Comparación, Ordenamiento y Colecciones en Java.

El **martes** tratamos conceptos clave en Java relacionados con **comparación, ordenamiento y uso de colecciones**. A continuación, les dejamos una versión organizada y limpia de lo que realizamos:

1. Comparación entre Objetos en Java.

1.1 Métodos Fundamentales.

a) `equals(Object obj)`

- Determina si dos objetos son iguales **en contenido**.

b) `hashCode()`

- Devuelve un valor `int` que representa el **hash del objeto**.
- Es utilizado para búsquedas y almacenamiento eficiente en estructuras como `HashSet`, `HashMap`.
- **Regla importante**:

  > Si `a.equals(b)` es `true`, entonces `a.hashCode() == b.hashCode()` debe cumplirse.

Ejemplo:

```
"dario".hashCode()              -> 1234
new String("dario").hashCode()  -> 1234
"dari".hashCode()               -> 8534
"darios".hashCode()             -> 6666
```

c) `compareTo(T o)` – Interface `Comparable<T>`

- Proporciona el **orden natural** entre objetos.
- Devuelve:
  - `-1` si `this < o`
  - `0` si `this == o`
  - `1` si `this > o`
- Usado en: `Collections.sort()`, `TreeSet`, etc.

d) `compare(T o1, T o2)` – Interface `Comparator<T>`

- Permite definir **otros criterios de ordenamiento**.
- Ideal cuando no se puede modificar la clase original (por ejemplo, librerías externas).

2. Colecciones en Java.

2.1 Colecciones Básicas (Admiten duplicados).

- `List` (`ArrayList`, `LinkedList`)
  - Permiten elementos duplicados.
  - Se pueden ordenar con `Collections.sort()` si los elementos implementan `Comparable` o usando un `Comparator`.

2.2 Colecciones Avanzadas (No admiten duplicados).

- `Set` – Basado en `equals()`
  - `HashSet`:
    - No garantiza orden.
  - `LinkedHashSet`:
    - Mantiene el orden de **inserción**.
  - `TreeSet`:
    - Ordena automáticamente (natural o con `Comparator`).
    - Usa internamente un **árbol rojo-negro (Red-Black Tree)**.
    - ⚠ Si los objetos no implementan `Comparable` o no se pasa un `Comparator`, lanza `ClassCastException`.

3. Consideraciones Prácticas.

- Convertir `Set` (sin duplicados) → `List` (permite duplicados): posible.
- Convertir `List` → `Set`: se **pierden elementos duplicados**.

Muy Importante:

Los métodos `equals` y `compareTo` deben ser **consistentes**.

Si:

```
a.equals(b) == true
```

entonces:

```
a.compareTo(b) == 0
```

De lo contrario, colecciones como `TreeSet` pueden comportarse de forma **inesperada**.

4. Ejemplo Práctico: Gestión de Clientes en un Bar.

Durante la clase trabajamos un ejercicio que puede formalizarse con el siguiente enunciado:

Enunciado.

El bar **"DE LOS NO HOMEROS"** desea implementar un sistema para gestionar a sus clientes. El sistema debe cumplir con los siguientes **requisitos funcionales**:

a) No se admiten clientes duplicados:

- Dos clientes con el **mismo nombre** se consideran iguales y **no deben repetirse** en la colección.

b) Los clientes deben almacenarse ordenados alfabéticamente por nombre:

- El orden debe ser automático, no basado en la inserción.

c) El cliente más adulto (mayor edad) debe recibir una cerveza gratis.

Requisitos Técnicos para el Desarrollo.

Se debe aplicar **TDD (Test Driven Development)** desde el comienzo.

- Crear una clase `Cliente` con al menos los atributos:
  - `String nombre`
  - `Integer edad`
- Implementar correctamente:
  - `equals()` y `hashCode()` → Comparación por **nombre** para evitar duplicados.
- Implementar `Comparable<Cliente>` o utilizar un `Comparator`:
  - Para ordenar a los clientes **por nombre**.
- Utilizar una colección adecuada:
  - Que **no admita duplicados**.
  - Que **mantenga el orden natural** automáticamente.
- Utilizar una colección adecuada:
  - Que **no admita duplicados**.
  - Que **mantenga el orden natural** automáticamente.
- Implementar una función que retorne el cliente más adulto de la colección