# Tema 4 –Colecciones y Genericidad

Programación Orientada a Objetos

Grado en Ingeniería Informática

DIS
Departamento de Informática y Sistemas

UNIVERSIDAD DE MURCIA

# Introducción

- Este tema introduce la biblioteca de **colecciones** de Java y el concepto de **genericidad**.
- La genericidad es el mecanismo que permite programar una clase como `LinkedList` de modo que pueda ser utilizada con cualquier tipo de datos.
- El tipo al que se *parametriza* la colección se especifica entre <>:

```
LinkedList<Punto> puntos = new LinkedList<Punto>();
```

- En caso de utilizar **tipos primitivos**, es necesario hacer uso de su tipo envoltorio: `Integer` para `int`, etc.
- En el tema se presentan primero las colecciones y después el mecanismo de genericidad.

# Contenido

- **Parte 1 – Colecciones**
  - Tipos `Collection`, `SequencedCollection`, `List` y `Set`
  - Tipos `Map`, `HashMap`
  - Ordenación: `Comparable` y `Comparator`
    - `TreeSet` y `TreeMap`
  - Iteradores. Tipos `Iterable` e `Iterator`

- **Parte 2 – Genericidad**
  - Definición de clases genéricas
  - Declaración y construcción de tipos genéricos
  - Genericidad y tipos dinámicos
  - Métodos genéricos

# Contenido

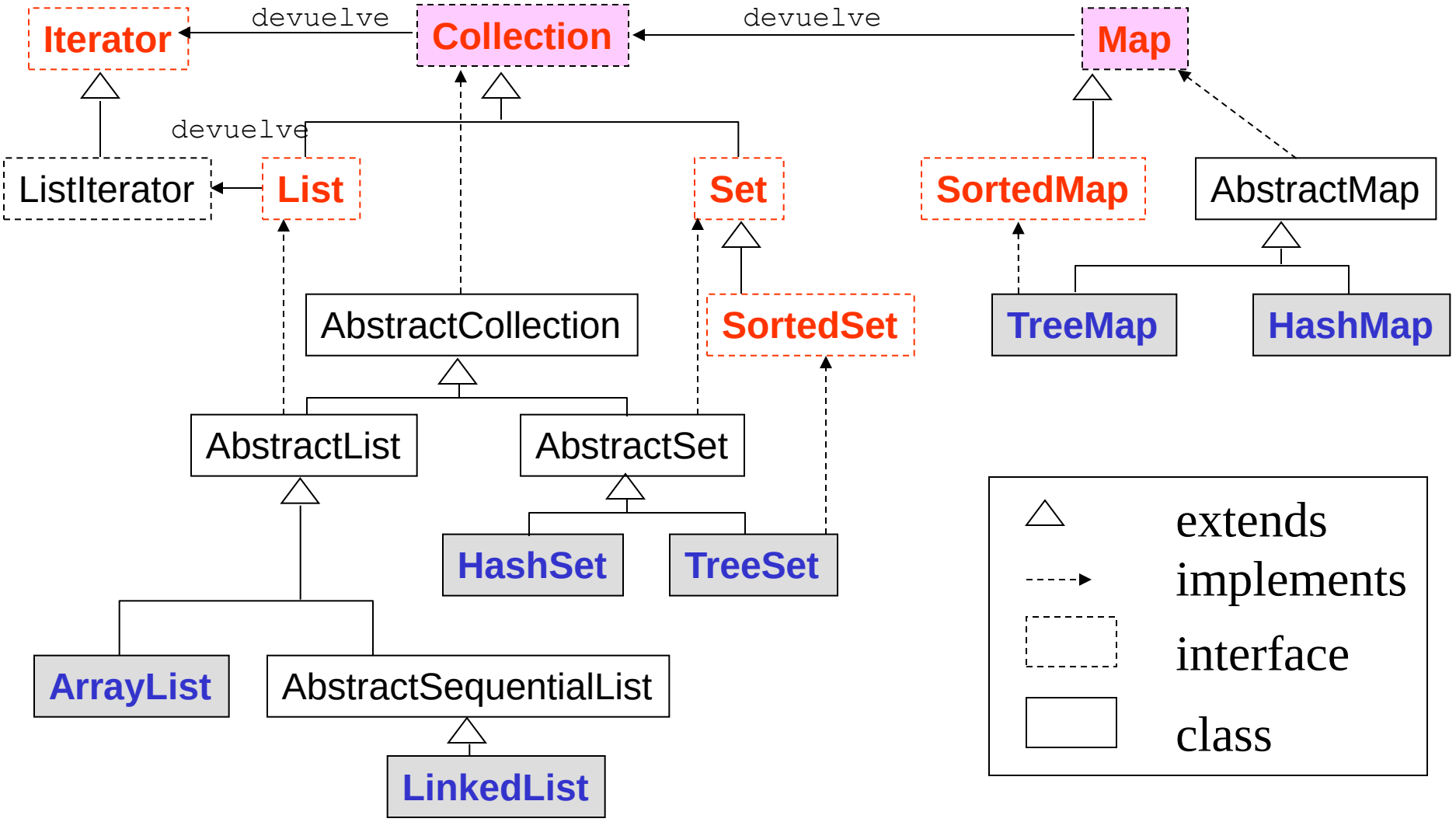
- **Parte 1 – Colecciones**
  - Tipos `Collection`, `SequencedCollection`, `List` y `Set`
  - Tipos `Map`, `HashMap`
  - Ordenación: `Comparable` y `Comparator`
    - `TreeSet` y `TreeMap`
  - Iteradores. Tipos `Iterable` e `Iterator`

- **Parte 2 – Genericidad**
  - Definición de clases genéricas
  - Declaración y construcción de tipos genéricos
  - Genericidad y tipos dinámicos
  - Métodos genéricos

# Parte 1 - Colecciones en Java

- Las colecciones en Java son un **ejemplo** destacado de implementación de **código reutilizable** utilizando un lenguaje orientado a objetos.
- Todas las colecciones son **genéricas**. Están disponibles en el paquete `java.util`.
- Los tipos abstractos de datos se definen como **interfaces**.
- Se implementan **clases abstractas** que permiten factorizar el comportamiento común a varias implementaciones.
- Un mismo **tipo de datos** puede ser implementado por varias clases → **`List`**: **`LinkedList`**, **`ArrayList`**

# Colecciones en Java (v1)



# Interfaz `Collection<T>`

- Define las operaciones comunes a todas las colecciones de Java.
- Permite usar colecciones basándonos en su interfaz en lugar de en la implementación.
- Los tipos básicos de colecciones son (subtipos de `Collection<T>`) :
  - Listas, definidas en la interfaz `List<T>`
  - Conjuntos, definidos en la interfaz `Set<T>`

# Interfaz `Collection<T>`

- **Operaciones básicas de consulta**:
  - `size()`: devuelve el número de elementos.
  - `isEmpty()`: indica si tiene elementos.
  - `contains(Object e)`: indica si contiene el elemento pasado como parámetro.

# Interfaz `Collection<T>`

- **Operaciones básicas de consulta:**
  - Observa que el método **`contains`** recibe como parámetro un objeto compatible con `Object`, es decir, de cualquier tipo de datos.
  - Es un error frecuente consultar en una colección si existe un elemento que no es del tipo de la colección.

```
LinkedList<String> lista = new LinkedList<String>();
// … añadir cadenas

lista.contains(10); // ¿compila?
```

# Interfaz `Collection<T>`

- **Operaciones básicas de consulta:**
  - Los tipos de colecciones (listas, conjuntos) determinan de forma distinta cuándo un elemento está en la colección.
  - Por ejemplo, las listas buscan los elementos utilizando el método `equals`.
  - En cambio, los conjuntos implementados con tablas de dispersión utilizan el código de dispersión (`hashcode`) y `equals`.

# Interfaz `Collection<T>`

- **Operaciones básicas de modificación**:
  - **`add`**`(T e)`**:** añade un elemento a la colección.
    - Retorna un **booleano** indicando si acepta la inserción.
    - Las listas siempre aceptan los elementos, por tanto, siempre retornan verdadero.
    - Sin embargo, los conjuntos retornan un valor falso si el elemento está repetido.
  - **`remove`**`(Object e)`**:** intenta eliminar el elemento.
    - Retorna un booleano indicando si ha sido eliminado.

**Nota**: al igual que sucede con `contains`, para ambas operaciones, listas y conjuntos determinan si un elemento está en la colección de forma diferente.

# Interfaz `Collection<T>`

- **Operaciones básicas de modificación**:
  - **`clear()`**: elimina todos los elementos.
  - **`addAll(otra)`**: añade todos los elementos de la colección `otra`.
  - **`removeAll(otra)`**: elimina los elementos de la colección (objeto receptor) que estén contenidos en la colección establecida como parámetro (`otra`).

# Interfaz `List<T>`

- La interfaz **`List<T>`** define **secuencias** de elementos a los que se puede acceder atendiendo a su posición.
- En la librería se ofrecen varias implementaciones: `LinkedList<T>` y `ArrayList<T>`.

- Las posiciones válidas van de `0` a `size()-1`.
  - En caso de acceso fuera de rango, se produce un error de ejecución.
- El método **`add`**`(T e)`:
  - Añade al elemento al final de la lista.
  - Siempre acepta la inserción y retorna verdadero.

# Interfaz `List<T>`

- Añade a las operaciones de `Collection` métodos de acceso por posición como:
  - `T` **`get`** `(int indice)`
  - `T` **`set`** `(int indice, T nuevo)`
    - Reemplaza el elemento situado en la posición `indice` por el elemento `nuevo`.
    - Retorna el elemento que ha sido sustituido.
  - `void` **`add`** `(int indice, T nuevo)`
    - Sitúa en la posición `indice` el elemento `nuevo`.
    - Los elementos que estuvieran situados en `indice` y posteriores son desplazados a la derecha.
  - `T` **`remove`** `(int indice)`
    - Elimina el elemento en la posición `indice` y lo retorna.
    - Los elementos situados en las posiciones `indice + 1` y siguientes son desplazados a la izquierda.

# Lista `LinkedList<T>`

- La clase `LinkedList<T>` ofrece una implementación basada en **listas de nodos doblemente enlazados**

- Añade a la interfaz `List` operaciones para gestionar los extremos de la lista:
  - `addFirst`, `addLast`, `removeFirst`, `removeLast`, `getFirst` y `getLast`
  - Con ellas podemos gestionar **pilas** y **colas**.

- Interesa utilizar `LinkedList` cuando la lista irá creciendo dinámicamente por los extremos.

# Lista `LinkedList<T>`

- **Ventajas**:
  - Las operaciones en los extremos de la lista (principio y final) son eficientes (orden constante).

- **Desventajas**:
  - Las consultas por posición son lentas.
  - Para obtener el elemento en la posición `i` es necesario recorrer los `i-1` nodos precedentes.
  - Inserciones y modificaciones en posiciones intermedias tampoco son eficientes.

# Lista `ArrayList<T>`

- La clase `ArrayList<T>` implementa la interfaz `List<T>` utilizando **arrays redimiensionables:**
  - Inicialmente tiene una *capacidad* (tamaño del array).
  - La capacidad por omisión es 10.
  - Cuando se agota la capacidad, construye un nuevo array de mayor tamaño y se copian los elementos del antiguo.

```
Construye un lista con capacidad inicial 15
ArrayList<String> lista = new ArrayList<String>(15);

// Añadimos 3 elementos, tamaño (size) es 3
Collections.addAll(lista, "hola", "hello", "hallo");
```

- **Nota**: `java.utils.Collections` ofrece rutinas de utilidad sobre las colecciones. El método `addAll` permite añadir una secuencia de elementos (argumento de tamaño variable).

# Lista `ArrayList<T>`

- **Ventajas**:
  - Las consultas por posición son rápidas (orden constante).
  - Las **inserciones por el final** (`add`) son eficientes si no se desborda la capacidad.

- **Desventajas**:
  - Las operaciones de inserción y modificación en posiciones intermedias son ineficientes:
    - Por ejemplo, eliminar el primer elemento supone tener que desplazar el resto de elementos del array.
  - Las operaciones de inserción pueden desbordar la capacidad y tener que reconstruir el array.

# Lista `ArrayList<T>`

- Interesa utilizar `ArrayList` cuando conocemos a priori el tamaño que tendrá la lista y la colección solo será consultada.
- Ejemplo:

```
class Procesador {
   private ArrayList<String> lista;

   public Procesador(String... palabras) {
      // Conocemos la capacidad de la lista
      this.lista = new ArrayList<String>(palabras.length);

      for (String palabra : palabras) {
         // Inserción eficiente, no supera capacidad
          this.lista.add(palabra);
      }
   }
}
```

# Búsqueda y borrado en listas

- En las listas, la operación de **búsqueda** (`contains`) y **borrado** (`remove`) de objetos utiliza el método **`equals`** para localizarlos.
- En general ambas operaciones no son eficientes:
  - Se recorre la colección desde el principio hasta localizar el objeto.
  - Las implementaciones `ArrayList<T>` y `LinkedList<T>` tienen similar rendimiento para localizar el objeto.
  - Sin embargo, en la operación de borrado, `LinkedList<T>` es más eficiente porque sólo requiere eliminar un nodo.
  - En cambio, `ArrayList<T>` puede necesitar desplazar objetos en el array.

# Interfaces `Set<T>` y `SortedSet<T>`

- La interfaz **`Set<T>`** define **conjuntos**, esto es, colecciones con elementos no repetidos.
  - Esta interfaz extiende la interfaz `Collection<T>`. Sin embargo, no añade nuevas operaciones.
  - Precisa la semántica del método **`add()`** para indicar que no se admiten elementos repetidos.

- La interfaz `Set<T>` es especializada por la interfaz **`SortedSet<T>`**, que define **conjuntos ordenados**.

# Interfaces `Set<T>` y `SortedSet<T>`

- La librería de Java ofrece dos implementaciones de conjuntos:
  - **`HashSet<T>:`** conjunto implementado con tablas de dispersión.
  - **`TreeSet<T>:`** **conjunto ordenado** implementado con árboles binarios de búsqueda balanceados
    - Para su funcionamiento es necesario definir un **orden** (se estudia más adelante).

# Conjunto `HashSet<T>`

- La clase **`HashSet<T>`** implementa un conjunto utilizando una **tabla de dispersión**.

- Para su correcto funcionamiento exige que la clase de los objetos que se almacenan en la colección ofrezca una **implementación consistente de los métodos `equals()` y `hashCode()`**:

  - Si `o1.equals(o2) == true` entonces `o1.hashCode() == o2.hashCode()`

# Conjunto `HashSet<T>`

- La clase **`HashSet<T>`** determina si un **objeto está repetido** en el conjunto utilizando los métodos `equals/hashCode`.
- Al insertar un nuevo objeto solicita su código de dispersión (método `hashCode`):
  - 1. Si el código de dispersión no está en la tabla, entonces se inserta el objeto (no está repetido).
  - 2. Si el código está en la tabla (colisión), consulta si el objeto con el mismo código de dispersión que ya está en la tabla es igual (`equals`) que el objeto que se quiere insertar.
  - 3. En caso de ser iguales, se descarta por ser repetido.
- En definitiva, la operación **`add()`** es **eficiente**.

# Conjunto `HashSet<T>`

- A diferencia de las listas, **los elementos de un conjunto no se pueden recuperar por posición**.
- Para recuperar los elementos de un conjunto es necesario utilizar un *iterador* (se estudia más adelante) o su versión alternativa mediante un **recorrido *for each***:

```
HashSet<String> conjunto = new HashSet<String>();
Collections.addAll(conjunto, "Juan", "Luis", "Pedro", "Juan");
System.out.println(conjunto.size()); // 3, ha rechazado uno

for (String nombre : conjunto) {
    System.out.println(nombre); // Luis, Pedro y Juan
}
```

# Conjunto `HashSet<T>`

- El orden en el que se recuperan los objetos de un conjunto durante el **recorrido** no necesariamente coincide con el orden en el que fueron insertados.
  - Se obtienen en el orden en el que aparecen en la tabla de dispersión.
  - En el ejemplo anterior el orden de recorrido no coincide con el orden en el que se han insertado.

- Los **conjuntos ordenados** (`TreeSet<T>`) mantienen los objetos ordenados (se estudia más adelante).

# Conjunto `HashSet<T>`

- Las operaciones de **consulta** (`contains`) y de **borrado** (`remove`) son **eficientes**.

- Se utiliza el código de dispersión para localizar el objeto en la tabla:
  - Un elemento está en el conjunto (`contains`) si el código de dispersión está registrado en la tabla y el elemento asociado al código es `equals` al elemento que se busca.
  - Borrar un elemento implica buscarlo (similar a `contains`) y si se localiza eliminarlo de la tabla.

# Colecciones secuenciadas

- Los problemas de las colecciones anteriores son:
  - No hay una forma homogénea de acceder al principio y al final de la colección.

| Tipo | Primer elemento | Último elemento |
|---|---|---|
| `List` | `lista.get(0);` | `lista.get(lista.size()-1);` |
| `LinkedList` | `lista.getFirst(0);` | `lista.getLast();` |
| `SortedSet` | `set.first();` | `set.last();` |

    - En un mapa habría que recorrer todas las entradas para llegar a la última.
  - No es fácil recorrer la colección en sentido inverso (del final hacia delante).
  - **Solución**: colecciones secuenciadas.

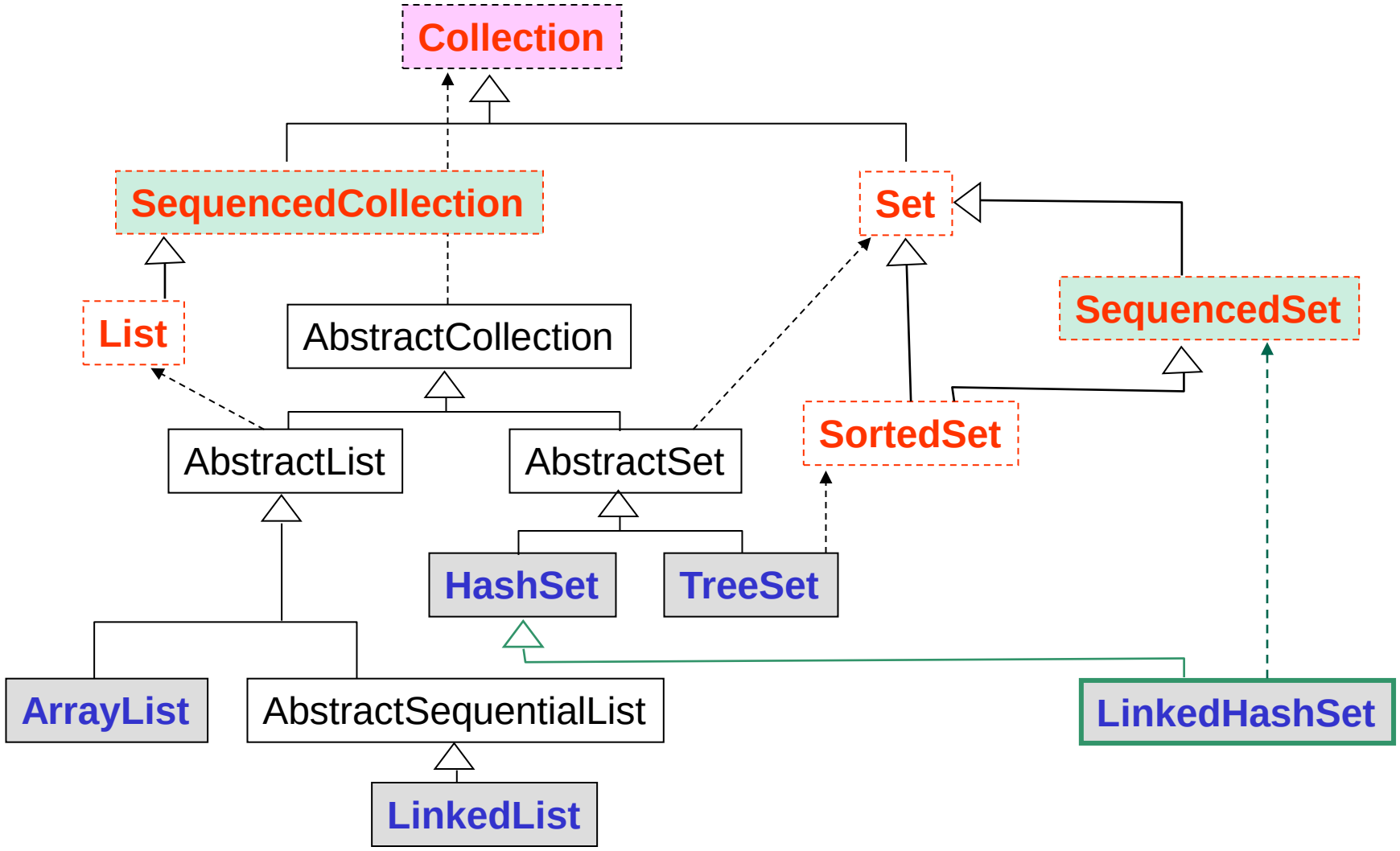
# Colecciones secuenciadas

- En Java 21 se introducen 3 nuevas interfaces:
  - `SequencedCollection`
  - `SequencedSet`
  - `SequencedMap`

- Representan colecciones secuenciales que soporta operaciones en los extremos y son reversibles.

- Las nuevas interfaces mantienen la compatibilidad con las versiones anteriores gracias a la implementación de la nueva funcionalidad utilizando métodos por defecto.

# Colecciones en Java JDK 21 (v2)



# Interfaz `SequencedCollection<E>`

```
interface SequencedCollection<E>
                    extends Collection<E> {

    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
    SequencedCollection<E> reversed();

}
```

# Interfaz `SequencedSet<E>`

```
interface SequencedSet<E>
          extends Set<E> {
          extends SequencedCollection<E>
{

     // redefine para aplicar
     // la regla covariante
     SequencedSet<E> reversed();

}
```

# Ordenación de una lista

- La librería de Java ofrece la clase `java.util.Collections` con rutinas de utilidad para gestionar colecciones.
- El método **`sort`** ordena una lista.
- El siguiente ejemplo muestra la ordenación de una lista de cadenas:

```
LinkedList<String> saludos = new LinkedList<>();
Collections.addAll(saludos, "hola", "hello", "hallo");

Collections.sort(saludos);

for (String saludo : saludos) {
    System.out.println(saludo); // hallo, hello, hola
}
```

# Ordenación de una lista

- El método `sort` es capaz de ordenar una lista si el tipo de datos de los objetos que contiene es *comparable*.
- Una clase es comparable si implementa la interfaz **`Comparable<T>`**:

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

  - El método `compareTo` compara el objeto receptor y el parámetro. Devuelve un entero positivo si el objeto receptor es mayor, negativo si es menor y cero si es igual al parámetro.
- En el ejemplo anterior la clase `String` es *comparable*: implementa el orden alfabético de las cadenas.

# Ordenación de una lista

- **Ejemplo**: clase que representa un pedido con dos propiedades (producto y cantidad).
- El criterio de ordenación está basado en el orden de la propiedad *producto*. Así pues, un pedido es menor que otro si el nombre del producto es menor en orden alfabético.

```
public class Pedido implements Comparable<Pedido> {
    private final String producto;
    private final int cantidad;

    // Se omite constructor y métodos get

    @Override
    public int compareTo(Pedido otro) {
        return this.producto.compareTo(otro.producto);
    }
}
```

# Ordenación de una lista

- Las clases que implementan la interfaz `Comparable<T>` se dice que tienen **orden natural**.
- No todas las clases ofrecen orden natural. Incluso si lo ofrecen, podría interesar ordenar los objetos de acuerdo a otro criterio.
- La clase `Collections` ofrece una versión sobrecargada del método `sort` que recibe como argumento un criterio de ordenación (interfaz **`java.util.Comparator<T>`**).
- A partir de la versión 8 de Java está disponible el *método por defecto* `sort` en la interfaz `List` que también recibe como parámetro un criterio de ordenación (`Comparator`).

# Criterios de ordenación

- Interfaz **`Comparator<T>:`**

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

- El método `compare` devuelve un entero positivo si `o1` es mayor que `o2`, negativo si es menor y cero si son iguales.

# Criterios de ordenación

- **Ejemplo**: implementación de un criterio de ordenación para la clase `Pedido`.

- La ordenación que realiza es la siguiente:
  - Primero ordena por orden alfabético (orden natural) de la propiedad *producto*.
  - En caso de empate, ordena por orden ascendente de la cantidad (orden natural de la clase `Integer`).

- Observa que para utilizar el orden natural de un tipo primitivo se utiliza el tipo envoltorio, por ejemplo, `Integer` para `int`.

# Criterios de ordenación

```
public class ComparadorPedidos implements Comparator<Pedido> {

    @Override
    public int compare(Pedido arg0, Pedido arg1) {

        int criterio1 =
          arg0.getProducto().compareTo(arg1.getProducto());
        int criterio2 =
        ((Integer)arg0.getCantidad()).compareTo(arg1.getCantidad());

        if (criterio1 == 0) { // empate criterio 1
            return criterio2;
        }
        else return criterio1;
    }
}
```

# Criterios de ordenación

- Utilizamos el comparador implementado para ordenar una lista de pedidos:

```
LinkedList<Pedido> pedidos = new LinkedList<Pedido>();
// Se añaden pedidos ...

// Opción 1
Collections.sort(pedidos, new ComparadorPedidos());

// Opción 2
pedidos.sort(new ComparadorPedidos());
```

- En este ejemplo no importa que la clase `Pedido` tenga orden natural (`Comparable`).

- El método `sort` ordena la colección utilizando el criterio que se establece como parámetro.

# Orden natural

- Utilizamos el ***orden natural*** para ordenar una lista de pedidos:

```
LinkedList<Pedido> pedidos = new LinkedList<Pedido>();
// Se añaden pedidos ...

// Opción 1
Collections.sort(pedidos);

// Opción 2
pedidos.sort(null);
```

- Los objetos contenidos en la lista se tienen que poder comparar.

- La clase `Pedido` tiene que implementar la interfaz `Comparable`.

# Conjuntos Ordenados `TreeSet<T>`

- La clase **`TreeSet<T>`** implementa la interfaz `SortedSet<T>` que define **conjuntos ordenados**.

- Un conjunto ordenado **evita elementos repetidos** y además permite recorrer los **elementos en orden**.

```
TreeSet<String> ordenado = new TreeSet<String>();
Collections.addAll(ordenado, "hello", "hola", "hallo");

for (String saludo : ordenado) {
    System.out.println(saludo); // hallo, hello, hola
}
```

# Conjuntos Ordenados `TreeSet<T>`

- La clase **`TreeSet<T>`** requiere que las clases de los objetos de la colección implementen un orden natural (`Comparable`):
  - En el ejemplo anterior, la clase `String` es *comparable*.

- Si la clase no tiene orden natural o queremos que se ordene de acuerdo a otro criterio, en el constructor se estable un objeto `Comparator<T>`.

```
TreeSet<String> ordenado =
          new TreeSet<String>(new OrdenInverso());
Collections.addAll(ordenado, "hello", "hola", "hallo");

for (String saludo : ordenado) {
    System.out.println(saludo); // hola, hello, halo
}
```

# Conjuntos Ordenados `TreeSet<T>`

- La interfaz **`SortedSet<T>`** que implementa `TreeSet<T>` añade operaciones a la interfaz `Set<T>`.
- Las más destacables son:
  - `first()` y `last()`: retorna el menor o mayor elemento del conjunto, respectivamente.
  - `headSet(T elem)` y `tailSet(T elem)`: retornan un **conjunto ordenado** con los elementos estrictamente menores (`headSet`) o mayores (`tailSet`) que el parámetro.
- No obstante, la característica más destacada de un conjunto ordenado es que durante un recorrido los elementos se obtienen en el orden establecido.

# Conjuntos Ordenados `TreeSet<T>`

- La clase `TreeSet<T>` utiliza el criterio de ordenación (`Comparable` o `Comparator`) para:
  - Para insertar un elemento evitando repetidos (`add`).
  - Localizar un elemento en la colección (`contains`).
  - Borrar un elemento de la colección (`remove`).

- Por ejemplo, al insertar un elemento se considera que está repetido si al compararlo con algún elemento de la colección el criterio de ordenación da como resultado 0.

- Por tanto, **no se utiliza `equals` en ninguna de esas operaciones**.

# Colecciones ordenadas

- Para trabajar con colecciones ordenadas tenemos dos opciones:
  - Lista que ordenamos con `sort` cuando sea necesario.
  - Conjunto ordenado.

- La elección de una u otra depende de la gestión de los elementos repetidos:
  - Si queremos mantener **elementos repetidos**, la opción es utilizar una **lista**.
  - Si la semántica **conjunto** es importante, debemos optar por un conjunto ordenado.

# Recorridos

- El **problema** de los recorridos:
  - Solo las listas permiten recorrido por posición.
  - El rendimiento de un recorrido por posición varía según la implementación (`ArrayList` vs `LinkedList`)
  - No es posible recuperar individualmente los elementos de un conjunto.
  - Los conjuntos ordenados solo ofrecen operaciones para recuperar los elementos por los extremos (primero y último) y para extraer subconjuntos.
- Es necesario utilizar un **esquema de recorrido que sea eficiente y común a todas las colecciones**.
- La solución al problema son los **iteradores**.

# Iteradores

- Un **iterador** es un objeto que permite recorrer los elementos de una colección.

- El **esquema de recorrido** que ofrece un iterador es el siguiente:

  - `Mientras` *`quedan elementos:`*
    - *`Recupera elemento`*
    - `Procesa el elemento …`

- Este esquema es definido por la interfaz **`Iterator<T>`**

# Iteradores

- Interfaz **`Iterator<T>`**:
  - **`hasNext`**`()` : indica si quedan elementos en la iteración.
  - **`next`**`()` : devuelve el siguiente elemento de la iteración.
  - **`remove`**`()` : elimina el último elemento devuelto por el iterador.

```
public interface Iterator<T> {

    boolean hasNext();

    T next();

    void remove();
}
```

# Iteradores

- Las colecciones de Java ofrecen un iterador para su recorrido utilizando el método **`iterator()`**.
- **Ejemplo**:

```
public static int contarBurbujasExplotadas(List<Burbuja> burbujas) {

        Iterator<Burbuja> iterador = burbujas.iterator();
        int contador = 0;
        while (iterador.hasNext()) {

            Burbuja burbuja = iterador.next();
            if (burbuja.isExplotada())
                contador++;
        }
        return contador;
}
```

# Iteradores

- Las colecciones implementan la interfaz **`Iterable<T>`** lo que les obliga a ofrecer un iterador:

```
public interface Iterable<T> {

    Iterator<T> iterator();
}
```

- Cualquier otro tipo de datos que quiera ser *iterable* debe implementar esta interfaz.

- Los **arrays** también son iterables.

# Recorrido *for each*

- El recorrido **for each** permite recorrer objetos *iterables* sin manejar un objeto iterador.
- Es la opción más común de recorrido.

```
public static int contarBurbujasExplotadas(List<Burbuja> burbujas) {

        int contador = 0;
        for (Burbuja burbuja : burbujas) {
            if (burbuja.isExplotada())
                contador++;
        }
        return contador;
}
```

- El código anterior es equivalente a utilizar un iterador.

# Eliminar elementos en un recorrido

- Durante el **recorrido** de una colección con iterador explícito o con *for each* (implícito) **no está permitido modificar la colección** (añadir o quitar elementos) .
- A través de un recorrido con **iterador explítico** sí podemos eliminar el último elemento recuperado:

```
public static void eliminarExplotadas(List<Burbuja> burbujas) {

        Iterator<Burbuja> iterador = burbujas.iterator();
        while (iterador.hasNext()) {
            Burbuja burbuja = iterador.next();
            if (burbuja.isExplotada())
               iterador.remove();
        }
}
```

# Mapas

- La interfaz **`Map<K,V>`** representa una estructura de datos (mapa) que asocia pares <*clave, valor*>.

- En general un mapa es gestionado como un **conjunto** en el que los elementos son las **claves** y estas claves tienen asociado un valor.

- Por tanto, la implementación de un mapa y un conjunto es similar. Encontramos dos implementaciones:
  - **`HashMap<K,V>`**: implementación basada en una tabla de dispersión.
  - **`TreeMap<K,V>`**: implementación basada en árboles binarios de búsqueda balanceados. Representa **mapas ordenados**.

# Mapas

- Un mapa no es una colección de elementos, sino una *tabla* que asocia <clave, valor>.
- Un mapa no es *iterable*. Sin embargo, su contenido se puede recorrer a través de sus claves.
- Aunque no es una colección, ofrece operaciones similares a las colecciones:
  - **Consulta**: `size()`, `isEmpty()`, **`containsKey(clave)`**, **`containsValue(valor)`**, **`get(clave) -> valor`**
  - **Modificación**: **`put(clave, valor)`**, **`remove(clave)`**, `clear()`, `putAll(otroMapa)`

# Mapas

- **Ejemplo**: construye un mapa que asocia palabras con el número de veces que se repiten en una secuencia

```
public static Map<String, Integer> contarPalabras(String... palabras) {

        HashMap<String, Integer> mapa = new HashMap<>();
        for (String palabra: palabras) {
            if (! mapa.containsKey(palabra)) { // primera aparición
                mapa.put(palabra, 1);
            }
            else { // suma 1 al contador
                int contador = mapa.get(palabra);
                mapa.put(palabra, contador + 1);
            }
        }
        return mapa;
}
```

# Mapas

- Aunque un mapa no es iterable, podemos recorrerlo a través de sus claves.
- El método **`keySet()`** retorna el **conjunto de claves** del mapa.
- Utilizando la clave, podemos recuperar el valor asociado con el método **`get(clave)`**.

```
Map<String, Integer> mapa =
          contarPalabras("hola", "Juan", "hola", "adiós");

Set<String> claves = mapa.keySet();

for (String clave : claves) {
    System.out.printf("Clave: %s -> Valor: %s \n",
                      clave, mapa.get(clave));
}
```

# Mapas

- También podemos obtener una colección con todos los valores que están asociados a claves del mapa.
- El método **`values()`** retorna una colección (interfaz `Collection`) con los valores.
- **Ejemplo**: calcular el mayor número de repeticiones del mapa.

```
Map<String, Integer> mapa =
   contarPalabras("hola", "Juan", "hola", "adiós");
Collection<Integer> valores = mapa.values();

int mayor = 0;
for (int valor : valores) {
   if (valor > mayor)
      mayor = valor;
}
```

# Mapas

- De forma análoga a como sucede con las colecciones, no podemos modificar un mapa mientras recorremos sus claves o valores.
- **Ejemplo**: eliminar asociaciones cuyo valor sea 1.

```
Map<String, Integer> mapa =
    contarPalabras("hola", "Juan", "hola", "adiós");

for (String clave : mapa.keySet()) {

    if (mapa.get(clave) == 1)
        mapa.remove(clave); // Se produce un error
}
```

- El fragmento de código anterior NO FUNCIONA.

# Mapas

- En un mapa, la modificación de sus colecciones de claves (`keySet`) o valores (`values`) **tiene efecto sobre el mapa**.
- Por tanto, si quitamos una clave de la colección de claves estaríamos quitando una entrada del mapa.
- De nuevo, utilizando un **iterador explícito** podríamos borrar durante el recorrido:

```
Map<String, Integer> mapa =
    contarPalabras("hola", "Juan", "hola", "adiós");

Iterator<String> iterador = mapa.keySet().iterator();

while (iterador.hasNext()) {
    String clave = iterador.next();
    if (mapa.get(clave) == 1)
        iterador.remove();
}
```

# Mapas

- **Ejemplo**: añadir nuevas entradas al mapa cuya clave sea igual a la original con el prefijo “>” y que mantenga el mismo valor.

```
Map<String, Integer> mapa =
      contarPalabras("hola", "Juan", "hola", "adiós");

HashMap<String, Integer> nuevasEntradas = new HashMap<>();
for (String clave : mapa.keySet()) {
   String nuevaClave = ">" + clave;
   nuevasEntradas.put(nuevaClave, mapa.get(clave));
}

mapa.putAll(nuevasEntradas);
```

# Mapas ordenados

- Al igual que los conjuntos, podemos optar por dos implementaciones de un mapa: `HashMap<K,V>` y `TreeMap<K,V>`, esta última implementa la interfaz **`SortedMap<K,V>`**.
- La diferencia fundamental es que un **`TreeMap<K,V>`** ordena las entradas según la clave.
- De este modo, al recorrer las claves del mapa (`keySet`) obtenemos las claves ordenadas.

```
public static SortedMap<String, Integer> contarPalabras(
                                                String... palabras) {
   TreeMap<String, Integer> mapa = new TreeMap<>();
   // El resto igual ...
   return mapa;
}
```

# Mapas ordenados

- La clase **`TreeMap<K,V>`** ofrece un constructor en el que podemos establecer un criterio de ordenación (`Comparator`).
- Resulta útil cuando queremos aplicar un orden a las claves distinto al orden natural de la clase que implementa las claves o bien si las claves no tienen orden.
- En el ejemplo las claves son cadenas (`String`) y le aplicamos un orden alfabético descendente.

```
public static SortedMap<String, Integer> contarPalabras(
                                                String... palabras) {

   TreeMap<String, Integer> mapa = new TreeMap<>(new OrdenInverso());

   // El resto igual ...
   return mapa;
}
```

# Mapas - Rendimiento

- El funcionamiento de los mapas es análogo al de los **conjuntos**.
- La clase `HashMap<K,V>` utiliza una implementación similar a `HashSet<T>`:
  - Por tanto, se utilizan del mismo modo los métodos `hashCode/equals` para determinar si una clave está en el mapa.
- La clase `TreeMap<K,V>` comparte implementación con `TreeSet<T>`:
  - Así pues, las operaciones para añadir entradas (`put`), buscar claves (`containsKey`) y eliminar entradas (`remove`) solo utilizan el criterio de ordenación (`Comparable` o `Comparator`). No se utiliza el método `equals`.

# Interfaces – métodos por defecto

- La introducción de métodos por defecto en las interfaces ha **enriquecido la funcionalidad de las colecciones** en Java 8.
- **Ejemplo**: nuevos métodos en los mapas

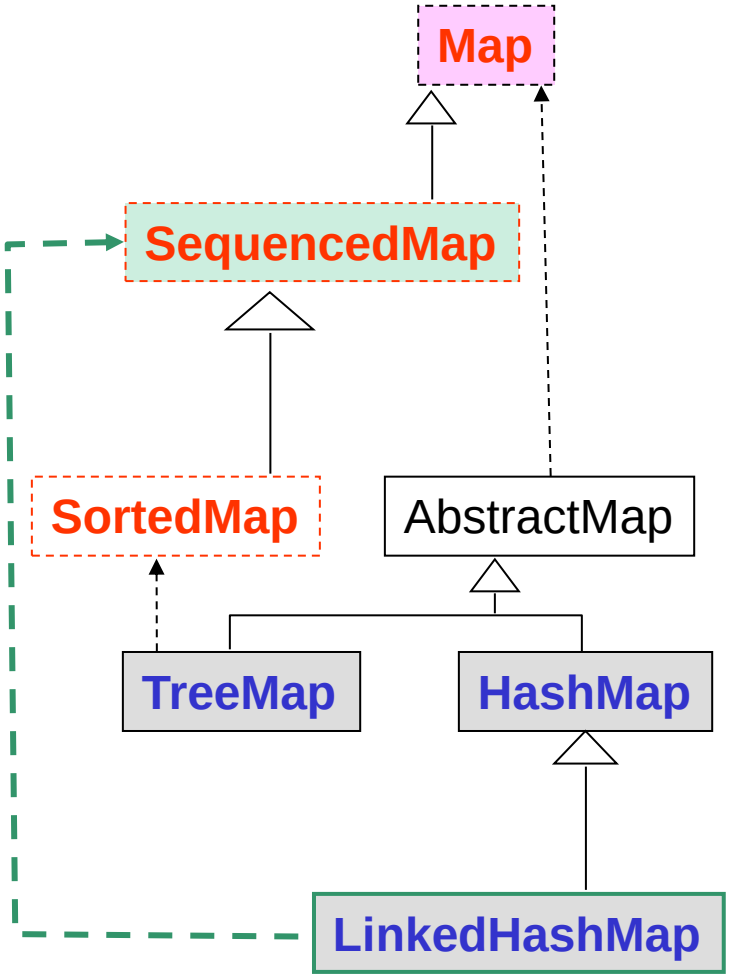
```
HashMap<String, Integer> mapa =
                new HashMap<String, Integer>();

mapa.putIfAbsent("juan", 10);

int valor = mapa.getOrDefault("pedro", 0);
```

  - El método `putIfAbsent` realiza la inserción si la clave no está previamente registrada en el mapa.
  - El método `getOrDefault` retorna un valor por defecto en el caso de no existir la clave en el mapa.

# Colecciones en Java (v2)




- En Java 21 se introduce la interfaz `SequencedMap`
- Ahora la Clase `LinkedHashMap` implementa esta interfaz.

# Interfaz `SequencedMap<E>`

```
interface SequencedMap<K,V> extends Map<K,V> {
    SequencedMap<K,V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K,V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

# Interfaz `SequencedMap<E>`

- Los métodos `sequencedKeySet`, `sequencedValues` y `sequencedEntrySet` son equivalentes los métodos `keySet()`, `values()` y `entrySet()`
  - Devuelven una vista de la colección como una colección secuenciada.
- Las modificaciones de las vistas son visibles en el mapa y viceversa.
- Las operaciones `add` y `addAll` sobre las vistas no están permitidas.
  - Ocurre la excepción `UnsupportedOperationException`

# Métodos de la clase Object

- Las clases que implementan las colecciones (`LinkedList<T>`, `HashSet<T>`, etc.) redefinen los métodos `equals`, `hashCode`, `toString` y `clone`.

- Las implementaciones de los métodos **`equals/hashCode`** es consistente.
- **Igualdad de listas**:
  - Dos listas son iguales si tienen el mismo tamaño y los elementos en cada posición son iguales (`equals`).
- **Igualdad de conjuntos**:
  - Dos conjuntos son iguales si tienen el mismo tamaño y todos los elementos de un conjunto están incluidos (`contains`) en el otro conjunto.

# Métodos de la clase Object

- La implementación del método **`toString`** muestra el contenido de la colección utilizando el método `toString` de sus elementos.
- El método **`clone`** solo puede ser utilizado cuando el tipo de la variable es una clase (por ejemplo, `LinkedList`) y no se puede utilizar si fuera una interfaz (por ejemplo, `List`):
  - Ninguno de los métodos de la clase `Object` están declarados en las interfaces.
  - Dado que `equals`, `hashCode` y `toString` son públicos, siempre están disponibles en cualquier objeto
  - Sin embargo, el método `clone` se hace público en la implementación de las clases.

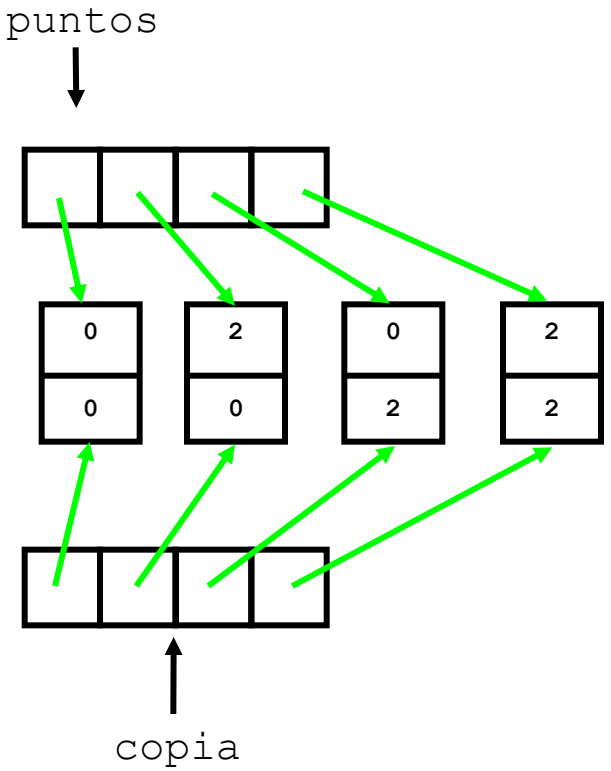
# Copia de colecciones

- Todas las clases que implementan colecciones ofrecen un constructor de copia y el método `clone`.
- En ambos casos construye una **copia superficial** del objeto receptor.

```
LinkedList<Punto> puntos;
…
LinkedList<Punto> copia;

// Opción 1: copia con clone
copia = (LinkedList<Punto>)puntos.clone();

// Opción 2: constructor de copia
copia = new LinkedList<Punto>(puntos);
```



# Aliasing

- En general, debemos evitar compartir las referencias a las colecciones:
  - Al recibirlas como parámetro de una operación.
  - En los métodos de consulta.
- **Solución 1**:
  - **Copiar la colección** (`clone` o constructor de copia).

```
// atributos
private LinkedList<Punto> vertices;
// ...

public List<Punto> getVertices() {
    return (List<Punto>) vertices.clone();
}
```

# Aliasing

- **Solución 2**:
  - Devolver una **vista no modificable** de la colección:

```
// atributos
private LinkedList<Punto> vertices;
// ...
public List<Punto> getVertices() {
   return Collections.unmodifiableList(vertices);
}
```

  - Es recomendable documentar que se devuelve una *vista* no modificable.

  - Es más eficiente que construir una copia.

# Aliasing

- **Solución 2**:
  - `Collections` proporciona una operación análoga para cada interfaz de las colecciones, incluidas las colecciones secuenciadas:
    - `unmodifiableSequenceCollection`
    - `unmodifiableList`,
    - `unmodifiableSequencedSet unmodifiableSet`, `unmodifiableSortedSet`,
    - `unmodifiableSequencedMap`, `unmodifiableMap`.

# Colecciones no modificables

- Las interfaces `List`, `Set` y `Map` proporcionan el método estático `of` para crear colecciones no modificables.
- No se pueden añadir, eliminar o reemplazar los elementos.
- Si los datos no se van a modificar son más eficientes que las colecciones modificables.
- Ejemplo:

```
List<String> cadenas = List.of("a", "b", "c");
```

equivale a:

```
List<String> cadenas = Arrays.asList("a", "b", "c");
cadenas = Collections.unmodifiableList(cadenas);
```

# Colecciones no modificables

- Las interfaces `List`, `Set` y `Map` proporcionan el método estático `copyOf` para crear una copia no modificable de una colección.
- Ejemplo:

```
List<Item> list = new ArrayList<>();
list.addAll(getItemsFromSomewhere());
list.addAll(getItemsFromElsewhere());
List<Item> snapshot = List.copyOf(list);
```

- Utilizar el método `of` no sería adecuado puesto que tendríamos que convertir la lista en un array previamente.
- Si la colección a copiar no es modificable devuelve la referencia a la colección.
- Los objetos contenidos en la colección no se copian.

# Colecciones vs. vistas no modificables

- Las colecciones no modificables se comportan igual que las vistas no modificables, pero no son vistas.

- Las colecciones no modificables son estructuras de datos implementadas por clases que son inmutables.
  - Los objetos que contienen si se pueden modificar.

- La colección subyacente a la vista no modificable SI se puede modificar.
  - El cambio de la colección se ve reflejado en la vista

# Recomendaciones

- **Programar hacia el tipo de datos**
  - En **constructores y métodos públicos**, el tipo de retorno y el tipo de los parámetros se especifica utilizando la interfaz (por ejemplo `List` en lugar de `LinkedList`)
  - Observa como el método `contarPalabras` presentado en las diapositivas anteriores declara retornar un mapa (`Map`) y no un `HashMap`.

```
public static Map<String, Integer> contarPalabras(String... palabras) {

    HashMap<String, Integer> mapa = new HashMap<>();
    // ...
    return mapa;
}
```

# Recomendaciones

- **Evitar el uso de arrays**
  - Los arrays tienen una funcionalidad limitada.
  - Además, las operaciones fundamentales de la clase `Object` no están redefinidas en los arrays.
  - Es necesario utilizar los métodos static de la clase `java.util.Arrays` para comparar por igualdad dos arrays (`equals`), obtener el código de dispesión (`hashCode`) y la representación textual (`toString`).
  - Podemos obtener una lista a partir de un array:

```
String[] array = {"a", "b", "c", "d"};
List<String> lista = Arrays.asList(array);
```

  - **Nota**: la lista obtenida es de solo consulta.

# Guía Rápida

- En el anexo del tema 4-1 está disponible una **guía rápida** de uso de las colecciones.

- Incluye los siguientes tópicos:
  - Uso de `LinkedList` (**listas**).
  - Uso de `HashSet` (**conjuntos**).
  - Uso de `HashMap` (**mapas**).
  - **Ordenación de listas**. Interfaces Comparable y Comparator.