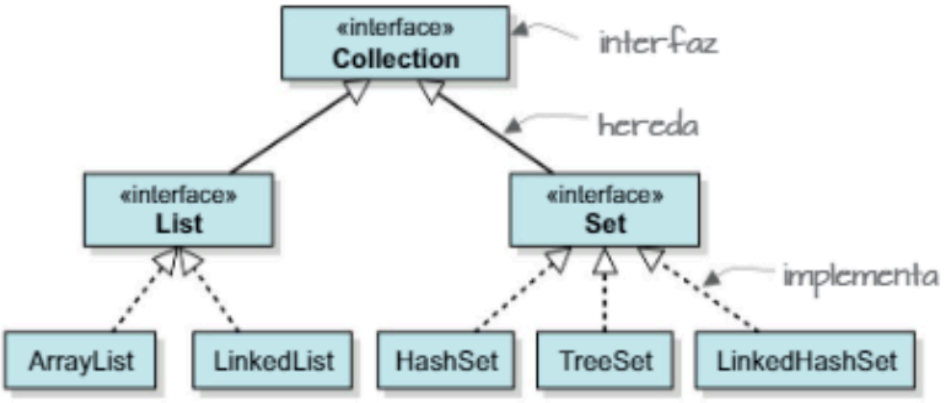
UD6.1. Colecciones

1.Introducción

Cuando no sabemos el número de elementos a guardar, los arrays no son la solución adecuada, ya que su tamaño debe permanecer fijo una vez creados.
Al redimensionarlos, se crea uno nuevo y se copian todos los datos desde el array antiguo → sobrecarga de memoria.

La solución son las estructuras dinámicas de datos. Permiten insertar, eliminar, y cambiar el tamaño de la estructura, sin modificar los datos ya añadidos, en tiempo de ejecución.
Java proporciona unas estructuras dinámicas que comparten un conjunto de métodos en la **interfaz Collection**. Todas la implementan, aunque de distinta forma.



Colección: objeto que sirve para agrupar un conjunto de objetos, los **elementos**, que generalmente guardan una relación entre ellos.

Los **métodos** de las colecciones nos permiten hacer diferentes cosas con sus elementos, *(inserción, eliminación, búsqueda, ordenación…)*
Hay colecciones de diferentes tipos, adaptadas a distintos fines, así que hay un **framework con una estructura jerárquica de interfaces** que se implementan en distintas clases, con algoritmos que permiten la manipulación de los datos almacenados en las colecciones.

Tipos fundamentales de estructuras:

- **Listas**: sucesiones de datos que pueden estar repetidos y cuyo orden puede ser relevante.
  "sustituyen" a las tablas, con la diferencia de que podemos insertar o eliminar datos en ellas sin limitaciones de espacio. → *interfaz List*

- **Conjuntos**: lo importante es la pertenencia, no el orden. No hay repetidos. → *interfaz Set*
- **Mapas o diccionarios**: dentro del framework, aunque no implementan la interfaz Collection. Guardan datos identificados por claves que no se repiten. → *interfaz Map, **no** hereda de Collection.*

---

1. Tipos parametrizados o genéricos

Una particularidad de **Collection** es que trabaja con **tipos genéricos** de datos.

El uso de los tipos genéricos se debe a la **necesidad de disponer de clases, interfaces o métodos que se puedan usar con muchos tipos de datos distintos**, pero haciendo *comprobaciones de tipo en tiempo de compilación*.

Ejemplos: **compareTo()** y **compare()**, de las interfaces **Comparable** y **Comparator**.
Están pensadas para comparar objetos de cualquier clase, y hay que implementar Comparable para cualquier clase de objetos que insertemos en cualquier tabla o colección que pretendamos ordenar.

Antes de los genéricos compareTo() recibía como parámetro una variable de tipo Object, que es la clase más general de todas, y se casteaba.
Problema: incomodidad del casting + un tipo erróneo del parámetro **no** se manifiesta como un **error de compilación**, sino como un **error de ejecución**, lanzando una **excepción**. Los genéricos permiten la implementación con tipos tan generales como Object, pero comprobándolos y detectando errores antes de ejecutar el programa.

1.1. Clases con parámetros genéricos

Si **no tenemos control sobre el tipo del objeto guardado**, el **compilador** *no hace comprobaciones de tipos y se pueden guardar objetos de distintas clases mezclados*.
Podemos escribir:

```
ContenedorO c = new ContenedorO();
c.guardar (7);
c.guardar ("roca");
Double x = (Double) c.extraer(); //error de ejecución
```

Compila sin problema, aunque sabemos que el Object extraído es una cadena y no un Double. El error se producirá en la última línea del programa → **ClassCastException.**

En vez de implementar un `Contenedor` para `Integer`, para `Double` … se recurre a genéricos.

T es el tipo de datos que se va a usar en la clase en cada declaración, y tiene que ser una **clase o interfaz**, nunca un tipo primitivo.
La expresión **<>** se llama **operador diamante.**

Ahora podemos crear un Contenedor para enteros con la siguiente sintaxis:

```
Contenedor<Integer> c = new Contenedor<Integer>();
Contenedor<Integer> c = new Contenedor<>();
```

El segundo Integer puede no ponerse, el compilador puede verlo desde la izquierda.

En este Contenedor, **el compilador comprueba el tipo del valor** que pasamos al método `guardar`(), que tiene que ser `Integer`. Cualquier otro tipo habría dado un error en la compilación.

La misma clase genérica nos sirve para crear un Contenedor de números reales o Clientes.

```
Contenedor<Double> c1 = new Contenedor<>();
Contenedor<Cliente> c2 = new Contenedor<>();
```

Para definir una clase de tipo genérico T, se escribe <T> después del nombre de la clase. `class nombreClase<T> {}`

Se suele usar la letra T para el tipo genérico, *pero puede ser cualquier otra*, aunque es costumbre reservar **E** para elementos de colecciones, **K** para claves, **V** para valores o **N** para números.

En la implementación de una clase puede haber **más de un tipo genérico**→ se especifican los parámetros **separados por comas**. `class nombreClase<U,V…> {}`

Las clases definidas con tipos genéricos también pueden usarse sin ellos → el compilador trabaja por defecto con variables de tipo Object.

1.2. Interfaces con genéricos

También se pueden definir interfaces con tipos genéricos y la sintaxis es idéntica.
`Interface nombreInterfaz<T> {}`

ejemplos de **interfaz con tipo genérico**: **Comparable**, de java.lang + **Comparator**.

Para **invertir el criterio de orden natural** de una clase **T** que implemente Comparable, podemos implementar una clase comparadora nosotros mismos o bien extraerla de T por el **método estático naturalOrder() de la interfaz Comparator**.

→ ejemplo, conseguir un comparador con el criterio de orden de la clase Integer

```
Comparator<Integer> ordenInteger = Comparator.naturalOrder();
Comparator<Integer> ordenIntegerInverso = ordenInteger.reversed();
```

1.3. Parámetros genéricos limitados

Implementar una clase con el tipo genérico T implica que, en sus métodos, se van a realizar operaciones con variables de ese tipo, pero a veces no tiene sentido. (Operaciones aritméticas con String). Para casos así, existen los **tipos genéricos limitados.**

La idea es **limitar los posibles tipos de T:**

1. Limitar **T** a una determinada clase **claseLimite y todas sus subclases** (si claseLimite es un límite superior)

ejemplo: clase Calculadora, de tipo genérico T, que realiza operaciones aritméticas. Limitarmos a las clases envoltorio que heredan de **Number** (Integer, Double...)

```
class Calculadora<T extends Number> {
    T a, b;
    //operaciones con a y b
}
```

El compilador comprobará si el tipo declarado está dentro de los límites del parámetro genérico que aparece en la definición de la clase.

```
Calculadora<Double> c = new Calculadora<>(); //CORRECTA (Double)
Calculadora<Object> c = new Calculadora<>(); //ERROR COMP (Object /=
Number)
```

2. Limitar **T** a una clase clase Limite y todas sus superclases (si claseLimite es un limite inferior).

```
class nombreClase<T super claseLimite> {
   // Cuerpo de la clase}
```

3. Limitar **T** a tipos que implementan una o más interfaces. En este caso no se usa la palabra implements, sino **extends**, como si fuera una herencia. Esta es una particularidad exclusiva de la sintaxis de los parámetros genéricos.

```
class MiClase <T extends MiInterfaz> { //¡no ponemos implementa!
    // Cuerpo de la clase
}
```

1.4. Métodos genéricos

Los parámetros genéricos de una clase o interfaz suelen aparecer en los métodos implementados dentro de ella.

→ ejemplo, en guardar() y extraer() de Contenedor, aparece el parámetro T.

Métodos genéricos: métodos con sus propios parámetros genéricos que estén definidos dentro de cualquier clase, aunque no esté implementada con parámetros genéricos. Este método se puede incluir en cualquier clase y no depende de los parámetros propios de ella.

El **tipo asociado a un método genérico** también **puede estar limitado**.

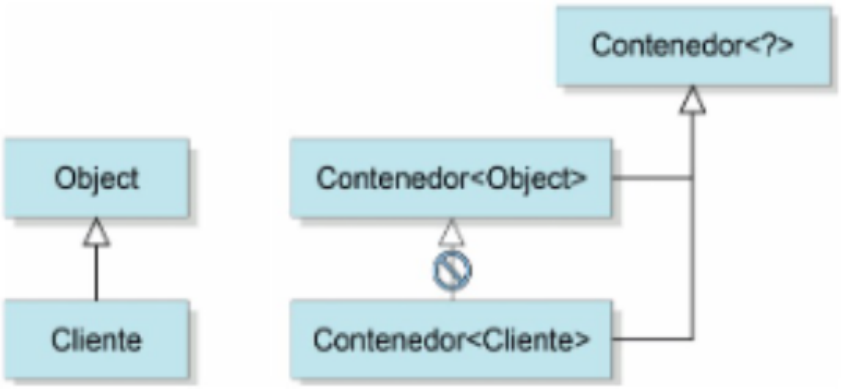
para que numeroDeNulos() solo funcionara para tablas numéricas, `<U extends Number>` en vez de `<U>`.

1.5. Comodines

Los comodines o ***wildcards*** se suelen **usar en la declaración de atributos, variables locales o parámetros pasados a una función**. Un comodín se representa con el símbolo **?** → **cualquier tipo**.

Por ejemplo `Contenedor<?> c;`
La variable c puede referenciar un Contenedor de Integer, String o de cualquier otra clase → todos los objetos Contenedor pertenecen a alguna subclase de Contenedor<?>.

Error común de uso: que Integer sea subclase de Number no implica que `Contenedor<Integer>` sea subclase de `Contenedor<Number>`. De igual modo `Contenedor<Clientes>` no es subclase de `Contenedor<Object>`.
La relación de herencia entre los valores del parámetro genérico no tiene nada que ver con la relación entre las instancias de la clase Contenedor.



Con el comodin también se puede usar:

- clase límite superior `<? extends T>` → cualquier clase que herede de T
- clase límite inferior `<? super T>` → T o cualquier superclase de T.

`Contenedor<? extends Number> c;` → declara una variable de tipo Contenedor de clase numérica. Puede referenciar un objeto Contenedor<Number>, Contenedor<Integer>, Contenedor<Double>....subclases de Contenedor<? extends Number>.

Ejemplos con **límite inferior** se presentan con frecuencia en los **comparadores**.

1.6. Cosas que no se pueden hacer con parámetros genéricos

Limitaciones:

- Los **genéricos nunca** pueden ser **tipos primitivos**. → usar **clases envoltorio**
- **No** se pueden **crear instancias de tipo genéric**o → `new T();`
- **No** se pueden **crear tablas de tipos genéricos** → `new T[10]`.
  si hacen falta, **se pasan como argumento** al método donde van a ser usadas, que puede ser un constructor, **o deberán ser devueltos** por algún método definido fuera de la clase.
- **No se pueden crear tablas de clases parametrizadas**, como
  `new Contenedor<Integer> [5];`
- No se pueden usar **excepciones** genéricas.

2. Interfaz Collection

Collection: define funcionalidades comunes de todas las colecciones (Listas y Conjuntos). Los Mapas no son Colecciones.
List y Set no implementan directamente **Collection**, si no que **son extensiones.**

2.1. Presentación de Listas [ArrayList,LinkedList]

Implementan List. → almacenan datos que se pueden repetir y en las que el orden puede ser relevante.

Diferencias: implementación interna.

ArrayList → + rendimiento en operaciones para recorrer la lista y modificarla.
LinkedList → + rendimiento en operaciones de borrado e inserción.

```
ArrayList <E> lista = new ArrayList<>;
List <Cliente> lista = new ArrayList<>;
```

2.2. Métodos básicos de la Interfaz Collection

Podemos distinguir los métodos de la interfaz Collection en dos grupos:

- **métodos básicos** – afectan a elementos individuales.
- **métodos globales** – afectan a grupos de elementos

Métodos básicos interfaz Collection:

- `boolean add(E e)`
- `boolean remove(Object o)`
- `void clear()` → borra una colección y la deja vacía
- `int size()`
- `boolean isEmpty()`
- `boolean contains(Object o)`
- `String toString()`
- `Iterator<E> iterator()`
  - `boolean hasNext()`
  - `E next()`
  - `default void remove()`

Nota: métodos como remove(), contains(), etc. utilizan **equals()** como criterio de igualdad para eliminar un elemento, búsquedas, etc.

Iteradores.

1. Reutilizar un iterador para recorrer de nuevo una colección → reinstanciarlo.
   No hay un método para reinicializar el puntero.

2. Si entre la creación de iterador y su uso hacemos cambios en la lista (inserciones o borrados) tendremos un error al usar el iterador

3. **No** podemos eliminar elementos de una colección usando un bucle foreach. Tenemos que usar un iterador.

2.3. Métodos globales de la interfaz Collection

Métodos globales:

- `boolean containsAll(Collection<?> c)`
- `boolean addAll(Collection<? extends E> c)`
- `boolean removeAll(Collection<?> c)`
- `boolean retainAll(Collection<?> c)`
- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`
  - Clase Arrays =>
    - `static <T> List<T>asList(T... a)`

2.4. Métodos de tabla de la interfaz Collection

De List a Array → Object[] toArray y <T> T[] toArray(T[] t)

- `Object[] toArray`
  `Object[] tabla1 = tablaClientes.toArray();`
  Para acceder a las posiciones de esta tabla hay que Castear

```
((Cliente)tabla1[0].nombre; //ana
```

- `<T>T[] toArray(T[] t)`: igual, pero con genérico. Se invoca por la colección de tipo T y se le pasa como parámetro una tabla del mismo tipo. No se inicializa la tabla ni importa su tamaño.

```
Cliente[] t2 = otrosClientes.toArray(new Cliente[0])
```

De Array a Lista → static <T> List <T> asList (T a)

Se hace con el método static de la clase Arrays.

- `static <T> List <T> asList (T a )`

```
Integer[] tabla = {1,2,3,4}
Collection<Integer> lista = new ArrayList<>(); //no inmutable
lista.addAll(Arrays.asList(tabla));
```

3. Interfaz List

Métodos específicos interfaz List:

- `E get(int index)`
- `E set(int index, E element)`
- `void add(int index, E element)`
- `boolean addAll(int index, Collection<? extends E> c)`
- `E remove(int index)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`
- `boolean equals(Object o)`
- `default void sort(Comparator<? super E> c)`

Las listas implementan la interfaz List, por lo que tienen métodos y funcionalidades que no comparten con los Sets.
La funcionalidad exclusiva más importante de las listas es poder acceder a sus elementos mediante índices, como en Arrays.

`add()` inserta el elemento al final.
`void add(int indice, E elem)` → inserta en la posición y el resto de elementos se **desplazan** hacia el final, dejando hueco al elemento
`E set (int indice, E elem)` → inserta el elemento en la posición que se le indique, y **machaca** el valor que hubiera ahí previamente.

```
Integer y = listaEnteros.set(3,10);
listaEnteros = [ 0, 1, 2, 3, 4, 5];  //val original 3
listaEnteros.set(3,10);
sout y + lista
y: 3 //devuelve el valor original
lista: [0, 1, 2, 10, 4, 5] //machaca el anterior
```

`boolean addAll(int indice, Collection<? extends E> c)` → inserta en el índice y el resto se desplazan, tantos lugares como sean necesatios. Los elementos de c tienen que ser del mismo tipo o subtipo que E.

`E remove (int indice)` → elimina y devuelve el valor de ese índice.

Si se usa en una lista de Integer**, ojo.**
`listaEnteros.remove(5),` no borrará el número de valor 5, si no **el índice 5.**
Borrar los 5 sería: **`listaEnteros.remove(Integer.valueOf(5))`**

---

4. Interfaz Set

Trata los datos como un conjunto matemático, sin repetidos y sin orden establecido. Todos los métodos/iteradores son iguales a Collection salvo que, a la hora de añadir un duplicado, simplemente no lo hará.

No tienen los métodos de las listas (no se puede acceder a un índice en concreto), pero sí iteradores. Además, las 3 implementaciones de Set se comportan de manera diferente, por lo que se aprovecha el polimorfismo.

Implementaciones de Set:

1. **HashSet**: no garantiza **ningún orden de inserción**, pero buen rendimiento
2. **TreeSet**: peor rendimiento, pero garantiza el **orden** según el valor de los elementos insertados por **criterio natural.**
3. **LinkedHashSet**: inserta al final, **orden** basado en **inserción**.

5. Conversiones entre colecciones

De conjunto desordenado (HashSet o LinkedHashSet) a ordenado (TreeSet).

1. se construye TreeSet con el criterio de ordenación deseado + addAll() con el conjunto a ordenar.
2. Si el criterio de ordenación es el natural, se puede construir un treeSet pasando en el constructo el conjunto a ordenar.

```
Set<Integer> conjuntoEnteros = new LinkedHashSet<>(); // [4 , 1, 5, 10,
3]
//1:
Set<Integer> conjuntoEntOrdenados = new TreeSet<>();
conjuntoEntOrdenados.addAll(conjuntoEnteros) // [1, 3, 4, 5, 10]
//2
Set<Integer> conjuntoEntOrdenados = new TreeSet<>(conjuntoEnteros); //
[1, 3, 4, 5, 10]
```

De la misma manera, si se hiciese con Clientes, se ordenarían por orden alfabético de nombre. PERO si se **instancia un TreeSet con un Comparator, Java no añade los repetidos.** *Es decir, no saldrán dos personas que se llaman María, aunque sean personas diferentes.*

A través del constructor se pueden hacer conversiones entre todos los tipos de colecciones. Si se trabaja con colecciones de tipos diferentes, se definen las variables como tipo Collection.

```
Collection<Integer> coleccion = new ArrayList<>();
```

---

6. Clase Collections

No confundir con interfaz Collection.
Reúne utilidades en forma de métodos estáticos que trabajan con genéricos. Casi todos operan sobre listas, pero algunos valen para cualquier colección

Métodos estáticos de la clase Collections:

- `static <T> void sort(List<T> list)`
- `static <T> void sort(List<T> list, Comparator<T> c)`
- `static <T> int binarySearch(List list, T key)`
- `static <T> int binarySearch(List list, T key, Comparator<T> c)`
- `static void swap(List<?> list, int i, int j)`
- `static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)`
- `static <T> void fill(List<T> list, T obj)`
- `static <T> void copy(List<T> dest, List<T> src)`
- Otros: **`shuffle()`**, **`frequency()`**, **`max()`**, **`min()`**, **`reverse()`**, **`singleton()`**

Métodos de ordenación

- `Collections.sort(lista)` → ordena una lista en criterio natural, o del compareTo(). Los conjuntos no se pueden ordenar.
- `Collections.sort(lista, new ComparaNombres());` → ordena por el criterio del Comparator

Métodos de búsqueda:

- `binarySearch( List lista, T clave )` : búsqueda binaria de un objeto por clave de búsqueda. La lista debe estar ordenada. Se pasan como parámetros la lista y el objeto a buscar.

    p. ej → buscar a Carlos, con DNI 112

```
int indice = Collections.binarySearch(lista, new Cliente ("112"));
```

Devuelve un entero negativo, a partir del que se puede sacar su índice si se insertase manteniendo el orden.

fórmula: `indiceInsercion = -indice - 1;`

```
por ejemplo:
      Cliente eva = new Cliente("555","Eva");
      int indice = Collections.binarySearch(lista,eva)
      if indice < 0 {
            lista.add(-indice - 1, eva)  //la lista sigue ordenada
}
```

- Por criterio diferente al natural, se pasa un Comparator como tercer parámetro

Métodos de manipulación de datos

- `swap (List lista, int i, int j)` → intercambia
- `replaceAll (List lista, T antiguo, T nuevo)` → cambia todas las apariciones
- `fill(List lista, T valorRelleno)` → sustituye todos los valores por uno, del mismo tipo
- `copy(List listaDestino, List listaOrigen)` → copia si son del mismo tipo

Otros:

- `shuffle();`
- `static int frequency(Collection col, Object obj );`
- `max();`
- `max() con Comparator;`
- `reverse (List lista);`
- `static Set Set<T> singleton(T elem)` → devuelve un conjunto con elem como único elemento. Es inmutable. Se suele usar para eliminar todas las ocurrencias de un elemento repetido sin usar bucles.

---

7. Interfaz Map

Mapas (diccionarios,tablas hash): son una estructura de datos muy utilizada que permiten almacenar y procesar datos de manera eficiente.

Usos habituales:

1. **Almacenamiento de pares clave-valor:** básico, almacenar y recuperar datos, asociando una clave única a un valor determinado. Permite acceder a los valores usando sus claves.
2. **Conteo de frecuencias:** contar las veces que aparece cada palabra en un texto.
3. **Implementación de caches:** permiten almacenar y recuperar rápidamente datos que se utilizan con frecuencia, mejorando el rendimiento de las aplicaciones.
4. **Representación de datos estructurados:** representación de datos complejos con relaciones entre ellos → *modelar info de personas con sus datos de contacto.*
5. **Procesamiento de datos geoespaciales:** almacenar y procesar datos geográficos, como coordenadas, polígonos, rutas, etc. → hay librerías específicas
6. **Implementación de árboles de decisión:** las implementaciones ordenadas como TreeMap, pueden utilizarse para construir estructuras de árbol de decisión, útiles en tareas de machine learning y análisis de datos.

En los mapas, los elementos se llaman **entradas** y son pares **clave/valor**, no valores individuales. Se insertan entradas con clave, que no se puede repetir, y valor, que sí.
No son colecciones, pero están relacionados con ellas y funcionan en el mismo entorno.

Implementaciones: HashMap, TreeMap, LinkedHashMap, similares a los sets.
Operaciones básicas: inserción, lectura, eliminación.

```
Map <K,V> mapa = new HashMap<>();
Map <String, Double> mapa = new HashMap<>();
mapa.put("Ana",1.65); //inserta ana
mapa.put("Ana",1.60); //ACTUALIZA la altura de Ana
```

Métodos básicos de los mapas (interfaz Map<K,V>):

- `V put(K key, V value)`
- `V remove(Object key)`
- `void clear()` → deja el mapa vacío
- `V get(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`

Vistas de claves, valores y entradas:

- `Set<K> keySet()`
- `Collection<V> values()`
- `Set<Map.Entry<K,V>> entrySet()`

* Podemos iterar sobre las vistas para eliminar entradas del mapa con el método `remove()` del iterador.

Método de la interfaz Map.Entry<K,V>:

- `K getKey()`
- `V getValue()`
- `V setValue(V value)`

Vistas Collection de los mapas.

Vista de mapa: colección respaldada por el mapa original, en la que cuando accedemos a un elemento se accede a la vista original del mapa.

- `Set<K> keySet()`
- `Collection<V> values()`

```
Set<String> claves = mapa.keySet(); //claves: Sara, Ana..
Collection<Double> estaturas = mapa.values(); //1.54, 1.60...
```

Inconvenientes del mapa: **no se puede iterar**, ni se puede usar un for each sobre el mismo mapa (sí sobre las colecciones) . Se tienen que usar los metodos del Iterator a **través de la vista** de colección **sólo para borrar**. **No se puede insertar**, para eso hay que utilizar put.

Implementación de Map

TreeMap, como TreeSet, permite inserción ordenada por orden natural creciente de las claves. Para hacer el orden diferente, podemos pasar un comparator al constructor.
LinkedHashMap mantiene el orden en que se insertan las entradas. Es eficiente en inserción y eliminación, pero más lento en búsquedas.

UD6.2. Stream

Introducción

Las colecciones aportan versatilidad y potencia al procesamiento y la manipulación de datos complejos. PERO recorrerlas nodo a nodo con iteradores es incómodo.

Desde Java 8, hay **herramientas** que permiten **efectuar operaciones globales con los elementos de una colección, sin necesidad de recorrerlas nodo a nodo**, aprovechando el procesamiento paralelo (ejecución simultánea de dos partes del código). Además, estas operaciones **pueden encadenarse**, una a continuación de otra, formando **tuberías.**

1. Interfaces funcionales y expresiones lambda

I**nterfaces funcionales:** interfaces que tienen **un solo método abstracto**.

Tienen una **sintaxis alternativa** más sencilla. (p.ej, **Comparator**)

Primera forma

Creamos una **clase ComparaNombres que implemente la interfaz Comparator**, para comparar objetos Cliente basándose en el atributo nombre.

```
public class ComparaNombres implements Comparator<Cliente> {
    public int compare(Cliente c1, Cliente c2) {
        return c1.getNombre().compareTo(c2.getNombre());
    }
}
```

A continuación, creamos un **objeto ComparaNombres** y lo pasamos al método sort():

```
Comparator<Cliente> comp = new ComparaNombres();
Collections.sort (clientes, comp);
```

Incluso podríamos prescindir de la variable comp, escribiendo una sola sentencia,

```
Collections.sort(clientes, new ComparaNombres());
```

Segunda forma

Si vamos a usar el comparador una sola vez, no merece la pena implementar la clase comparadora. Basta crear un objeto con una clase anónima.

```
Comparator<Cliente> comp2 = new Comparator<Cliente>() {
    @Override
    public int compare(Cliente c1, Cliente c2) {
        return c1.getNombre().compareTo(c2.getNombre());
    }
};
Collections.sort(clientes, comp2);
```

O bien

```
Collections.sort(clientes, new Comparator<Cliente>() {
    @Override
    public int compare(Cliente c1, Cliente c2) {
        return c1.getNombre().compareTo(c2.getNombre());
    }
});
```

En el segundo caso el constructor de Comparator usa el operador diamante, ya que Java infiere el tipo Cliente de la lista que se pasa como primer parámetro.

Tercera forma (expresiones lambda)
La sentencia anterior es la forma más corta de escribir el código para hacer la ordenación de la lista de clientes, pero en ella hay información redundante.
Podríamos preguntamos por qué es necesario especificar el nombre del método compare() cuando sabemos que la interfaz Comparator solo tiene ese método abstracto. Esa es la idea que subyace en la sintaxis de las expresiones lambda.

Para implementar una interfaz funcional con una expresión lambda, basta escribir la lista de parámetros y el cuerpo de la función abstracta separados por una **flecha** (**->**). En nuestro ejemplo, implementar el comparador de nombres de clientes consiste en implementar el método compare () que, en forma de expresión lambda, quedaría así:

```
Comparator<Cliente> comp3 = (Cliente c1, Cliente c2) -> {
    return c1.getNombre().compareTo(c2.getNombre());
};
Collections.sort(clientes, comp3);
```

Todo lo que está a la derecha del operador de asignación es la *expresión lambda del método compare() de la interfaz Comparator*, implementado para comparar nombres.

- El nombre del método no aparece, ya que Java lo infiere del lado izquierdo, donde aparece el de la interfaz Comparator, cuyo único método abstracto es compare(). Por tanto, Java sabe que en el lado derecho estamos implementando compare.
- Java también puede inferir el tipo de los parámetros de entrada (Cliente) que podríamos omitir del lado derecho.

- Si la implementación del método consta de una sola línea podemos omitir las llaves ({}) y el return.

```
Collections.sort(clientes, (c1, c2) -> c1.getNombre().compareTo(c2.getNombre()));
```

2. Algunas interfaces funcionales de la API

Algunas interfaces funcionales del paquete java.util.function:

- **Predicate<T>**. Comprueba una condición en un valor del tipo T.

  `boolean test(T t);` Devuelve true si la condición se verifica y false en caso contrario.

  Métodos por defecto:
  - `Predicate<T> negate()`
  - `Predicate<T> and(Predicate<T> o)`
  - `Predicate<T> or(Predicate<T> o)`

- **Function<T, R>**. Aplica una función a un valor de entrada y devuelve un resultado.

  `R apply(T t);` Acepta el parámetro de tipo T y devuelve un resultado de tipo R.

- **Consumer<T>**. Realiza una acción a partir del dato de entrada.

  `void accept(T t);` realiza la operación indicada.

Interface Iterable<T>
- `default void forEach(Consumer<? super T> action)`

En vista de la simplicidad y la versatilidad de las interfaces funcionales, se ha definido un cierto número de ellas que, como `Comparator`, corresponden a operaciones fundamentales, frecuentes en las tareas del programador. A continuación, vamos a ver las más importantes, que además serán necesarias con los objetos de tipo `Stream` que estudiaremos más adelante.

- **`Predicate<T>`**: se emplea para comprobar una condición en un valor del tipo genérico `T`. Su método abstracto es:

`boolean test(T valor)`: devuelve `true` si la condición se verifica para `valor` y `false` en caso contrario. Por ejemplo, para comprobar si un `Integer` es positivo, podemos definir el predicado.

```
Predicate<Integer> esPositivo = x -> x > 0;
```

Entoces,

```
esPositivo.test(5)
```

devolverá `true`.

El método `test()` es el único abstracto de la interfaz `Predicate`, pero junto a él hay otros tres métodos por defecto:

1. `Predicate<T> negate()`: devuelve un nuevo predicado que es la negación del predicado invocante. En nuestro caso,

   ```
   esPositivo.negate()
   ```

   nos devuelve un predicado que comprueba si un `Integer` no es positivo (es menor o igual que 0).

   ```
   Predicate<Integer> esNoPositivo = esPositivo.negate();
   ```

   La expresión

   ```
   esNoPositivo.test(5)
   ```

   devolverá `false`. En una sentencia única

   ```
   esPositivo.negate().test(5)
   ```

   que dará el mismo resultado, `false`.

2. `Predicate<T> and(Predicate<? super T> otro)`: devuelve un predicado que es la conjunción del predicado invocante y del que se pasa como parámetro, de modo que `test()` devolverá `true` cuando los dos predicados sean ciertos para el valor que se le pase como parámetro. El tipo genérico de `otro` debe ser igual o una superclase de `T` para garantizar que no va a contener ni evaluar más atributos que los de la clase `T`. Veámoslo con un ejemplo.

Para ello vamos a definir un segundo predicado,

```
Predicate<Integer> esPar = n -> n % 2 == 0;
```

que comprueba si un entero es par.

Si queremos saber si el entero 6 es par y positivo, escribimos

```
Predicate<Integer> esPositivoYPar = esPar.and(esPositivo);
```

Entonces, la expresión

```
esPositivoYPar.test(6)
```

devolverá `true`, ya que 6 es par y positivo a la vez.

También podemos poner

```
esPar.and(esPositivo).test(6)
```

En cambio,

```
esPar.and(esPositivo).test(-6)
```

y

```
esPar.and(esPositivo).test(7)
```

devuelven `false`, ya que –6 es par, pero no positivo y 7 es positivo, pero impar.

3. `Predicate<T> or(Predicate<? Super T> otro)`: devuelve un predicado cuyo método `test()` devolverá `true` cuando al menos uno de los dos predicados —invocante y otro— sea `true` para el valor que se le pase como parámetro.

```
Predicate<Integer> esPosotivoOPar = esPositivo.or(esPar);
esPositivoOPar.test(6) //true, par y positivo
esPositivoOPar.test(5) //true, es positivo
esPositivoOPar.test(-2) //true, es par
esPositivoOPar.test(-3) //false, no es par ni positivo
```

- **`Function<T, V>`**: coincide con la funcionalidad de las funciones matemáticas. Su único método abstracto es:

  `V apply(T x)`: acepta un parámetro de tipo `T` con el que hace una serie de operaciones que dan como resultado un valor de tipo `V`, que es devuelto por la función. Por ejemplo, si queremos definir una función que calcula el cuadrado de un valor real (de tipo `Double`),

  ```
  Function<Double, Double> cuadrado = x -> x*x;
  System.out.println(cuadrado.apply(2.0)); /*mostrará 4.0 por consola*/
  ```

  Además, `Function` añade tres funciones por defecto, que sirven para componer funciones. No las vamos a estudiar aquí por salirse del propósito de este libro.

- **`Consumer<T>`**: sirve para realizar una acción a partir de un argumento de entrada. Su método abstracto es:

  `void accept(T t)`: recibe uçn valor del tipo `T`, con el que hace operaciones sin devolver nada.

Por ejemplo, si queremos mostrar por pantalla un saludo a distintos clientes,

```
Consumer<Cliente> saludoClie = c -> System.out.println("Hola, " + c.nombre);
```

El método `accept()`, recibe como argumento un objeto `Cliente` y, a partir de él, creará un mensaje de saludo con su nombre.

```
Cliente clie=new Cliente("123", "Jorge", 20);
saludoClie.accept(clie); //se mostrará "Hola, Jorge"
```

A veces querremos que un objeto `Comsumer` actúe sobre un conjunto de instancias de una determinada clase. Para ello se usa el método `forEach()`, de la interfaz `Iterable<T>`,

```
default void forEach(Consumer<? super T> accion)
```

Este método podrá ser llamado por cualquier objeto que implemente `Iterable`, como, por ejemplo, las colecciones `ArrayList`, `LinkedList`, `HashSet`, `TreeSet` o `LinkedHashSet`. El método lo recorrerá y realizará la acción «para cada» (*for each*, en inglés) uno de sus elementos. Por ejemplo, si queremos saludar a todos clientes de `listaClientes`,

```
List<Cliente> listaClientes = new ArrayList<>();
listaClientes.add(new Cliente("111", "Marta", "12/02/2000"));
listaClientes.add(new Cliente("115", "Jorge", "16/03/1999"));
listaClientes.add(new Cliente("112", "Carlos", "01/10/2002"));
listaClientes.add(new Cliente("211", "Ana", "07/12/2001"));
listaClientes.forEach(saludoClie);
```

La API proporciona otras interfaces funcionales importantes que iremos viendo.

Una particularidad de las clases anónimas y de las expresiones lambda (en realidad, de todas las clases llamadas *locales*, cuyo estudio excede el objeto de este libro) es que dentro de ellas se pueden usar variables locales del ámbito donde está definida la expresión, es decir, dentro del mismo bloque de sentencias. Por ejemplo, en el siguiente código, se puede usar la variable `x` en la expresión lambda, pero no la `y`:

```
int y = 5;
{
        int x = 6;
        Function<Integer, Integer> f = a -> a + x;//Correcto
        Function<Integer, Integer> g = a -> a + y;// ¡Error!
}
```

Sin embargo, la variable local que se incluya en una expresión lambda (en nuestro caso, la `x`) debe ser una constante, bien declarada con el modificador `final`, o bien «efectivamente inmutable», que significa que, aunque no se haya declarado `final`, actúa como si lo fuera. Es decir que, una vez declarada e inicializada, no cambia su valor dentro de su ámbito de existencia, ya sea antes de la expresión lambda, dentro de ella o después de ella.

```
int x = 6;
x++; // ¡Error!
Function<Integer, Integer> f = a -> a + x++; // ¡Error!
x=10; // ¡Error!
```

2.1. Referencias a métodos: Clase::metodo

Pero cuando una función ya está implementada en un método de alguna clase, como ocurre con Math.sqrt() podemos escribirla como una **referencia al método.**
`Math::sqrt` → se puede colocar en lugar de la expresión lambda, `x -> Math.sqrt(x)`

Entonces, para calcular raíces cuadradas de valores Double, podemos implementar `Function<Double, Double> raiz = Math::sqrt;`

Las referencias a métodos se escriben poniendo el **nombre de la clase**, seguido de dobles dos puntos (**::**) y del **nombre del método** (sin paréntesis ni lista de argumentos) cuando éste es estático.
Si el método no es estático, en vez del nombre de la clase pondremos una referencia a un objeto de la clase donde está definido el método. En nuestro caso, hemos escrito una referencia al método estático sqrt(), definido en la clase Math de la API.

El método referenciado debe tener parámetros de entrada y tipo devuelto compatibles con el método que debe implementar.

A la hora de asignar una referencia a un método a una variable de tipo Function (o de cualquier otra interfaz funcional), el método puede estar implementado en una clase cualquiera. Por ejemplo, en una clase definida por nosotros como los siguientes.

```
class Calculos {
    Integer cuadrado(Integer a) {
        return a * a;
    }
    static Integer cubo(Integer x) {
        return x * x * x;
    }
}
```

Cualquiera de ellos puede ser asignado a una variable de tipo Function, ya que su estructura de parámetros de entrada y tipo devuelto es compatible con el método apply() definido en la interfaz. Se accede al **método estático** por medio del nombre de la clase y al **método no estático** a través de un objeto creado previamente.

```
Function<Integer, Integer> f1 = Calculos::cubo;
Calculos calc = new Calculos();
Function<Integer, Integer> f2 = calc::cuadrado;
```

No obstante, **si se trata de un método no estático de la propia clase a la que pertenece el valor al que se aplica**, se puede invocar con el nombre de la clase, sin necesidad de crear un nuevo objeto. Por ejemplo, si implementamos la clase Entero.

```
class Entero {
    Integer valor;
    public Entero(Integer valor) {
        this.valor = valor;
    }
    Entero siguiente() {
        return new Entero(valor + 1);
    }
    @Override
    public String toString() {
        return "Entero{" + "valor=" + valor + '}';
    }
}
```

Ahora podemos definir la función siguienteEntero, que nos devuelve un objeto con valor incrementado en 1.

```
Function<Entero, Entero> siguienteEntero = Entero::siguiente;
System.out.println(siguienteEntero.apply(new Entero (3))); //4
```

La referencia al método no estático siguiente se hace a través del nombre de la clase Entero. Esta situación la encontraremos frecuentemente al trabajar con la interfaz Stream.

Un ejemplo un poco más elaborado: un **método estático que aplica una transformación a todos los elementos de una tabla**, que también se le pasa como parámetro.

```
static <T> void aplicar(T[] tabla, Function<T, T> m) {
    for (int i = 0; i < tabla.length; i++) {
        tabla[i] = m.apply(tabla[i]);
    }
}
```

Podemos probar el método con una tabla de enteros que deberá elevar al cuadrado con el método cuadrado() de la clase Calculos.

```
Integer[] t = {1, 2, 3, 4, 5};
aplicar(t, f2); //o aplicar(t, calc::cuadrado):
System.out.println(Arrays.toString(t)); // [1 4 9 16 25]
```

Los nombres de los métodos son cuadrado() o cubo (), no apply().
Java infiere del tipo del parámetro de entrada m (la interfaz Function), que ambos métodos deben identificarse con apply(). Para que esto sea posible, los parámetros de entrada y el tipo devuelto de los métodos referenciados cuadrado () o cubo () tienen que ser compatibles con la definición de apply().

Todo esto es extensible a cualquier interfaz funcional, ya sea de la API o creada por nosotros.

También se pueden usar **referencias a constructores**. La sintaxis incluye el nombre de la clase y la palabra new: **Cliente::new**.

En el siguiente ejemplo, implementamos la interfaz Function para construir objetos de la clase Saludo.

```
class Saludo {
    String nombre;
    Saludo(String nombre) {
        this.nombre = nombre;
    }
    public String toString() {
        return "Hola, nombre";
    }
}
```

El método apply() de la interfaz Function recibirá una cadena con el nombre, y deberá construir y devolver un objeto Saludo con ese nombre.

```
Function<String, Saludo> construyeSaludo = Saludo::new;
Saludo s = construyeSaludo.apply("Claudia");
System.out.println(s); //¡Hola Claudia!
```

A la hora de ejecutar apply(), Java busca el constructor en la clase Saludo y lo ejecuta pasando el valor “Claudia” como parámetro.

Sobre hashCode y colecciones en Java

hasCode() y colecciones

Un HashSet es una colección que no permite duplicados y utiliza el código hash de los elementos para determinar su ubicación en la colección.
Cuando se agrega un elemento a un HashSet, se llama al método hashCode() del elemento para calcular su código hash. Luego, el HashSet utiliza este código hash para determinar la ubicación del elemento en la colección.

Las colecciones basadas en hash son muy eficientes para ciertas operaciones. (**búsqueda**, **recuperación de un valor asociado a una clave**), ya que utilizan el código hash para determinar su ubicación.
El **rendimiento depende de la calidad de la implementación del método hashCode()** de los elementos. Si no está implementado correctamente y produce muchos códigos hash iguales para elementos diferentes, el rendimiento disminuye.

Implementar hashCode()

La manera correcta de implementar el método hashCode() depende de la clase en la que se está implementando. En general, debe cumplir con ciertas reglas para garantizar un comportamiento correcto en colecciones basadas en hash.

- Forma 1 → función hash (clase Objects) para calcular el código a partir de los campos relevantes del objeto. El método hash **toma como argumentos los valores relevantes del objeto** y **utiliza** el **algoritmo de hash estándar para calcular un código hash combinado**.
  Este código hash se puede utilizar como valor de retorno del método hashCode() del objeto.

- Forma 2 → multiplicar por un número primo (31 en las JDK) y acumular la suma

hashCode() y equals()

equals() **no** llama a hashCode() para determinar la igualdad de dos objetos. Sin embargo, es importante que sean consistentes entre sí. Si se sobreescribe equals(), también se tiene que hacer el otro para conservar la devolución de objetos iguales.

hashCode() y compareTo()

hashCode() **no tiene una relación directa** con compareTo().

- hashCode: se utiliza para obtener un código hash que sirve como un identificador del objeto en colecciones basadas en Hash.
- compareTo: se utiliza para comparar objetos y determinar su orden en colecciones ordenadas como TreeSet y TreeMap.