Colecciones de Java

Sitio: Centros - Cádiz
Curso: Programación
Libro: Colecciones de Java

Imprimido por: Barroso López, Carlos
Día: martes, 21 de mayo de 2024, 23:39

Tabla de contenidos

1. Tipos

Los principales tipos de colecciones de Java son:

1. Set

La interfaz Set define un conjunto **SIN elementos duplicados**, donde el orden de inserción de los elementos no es relevante. Esta interfaz contiene únicamente los métodos heredados de *Collection*.

Método equals()

Un objeto está duplicado (o1 = o2) si **o1.equals(o2)** es **true**. Se debe redefinir este método para utilizar en objetos de clases propias.

Implementaciones:

1.1. HashSet

Características:

- Proporciona el **mejor rendimiento**.
- No garantiza **ningún orden** a la hora de realizar iteraciones.

Almacena los elementos en una tabla hash. Proporciona tiempos constantes en las operaciones básicas (*búsqueda, inserción y borrado*).

Método hashCode()

Proporciona un código hash asociado al objeto. Se debe redefinir para utilizar en objetos de clases propias.

1.2. TreeSet

Características:

- Menor rendimiento que HashSet.
- Mantiene los **elementos ordenados** en función de sus valores.

Almacena los elementos en una estructura de árbol. Garantiza un rendimiento de log(N) en las operaciones básicas.

Los objetos deben implementar la interfaz **Comparable<T>**, la cual incluye el método:

compareTo()

Compara el objeto con otro pasado por parámetro. Devuelve:

- entero negativo: si el objeto es **menor** (al pasado por parámetro)
- 0: si son **iguales**
- entero positivo: si el objeto es **mayor**

1.3. LinkedHashSet

Almacena los elementos en función del orden de inserción.

2. List

La interfaz List define una **sucesión de elementos**, admitiendo el acceso posicional a dichos elementos y elementos duplicados.

Implementaciones:

2.1. ArrayList

Se basa en un **array redimensionable** (dinámico) que aumenta su tamaño según crece la colección de elementos. Es la implementación que mejor rendimiento tiene en la mayoría de situaciones.

2.2. LinkedList

Implementación basada en una **lista doblemente enlazada** de los elementos. Mejora el rendimiento en ciertas situaciones.

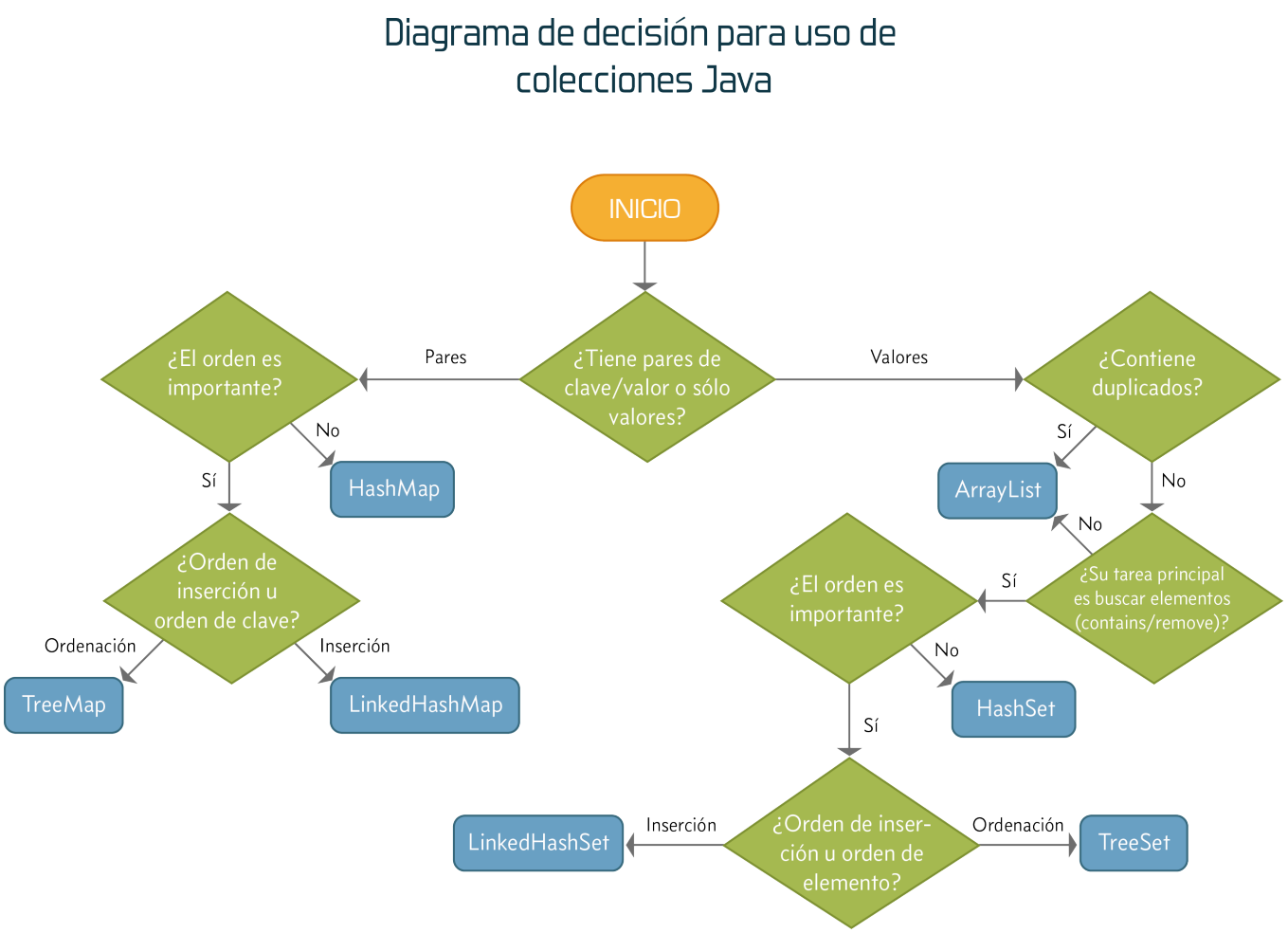
3. Map

La interfaz Map define un conjunto de **pares (clave, valor)**, SIN claves duplicadas.

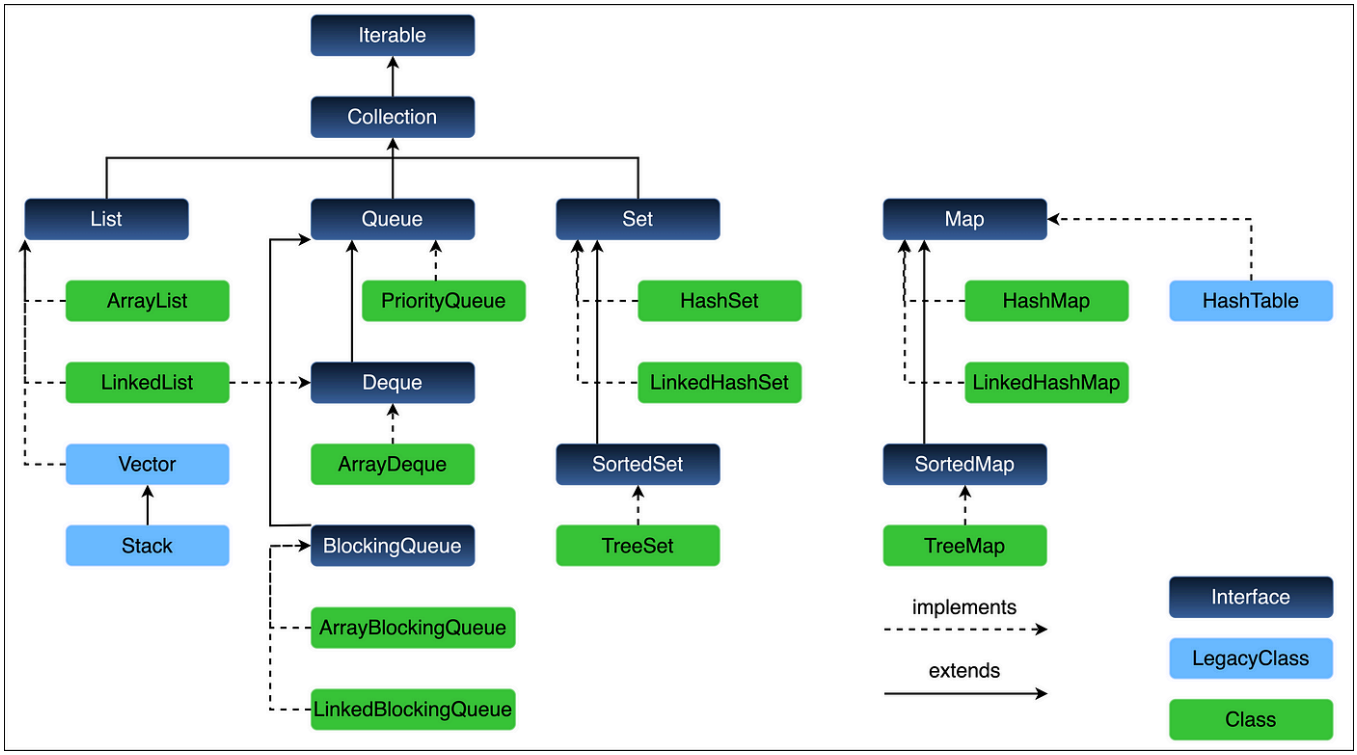
Existen los mismos tipos de implementaciones que para la interfaz Set, que son:

- **HashMap**
- **TreeMap**
- **LinkedHashMap**

1.1. Diagrama de decisión



1.2. JCF (Java Collections Framework)



2. Creación de colecciones

1. Declaración e instanciación

Sintaxis:

```
NombreColeccion<NombreClase> identificador = new NombreColeccion<NombreClase>();
```

```
NombreColeccion<NombreClase> identificador = new NombreColeccion<>();  // equivalente
```

Ejemplo:

```
ArrayList<String> lista = new ArrayList<>();
lista.add("Jose");
System.out.println(lista);
```

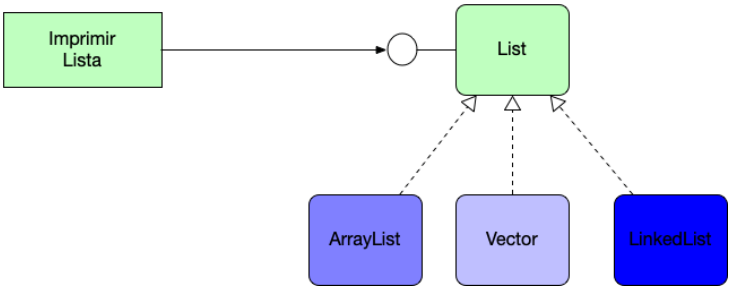
2. Uso de tipos genéricos

A la hora de utilizar colecciones, es muy recomendable utilizar tipos genéricos para dotar al código de mayor **flexibilidad**.

Ejemplo:

```
/*
  Función que muestra por pantalla los elementos de una lista
*/
public static void muestraLista(List<String> lista) {  // Usa el tipo genérico List (interface)
  for (String cadena : lista) {
    System.out.println(cadena);
  }
}
```

```
/* Programa principal */
List<String> lista = new ArrayList<>();
lista.add("Hola");
lista.add("Mundo");
muestraLista(lista);
/* --- */
List<String> lista2 = new LinkedList<>();
lista2.add("Top");
lista2.add("Programmer");
muestraLista(lista2);
```

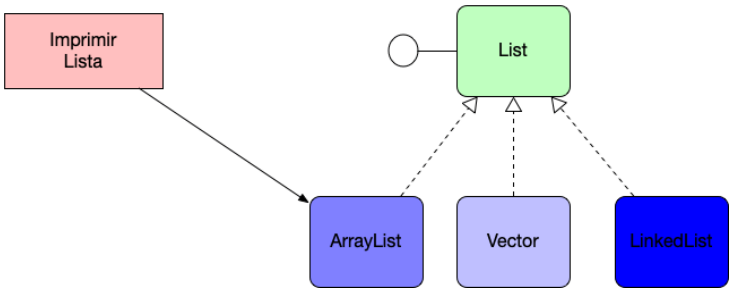


Sin usar tipos genéricos:

```
public static void muestraLista(ArrayList<String> lista) { ... }
```

...

```
muestraLista(lista2);  // ERROR
```



3. Clases contenedores (wrappers)

En Java todo es un objeto, salvo una excepción, los tipos primitivos de datos (*boolean, char, int, float, ...*).

Estos tipos de datos se mantienen por eficiencia, pero Java dispone de una **clase contenedora** para cada uno de ellos.

1. Definición y tipos

Un ***contenedor*** en Java es una clase especial que almacena el valor de un tipo primitivo, disponiendo de métodos para trabajar con dicho tipo de dato.

Tabla de clases contenedoras:

Tipo primitivo	**Contenedor**
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

2. Características

2.1. Objetos inmutables

Los objetos de estas clases contenedoras son **inmutables** (al igual que los objetos String), esto quiere decir que no se puede modificar el estado (atributos) de un objeto.

Por tanto, si una variable de este tipo cambia de valor, directamente se crea un objeto nuevo.

Ejemplo:

```
Integer a = 1;  // se crea un objeto Integer que almacena el valor entero 1
```

```
a = 2;  // se crea un nuevo objeto Integer con el valor 2
```

2.2. Embalaje/desembalaje automático (Autoboxing/Autounboxing)

Autoboxing

A una variable de clase contenedora se le puede asignar un valor de tipo primitivo.

Ejemplo:

```
int num = 5;
Integer a = num;
```

Autounboxing

A una variable de tipo primitivo se le puede asignar un objeto de una clase contenedora.

Ejemplo:

```
Integer a = 5;
int num = a;
```

3. Algunos métodos

Contenedor.valueOf(valor)

Devuelve un contenedor que contiene el valor (de tipo primitivo) pasado por parámetro.

Ejemplo:

```
Integer.valueOf(2)  // devuelve un objeto Integer que contiene el valor 2
```

Contenedor.parseXXX(String cadena)

Devuelve un contenedor que contiene el valor extraíado de la cadena pasada por parámetro.

Ejemplo:

```
String s = "12";
```

```
Integer i = Integer.parseInt(s);
```

4. Igualdad de objetos

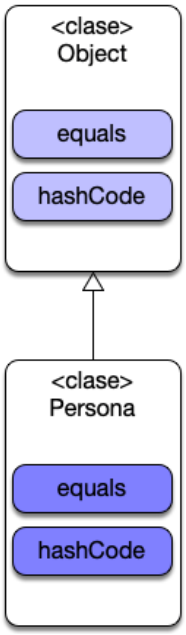
1. Métodos equals() y hashCode()

Para determinar si dos objetos son iguales, es decir, representan el mismo objeto, se utilizan los métodos:

equals(): indica si dos objetos son iguales.

hashCode(): devuelve un código asociado al objeto.

Son métodos heredados de la clase Object (clase raíz de Java), y que por tanto, cualquier objeto dispone de ellos.



Será necesario sobrescribir (**@override**) dichos métodos para adaptarlos a cada clase en particular.

2. Uso en estructuras "hash"

El método hashCode() es utilizado por estructuras que almacenan sus elementos mediante tablas hash, junto con el método equals(), para comprobar si dos objetos son iguales.

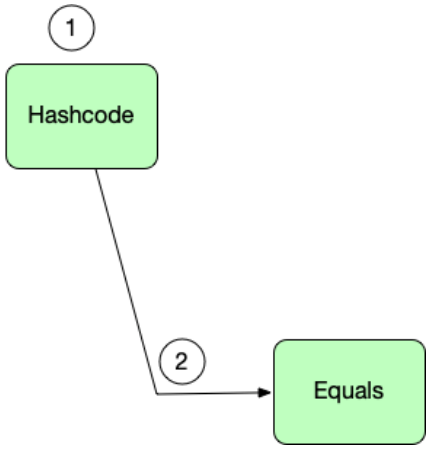
Funcionamiento:

1. Se invoca al método hashCode() de ambos objetos y se comparan.

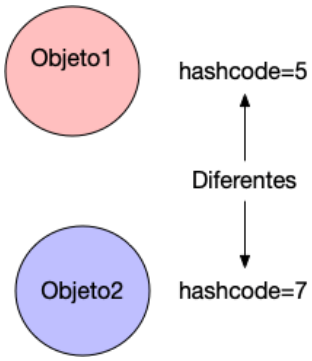
Un mismo código hash significa que están en el **mismo grupo** y pudieran ser iguales (aún no es seguro).

2.a. Si ambos objetos pertenecen al mismo grupo:

Se invoca al método equals() para comprobar si realmente representan al mismo objeto o no.



2.b. Si sus códigos hash son diferentes, no se sigue comparando, y se consideran objetos diferentes.



4.1. Método equals()

1. Definición y uso

Indica si el objeto pasado por parámetro representa el mismo objeto, es decir, **ambos objetos se consideran iguales a nivel semántico** o de reglas de negocio. Para ello, será necesario comparar algunos o todos sus atributos.

Implementación:

```
@Override
public boolean equals(Object obj) {
  if (this == obj)  // misma referencia
    return true;
  if (obj instanceof NombreClase) {  // misma clase
    NombreClase p = (NombreClase) obj;  // casting
    /*
      Comparación de atributos
    */
    return true;
  }
  return false;
}
```

Llamada:

```
o1.equals(o2)  // true si o1 y o2 son iguales, false en caso contrario
```

Ejemplo:

Dos objetos Persona son iguales si tienen el mismo "nombre":

Persona.java

```
@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj instanceof Persona) {
    Persona p = (Persona) obj;
    if ( this.nombre.equals(p.nombre) )
      return true;
  }
  return false;
}
```

Llamada:

```
Persona p1 = new Persona("Jose");
Persona p2 = new Persona("Pablo");
p1.equals(p2)  // false
```

```
Persona p3 = new Persona("Jose");
p1.equals(p3)  // true
```

2. Operador "==" vs "equals()"

Operador ==

Compara las **referencias** de las **variables**, por tanto, si las referencias son distintas (apuntan a objetos diferentes) devuelve false, aunque ambos objetos tengan el mismo contenido.

equals()

Compara el **contenido** (atributos) de los **objetos**, según la implementación realizada para cada clase. Si ambos objetos son el mismo (misma referencia), siempre devolverá true.

Ejemplo:

```
Persona p4;
p4 = p1;       // dos variables que apuntan a la misma referencia (mismo objeto)
p1.equals(p4)  // SIEMPRE devuelve true
p1 == p4       // true (las variables contienen la misma referencia)
```

```
p1.equals(p3)  // true (los objetos tienen el mismo contenido)
p1 == p3       // ¡¡FALSE!! son objetos (referencias) DIFERENTES
```

2.1. "Peculiaridad" de la clase String

Ejemplo:

```
String s1 = "Jose";
String s2 = "Pablo";
String s3 = "Jose";
String s4 = new String("Jose");
```

```
s1.equals(s2)  // false
s1.equals(s3)  // true
s1.equals(s4)  // true
```

```
s1 == s2  // false
s1 == s3  // ???
s1 == s4  // ???
```

Aclaración

Por eficiencia, Java asigna el mismo objeto a las variables de tipo String que se les asigne el mismo literal. No ocurre así cuando se crea el objeto con new o se opere con ellos.

Por tanto:

```
s1 == s3  // true
```

```
s1 == s4  // false
```

Ejemplo:

```
String a = "abc";
String b = "abcd";
a += "d";
```

```
a == b  // false
```

4.2. Método hashCode()

1. Definición y uso

Devuelve un código hash asociado al objeto.

Sintaxis:

```
@Override
public int hashCode() {
  /* Calcula y devuelve un código hash para los objetos de la clase */
}
```

Llamada:

```
obj.hashCode()  // devuelve el código hash asociado al objeto "obj"
```

Ejemplo:

El código hash de un objeto Persona es el hash de su nombre:

```
@Override
public int hashCode() {
  return this.nombre.hashCode();  // hash de la clase String
}
```

Llamada:

```
Persona p1 = new Persona("Jose");
```

```
p1.hashCode();
```

2. Generar códigos hash

El paquete de utilidades de java dispone del método estático **Objects.hash()**, el cual genera un código hash a partir de los parámetros proporcionados.

Ejemplo:

```
import java.util.Objects;
```

```
System.out.println( Objects.hash("Jose", 25, "casa", "1234") );
```

4.3. Ejemplo

Persona.java

```
public class Persona {
  // Atributos
  private String dni;
  private String nombre;
  private int edad;

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

  /*
    toString(): devuelve una cadena con las datos de un objeto Persona.
  */
  @Override
  public String toString() {
    return "Persona -> DNI: "+this.dni+" Nombre: "+this.nombre+" Edad: "+this.edad;
  }

  /*
    hashCode(): devuelve un código hash para un objeto Persona.
    Para ello, genera el hash del atributo DNI.
  */
  @Override
  public int hashCode() {
    return this.dni.hashCode();  // hash de la clase String
  }

  /*
    equals(): comprueba si dos objetos Persona son iguales (son la misma persona).
    Para ello, comprueba si tienen el mismo DNI (además de otras comprobaciones).
  */
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj instanceof Persona) {
      Persona p = (Persona) obj;
      if ( this.dni.equals(p.dni) )
        return true;
    }
    return false;
  }
}
```

5. Ordenación de objetos

1. Método sort()

El método sort() permite ordenar los elementos de una lista.

Sintaxis:

```
unaLista.sort(null);  // ordena los elementos según su orden natural
```

Ejemplo:

```
List<String> nombres = new ArrayList<>();
nombres.add("Paco");
nombres.add("Miguel");
nombres.add("Lidia");
System.out.println(nombres);  // [Paco, Miguel, Lidia]
nombres.sort(null);
System.out.println(nombres);  // [Lidia, Miguel, Paco]
```

2. ¿Cómo ordenar objetos de clases propias?

Ejemplo:

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona("4444", "Paco", 33);
personas.add(new Persona("2323", "Miguel", 15);
personas.add(new Persona("1234", "Lidia", 20);
pesonas.sort(null);  // ¿¿Qué ocurre??*
```

* **¡ERROR!**: Java NO sabe como ordenar los objetos Persona.

Para poder ordenar los objetos de una clase se deben poder COMPARAR.

Dos formas:

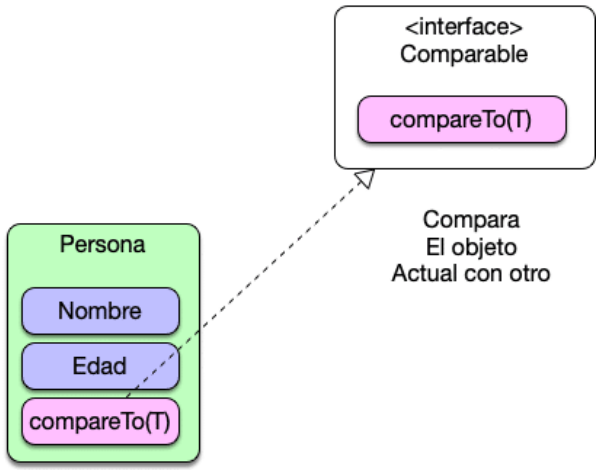
i. Implementación de la **interfaz Comparable<T>** (Orden natural).

ii. Uso de **comparadores**.

5.1. Interfaz Comparable

Una forma de poder comparar los objetos de una clase es que ésta implemente la interfaz Comparable<T>.

La implementación de esta interfaz obliga a la clase a sobrescribir el método **compareTo()**. A esta ordenación se conoce como el **orden natural** de los objetos.



1. Implementación

Sintaxis:

```
public class NombreClase implements Comparable<NombreClase>
```

Ejemplo:

```
public class Persona implements Comparable<Persona> { ... }
```

2. Método compareTo()

El método compareTo() devuelve un valor entero:

- **< 0**: si el objeto **es menor** que el objeto pasado por parámetro.
- **= 0**: si el objeto **es igual** que el objeto pasado por parámetro.
- **> 0**: si el objeto **es mayor** que el objeto pasado por parámetro.

Sintaxis:

```
@Override
public int compareTo(NombreClase obj) {
  /*
    (this < obj) -> return < 0
    (this = obj) -> return 0
    (this > obj) -> return > 0
  */
}
```

Ejemplo:

```
/*
  compareTo(): compara dos objetos Persona según el atributo edad.
*/
@Override
public int compareTo(Persona p) {
  return this.edad - p.edad;
}
```

5.2. Comparadores (interfaz Comparator)

En ocasiones es necesario ordenar los elementos de una lista de diferentes formas.

Ejemplo:

Ordenar la lista de objetos Persona por el "dni" o por el "nombre", en lugar de usar el orden natural (por la "edad").

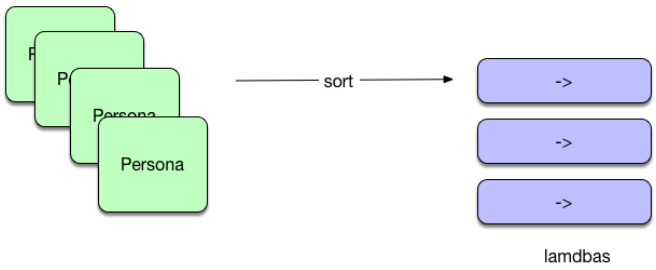
Para ello, Java permite definir un comparador (**Comparator**) para cada criterio de ordenación que se necesite.

1. Definición y uso

Sintaxis:

```
Comparator<T> nombreComparador = (expresión lambda);
```

```
unaLista.sort(nombreComparador);
```



Ejemplo:

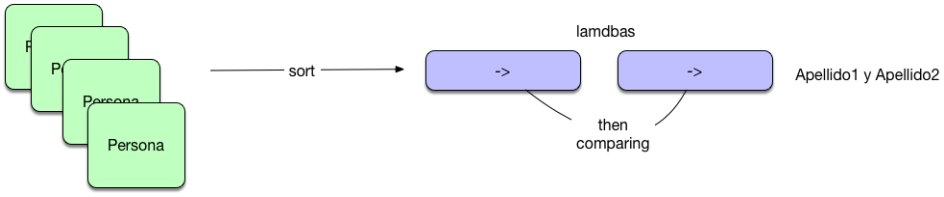
Ordenar la lista de objetos Persona por el "nombre":

```
Comparator<Persona> compNombre = (p1,p2)->p1.getNombre().compareTo( p2.getNombre() );
```

```
pesonas.sort(compNombre);  // recibe como parámetro el comparador
```

2. Varios niveles de ordenación

Es posible encadenar varias expresiones lambda a través del método **thenComparing()**, y establecer de este modo varios niveles de ordenación.



Ejemplo:

Ordenar la lista de objetos Persona por el "nombre" (primer nivel) y por la "edad" (segundo nivel):

```
Comparator<Persona> compNombreEdad = compNombre.thenComparing( (p1,p2)->p1.getEdad().compareTo( p2.getEdad() ) );
```

```
pesonas.sort(compNombreEdad);
```

5.3. Ejemplo

Persona.java (ampliación)

```
public class Persona implements Comparable<Persona> {
  ...
  // Métodos getters
  public String getDni() {
    return this.dni;
  }
  public String getNombre() {
    return this.nombre;
  }

  /*
    compareTo(): compara dos objetos Persona según el atributo edad.
  */
  @Override
  public int compareTo(Persona p) {
    return this.edad - p.edad;
  }
}
```

PruebaPersona.java

```
public class PruebaPersona {
  public static void main(String[] args) {
    List<Persona> personas = new ArrayList<>();
    personas.add(new Persona("4444", "Paco", 33);
    personas.add(new Persona("2323", "Miguel", 15);
    personas.add(new Persona("1234", "Lidia", 20);
    System.out.println(personas);
    personas.sort(null);  // ordena según el orden natural (por "edad")
    System.out.println(personas);
    personas.sort( (p1,p2)->p1.getNombre().compareTo( p2.getNombre() ) );  // ordena por "nombre"
    System.out.println(personas);
    Comparator<Persona> compDni = (p1,p2)->p1.getDni().compareTo( p2.getDni() );
    pesonas.sort(compDni);  // ordena por "dni"
    System.out.println(personas);
  }
}
```

6. Métodos

Métodos comunes

MÉTODO	DESCRIPCIÓN
Modificación	
add(elemento)	Añade un elemento si no existe previamente. Devuelve true si lo inserta. (boolean)
remove(elemento)	Elimina un elmento. Devuelve true si existe, y por tanto, lo elimina. (boolean)
clear()	Elimina todos los elementos de la colección.
Consulta	
size()	Devuelve el tamaño de la colección. (int)
isEmpty()	true si la colección está vacía. (boolean)
contains(elemento)	true si el elemento está en la colección. (boolean)
Recorrido	
iterator()	Devuelve un iterador para la colección.
Copia	
clone()	Devuelve una copia de la colección (los objetos no se copian)