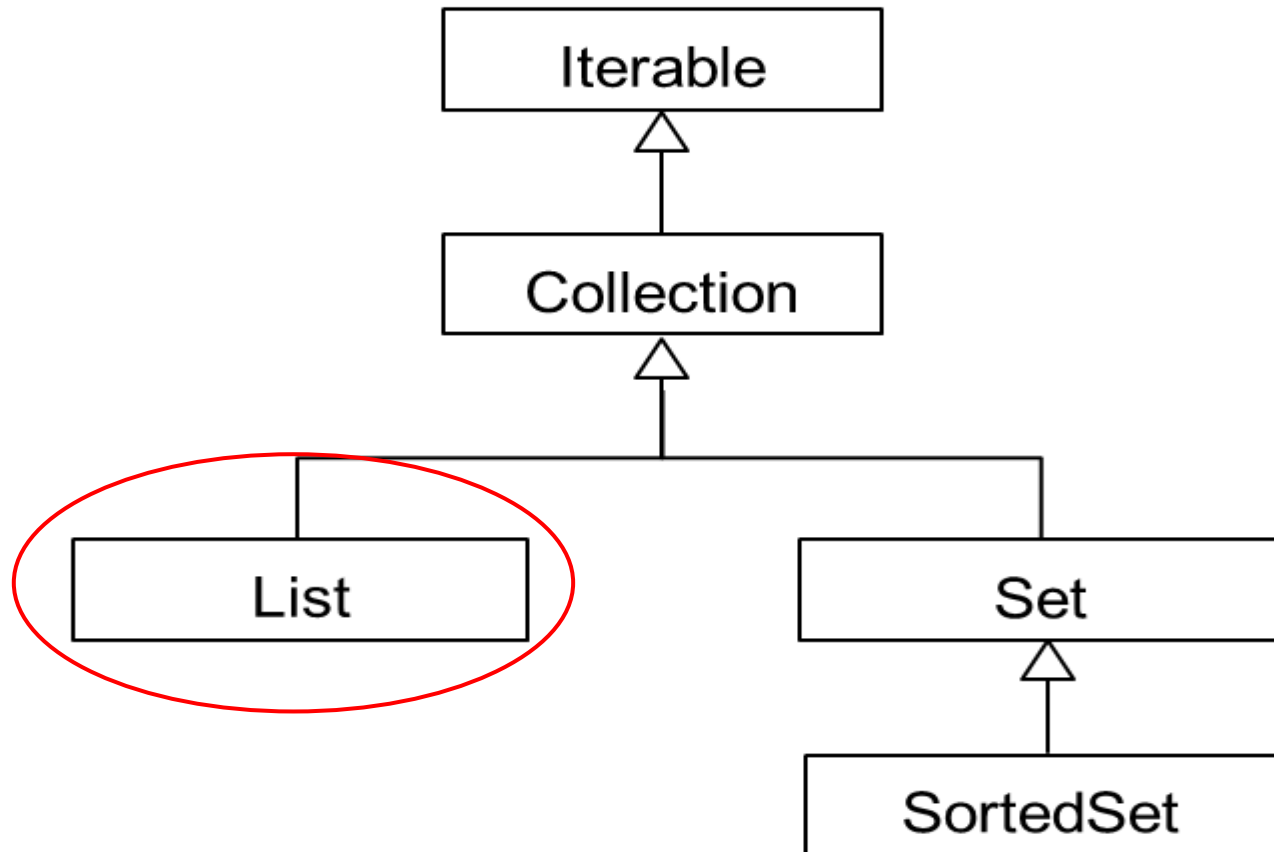
Observar que, en ningún caso, se ha hecho referencia a una posición concreta de un elemento dentro de la colección



2023/24

# Introducción a Collection

## Jerarquía de interfaces.







2023/24

# Tipo List (*Definición y Construcción*)

El tipo *List* es una interfaz del paquete “java.util” que se implementa mediante dos clases (*al igual que nuestro tipo Circunferencia*). Las dos clases que lo implementa son: *ArrayList* y *LinkedList*.

- Construcción: Para crear una lista vacía que almacene objetos de tipo E se utiliza la siguiente sintaxis según la implementación que se elija
  - *List*<E> lista1=new *ArrayList*<E>(); Esta implementación accede rápido a los elementos de la lista pero la inserción o supresión de un objeto en determinada posición de la lista puede ser lento (*pensar en una estantería de libros en la que no puede haber huecos*).
  - *List*<E> lista2=new *LinkedList*<E>(); Esta implementación permite la inserción o supresión de un objeto de manera rápida pero su localización es más lenta que la anterior (*pensar en una cadena de eslabones que para llegar a uno hay que pasar por los anteriores*).



2023/24

# Tipo List (*métodos específicos*)

## Métodos de List

Además de los 10 métodos de *Collection* anteriores las listas tienen *otros métodos* en los que interviene la *posición*. Los 8 más relevantes son:

- boolean **add** (*int p, E e*): Inserta el objeto “e” en la posición “p”, desplazando “*hacia la derecha*” los objetos desde la posición p. Con  $0 \leq p < \text{size}()$ . Si  $p \geq \text{size}()$  lo insertará en la última posición. Devuelve *true* si se añade.
- boolean **addAll** (*int p, Collection<? extends E> c*): Inserta todos los elementos de la colección c en la lista que invoca al método en la posición especificada, desplazando “*hacia la derecha*” los objetos desde la posición p, c.size() posiciones. Con  $0 \leq p < \text{size}()$ . Si  $p \geq \text{size}()$  la insertarán desde la última posición. Los objetos de c deben ser del mismo tipo o hijos de los elementos de la lista que invoca. Devuelve *true* si se añaden.



2023/24

# Tipo List (*métodos específicos*)

- E **get** (*int p*): Devuelve el elemento de la lista que invoca al método en la posición especificada. Con  $0 \leq p < \text{size}()$ .
- int **indexOf** (*Object o*): Devuelve el índice donde se encuentra por primera vez el objeto “o” (si no está devuelve -1).
- int **lastIndexOf** (*Object o*): Devuelve el índice donde se encuentra por última vez el elemento “o” (si no estuviera devuelve -1).
- E **remove** (*int p*): Borra el objeto en la posición p. Con  $0 \leq p < \text{size}()$ . Desplaza “hacia la izquierda” una posición los elementos desde p+1 en adelante. Devuelve el elemento borrado.
- E **set** (*int p, E e*): Reemplaza el objeto de la posición p por el objeto “e”. Con  $0 \leq p < \text{size}()$ . Devuelve el objeto reemplazado.



2023/24

# Tipo List (*métodos específicos*)

- List<E> **subList** (int p, int q): Devuelve una “*vista*” de la porción (una sublista) desde p (incluido) hasta q (sin incluir) [p,q).
  - Una vista significa que si se modifica la sublista se está modificando la original y viceversa. Con  $0 \leq p \leq q \leq \text{size}()$ .

```
List<E> miSublista=new ArrayList<E>();  
miSublista=listOriginal.subList(p,q);  
miSublista.add(E); → Modifica la listaOriginal
```

- Si no se quiere vinculación entre la lista original y la sublista, debe crearse una copia de esta con la siguiente sintaxis:

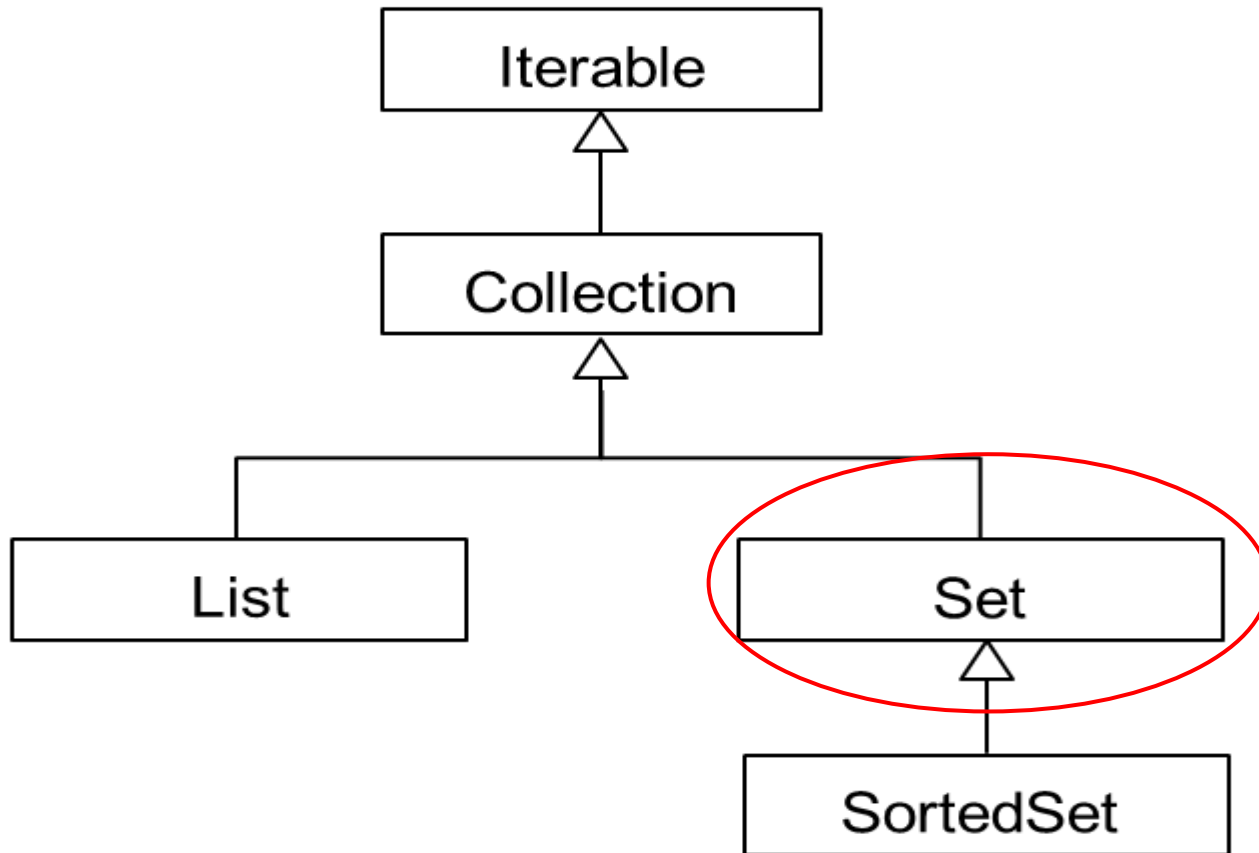
```
List<E> miSublista=new ArrayList<E>();  
miSublista.addAll (listOriginal.subList(p,q));  
miSublista.add(E); → NO modifica la listaOriginal
```



2023/24

# Introducción a Collection

## Jerarquía de interfaces.





2023/24

## Tipo Set (*Definición y Construcción*)

El tipo `Set` es una interfaz del paquete “`java.util`” que se implementa mediante tres clases (*al igual que nuestro tipo `Circunferencia`*). Las tres clases que lo implementa son: *`HashSet` y `LinkedHashSet`*.

- *Construcción*: Para crear un conjunto vacío que almacene objetos de tipo `E` se utiliza la siguiente sintaxis según la implementación que se elija:
  - `Set<E> conj1=new HashSet<E>();` es el más rápido, pero no se sabe el orden en que se devuelve los objetos cuando se recorre el conjunto).
  - `Set<E> conj2=new LinkedHashSet<E>();` devuelve los objetos ordenados por el orden en el que los elementos se van insertando en el conjunto (*cómo en las listas, pero no se puede acceder a las posiciones en que están los objetos*)



2023/24

# Tipo Set (*Métodos*)

## Métodos de Set

- La interfaz `Set` no aporta métodos adicionales de los comunes a `Collection` (*así que ya nos sabemos los 10 métodos que tiene Set*)

Particularidades del tipo `Set`.

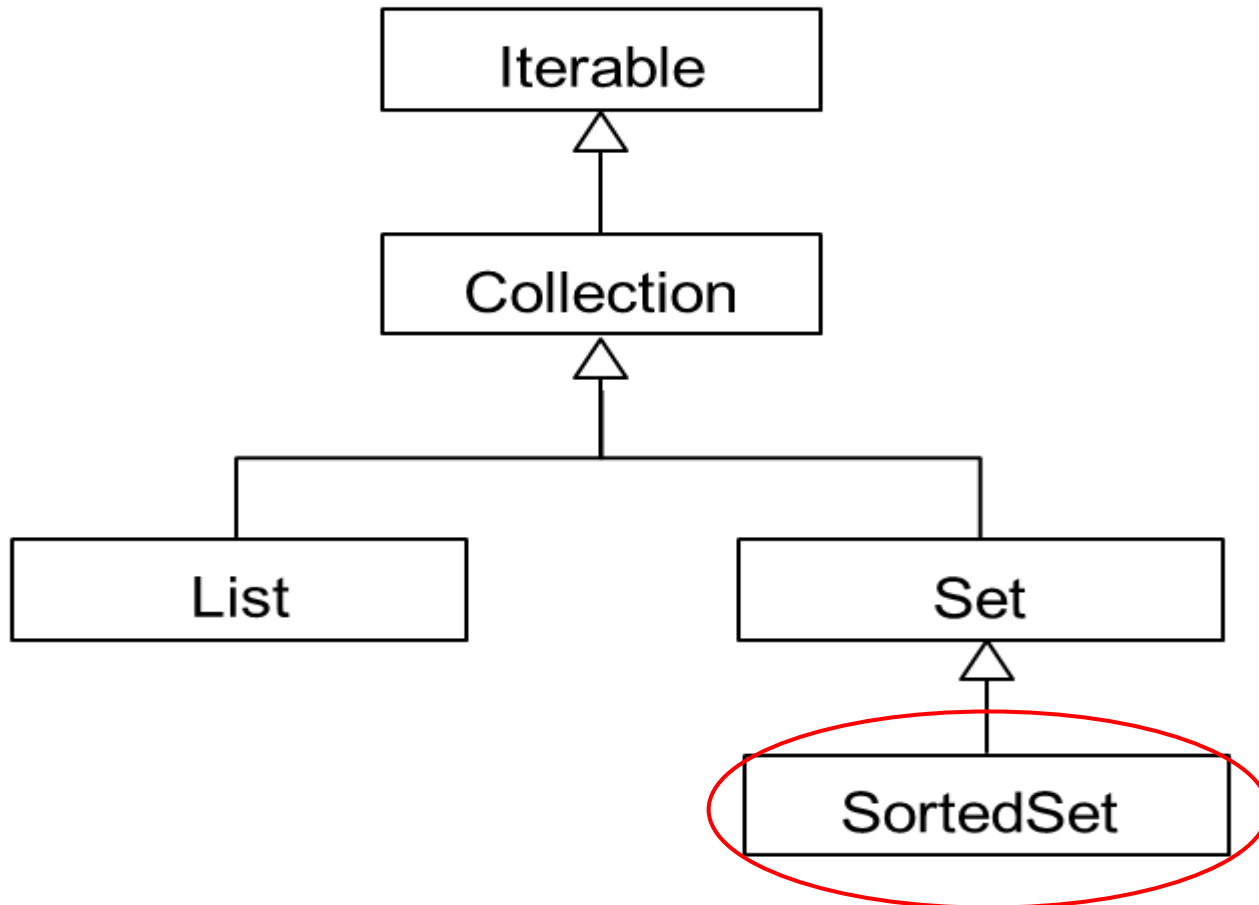
- En los conjuntos *no puede haber objetos repetidos*; Es decir, no puede haber dos objetos iguales según el criterio implementado en `equals` y `hashCode`.
- Las operaciones *`addAll`*, *`retainAll`* y *`removeAll`* se pueden identificar con la unión ( $\cup$ ), intersección ( $\cap$ ) y diferencia ( $-$ ) de conjuntos, respectivamente; la operación *`contains`* equivale a la pertenencia de un elemento en un conjunto ( $\in$ ); la operación *`containsAll`* se corresponde con la de subconjunto ( $\subseteq$ ).



2023/24

# Introducción a Collection

## Jerarquía de interfaces.



► Colecciones: Listas, Conjuntos y Conjuntos Ordenados





2023/24

## Tipo SortedSet (*Definición y Construcción*)

El tipo *SortedSet* es una interfaz subtipo de *Set* definida en el paquete “java.util” que se implementa mediante *TreeSet*,

*Construcción*: Para crear un conjunto ordenado vacío que almacene objetos de tipo *E* se utiliza la siguiente sintaxis según la implementación que se elija:

- *SortedSet*<*E*> conjuntoOrdenado=new *TreeSet*<*E*>(); cuando se recorre el conjunto, se devuelve los objetos ordenados por el orden natural (el que se establece en *compareTo*) de los elementos que almacena.

¡Ojo! para que haya orden natural *el tipo E tiene que implementar la interfaz Comparable* y por tanto, el método *compareTo()*..



2023/24

## Tipo SortedSet (*Ordenes alternativos*)

Para mantener los objetos ordenados en un *SortedSet* por un *criterio distinto del orden natural* es necesario utilizar el mismo constructor *TreeSet* pero pasando como parámetro un objeto de una Clase comparadora (la estudiaremos más adelante, cuando veamos las interfaces funcionales. En concreto la interfaz *Comparator*). De esta forma, si *cmp* es un objeto “comparador” que ordena los objetos del tipo E por un determinado criterio, la sintaxis será

- *Set<E>* conjuntoOrdenado=new *TreeSet*<E>(*cmp*); cuando se recorre el conjunto, se devuelve los objetos ordenados por el criterio inducido por la clase comparadora).



2023/24

# Introducción a Collection

## RESUMEN.

- **Collection** tiene tres subinterfaces: **List**, **Set** y **SortedSet**.
- **Collection** tiene **10 métodos** (por tanto, son también métodos de las tres subinterfaces).
- **List** tiene **8 métodos más habituales** (no valen para Set ni SortedSet, porque usan la posición en la colección y, en los Set y SortedSet, no existe el concepto de posición)
- **Las clases** que implementan dichas interfaces son:
  - List--> **ArrayList** o **LinkedList**
  - Set--> **HashSet** o **LinkedHashSet**.
  - SortedSet --> **TreeSet**

Una observación sobre los métodos **addAll**, **removeAll**, **retainAll** y **containsAll** de Collection: cuando se usan para objetos Set o SortedSet son la **unión**, **diferencia**, **intersección** y **subconjunto** de conjuntos.

► Colecciones: Listas, Conjuntos y Conjuntos Ordenados



2023/24

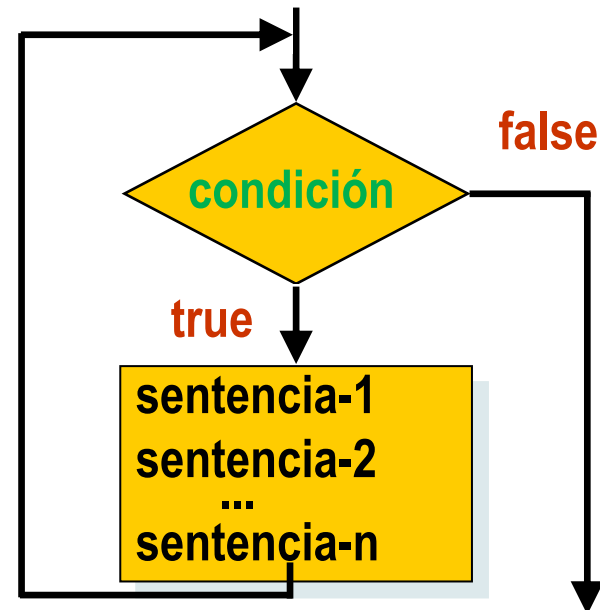
# Introducción a Collection (recorridos)

Hemos visto que Collection es un subtipo de Iterable. Ello quiere decir que las colecciones se pueden recorrer.

Las sentencias que permite recorrer son *while* y *for*  
*while*

- Ejecuta un bloque de sentencias mientras determinada condición evalúa a cierto (*true*).
- Sintaxis:

```
while (condición)  
{  
    sentencia-1;  
    sentencia-2;  
    ...  
    sentencia-n;  
}
```





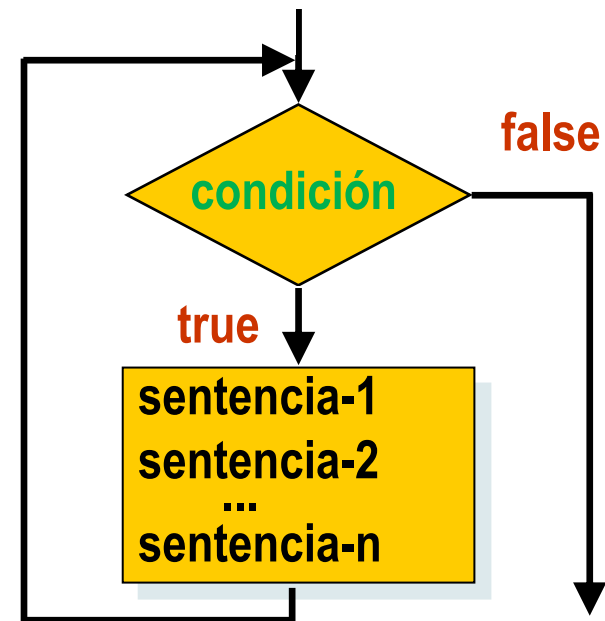
2023/24

# Introducción a Collection (recorridos)

## *while (ejemplo)*

- Sumar los números pares menores o iguales que 100

```
int suma=0;
int n=1
while (n<=100) {
    if (n%2==0){
        suma+=n;
    }
    n++;
}
```





2023/24

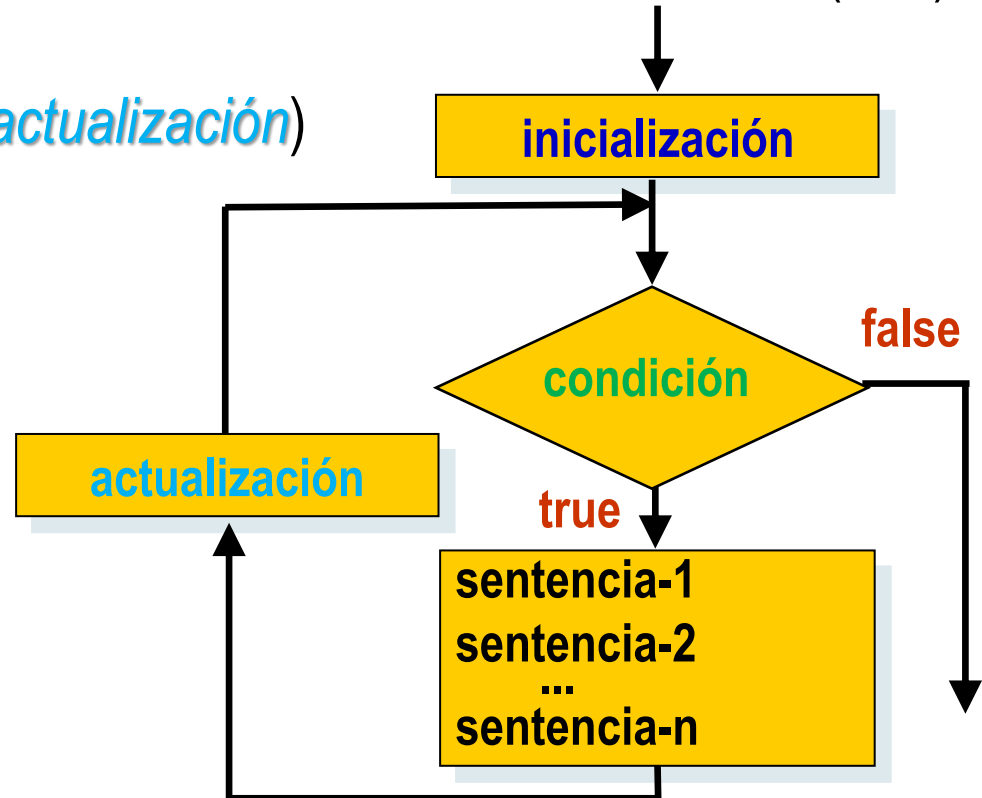
# Introducción a Collection (recorridos)

## *for-clásico*

- *a) Ejecuta* la inicialización y evalúa la condición y *b) ejecuta* bloque de sentencias y la actualización mientras la condición evalúa a cierto (*true*).

- *Sintaxis:*

```
for (inicialización; condición; actualización)  
{  
    sentencia-1;  
    sentencia-2;  
    ...  
    sentencia-n;  
}
```





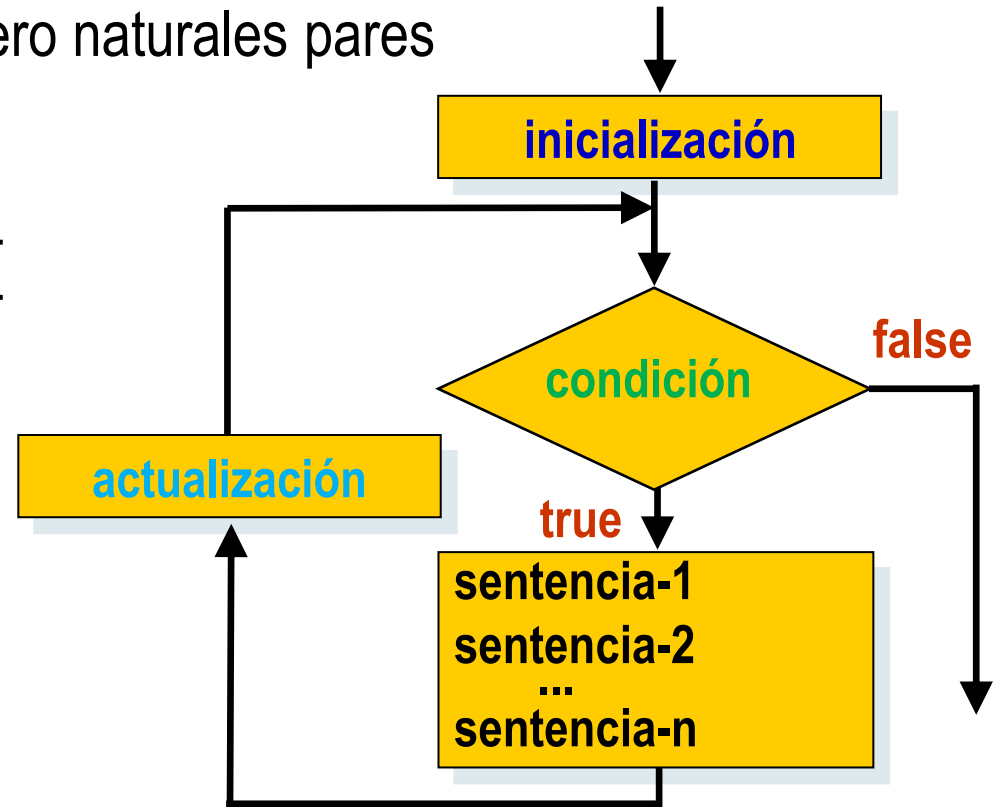
2023/24

# Introducción a Collection (recorridos)

## *for-clásico (ejemplo)*

- Sumar los 100 primeros números naturales pares

```
int suma=0;  
for (int n=1; n<=200; n++) {  
    if (n%2==0){  
        suma+=n;  
    }  
}
```





2023/24

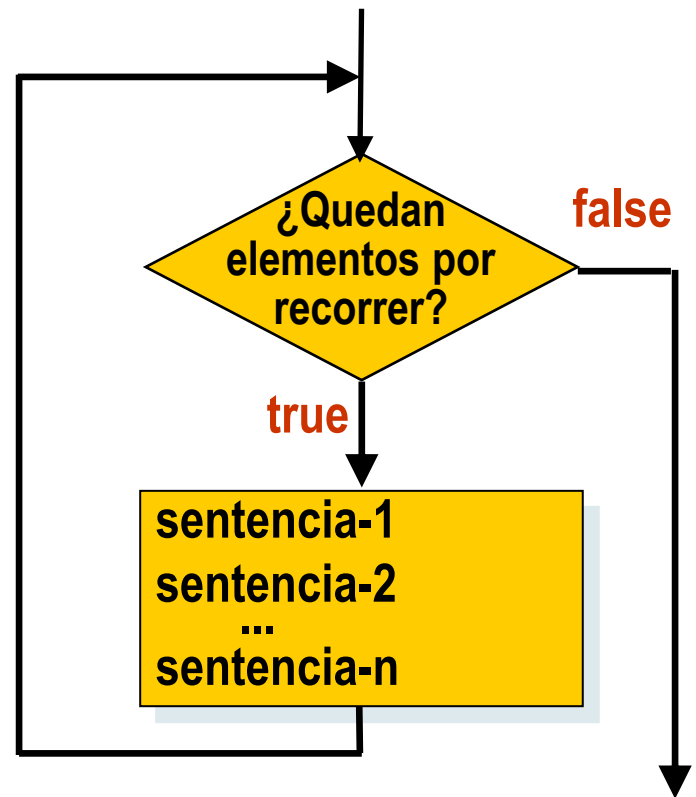
# Introducción a Collection (recorridos)

## *for-each o for-extendido*

- *Ejecuta* el bloque de sentencias mientras la colección tiene elementos pendientes de recorrer.

- *Sintaxis:*

```
for (Tipo variable: colección) {  
    sentencia-1;  
    sentencia-2;  
    ...  
    sentencia-n;  
}
```







2023/24

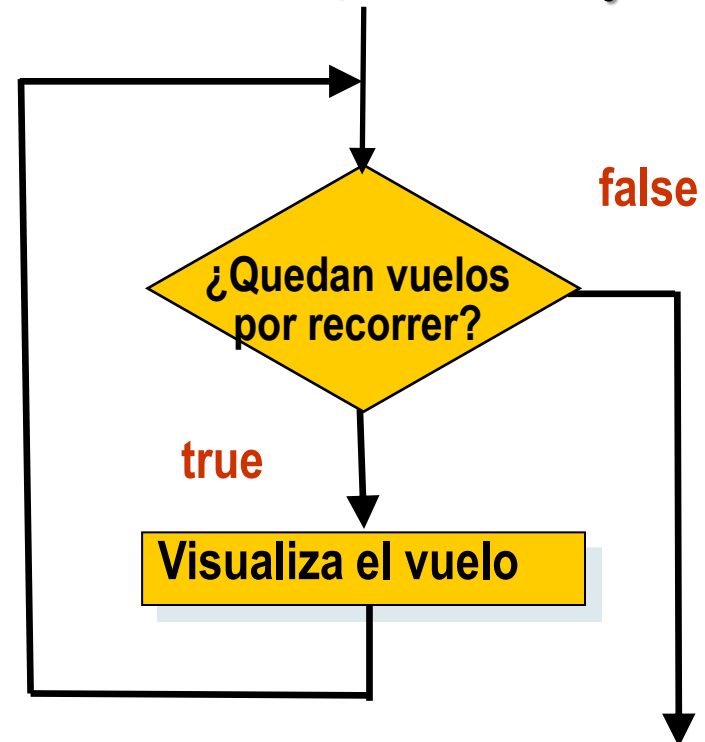
# Introducción a Collection (recorridos)

## *for-each o for-extendido (ejemplo)*

- Visualizar una lista de vuelos `List<Vuelo> misVuelos`, uno debajo de otro.

- Sintaxis:

```
for (Vuelo v: misVuelos) {  
    System.out.println(v);  
}
```





2023/24

# Ejercicio Aeropuerto

---

1. En la carpeta *doc* copie el *enunciadoAeropuerto02.pdf*
2. Realice los apartados 7 y 8

**Nota.**- Este proyecto irá creciendo hasta llegar aproximadamente hasta el enunciado 09 o 10, ya que servirá de base para practicar los conceptos que se expliquen en las clases de teoría.

Por lo que es importante llevarlo al día.