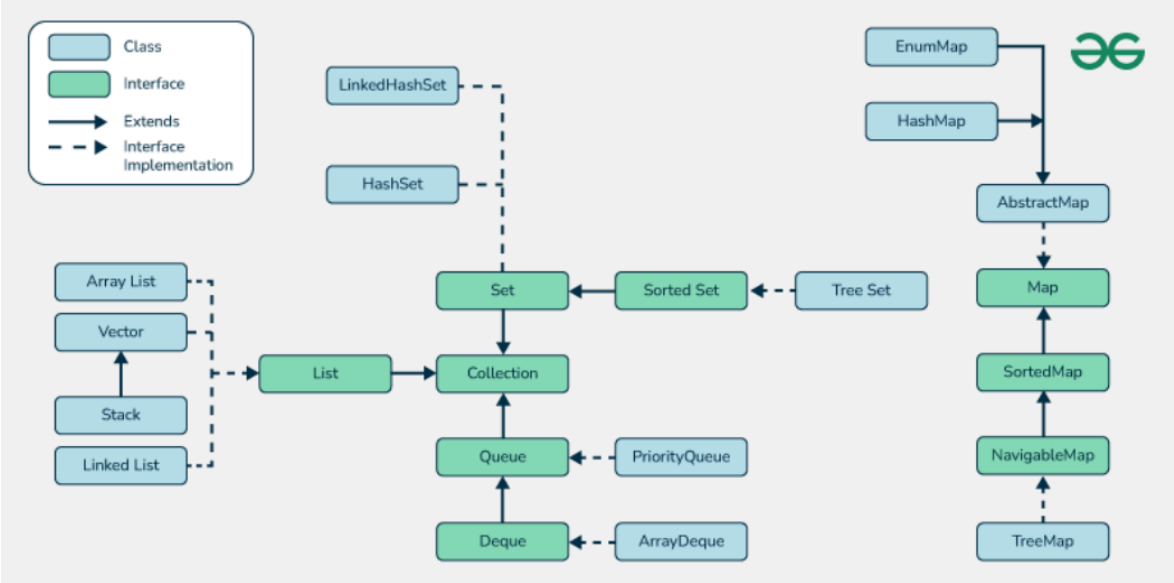
ESTRUCTURAS DINÁMICAS Y JAVA COLLECTIONS FRAMEWORK



1. CONCEPTOS BÁSICOS Y ESTRUCTURAS: ESTÁTICAS vs DINÁMICAS

1.1. ¿Qué es un dato?

Un dato es una unidad básica de información, que puede ser procesada por un sistema informático. Ya sean números enteros, decimales, texto etc..

1.2. ¿Qué es una estructura de datos?

Una estructura de datos es una forma de organizar y almacenar datos para que se puedan usar y modificar de manera eficiente. Modifican los datos de diferentes formas, ya sea en listas, pilas, colas o árboles.

1.3. Breve descripción de las principales estructuras dinámicas

Listas enlazadas: Son colecciones de elementos donde cada uno apunta al siguiente, permitiendo insertar y eliminar elementos fácilmente.

Pilas: Almacenan elementos en un orden específico, el último en entrar es el primero en salir (LIFO).

Colas: Almacenan elementos en un orden en el que el primero en entrar es el primero en salir (FIFO).

Árboles: Son estructuras jerárquicas que organizan los datos en nodos, con un nodo principal llamado raíz y nodos secundarios llamados hijos.

1.4. ¿Qué diferencia existe entre estructuras estáticas y dinámicas?

Estructuras estáticas: Tienen un tamaño fijo y no pueden cambiar una vez que se crean, como los arrays.

Estructuras dinámicas: Pueden cambiar de tamaño durante la ejecución del programa, como las listas enlazadas o las pilas.

Si se necesita velocidad y orden fijo, usa un **Array**. Si se busca flexibilidad y facilidad para modificar datos, mejor un **ArrayList**.

2. JERARQUÍA Y PRINCIPALES INTERFACES EN JAVA.UTIL

2.1. ¿Qué es la interfaz Collection y qué métodos básicos ofrece?

La interfaz **Collection** en Java es la base de todas las colecciones. Ofrece métodos básicos como:

1. **add(E e)**: Añade un elemento.
2. **remove(Object o)**: Elimina un elemento.
3. **size()**: Devuelve el número de elementos.
4. **isEmpty()**: Indica si la colección está vacía.
5. **contains(Object o)**: Verifica si un elemento está presente.
6. **clear()**: Elimina todos los elementos.

2.2. ¿Cuáles son las principales interfaces del framework y qué características tienen?

Collection: Es la interfaz base que agrupa todas las colecciones. Proporciona métodos para agregar, eliminar y consultar elementos.

List: Extiende de Collection y permite almacenar elementos en un orden específico. Los elementos pueden repetirse y se pueden acceder por su índice.

Set: Extiende de Collection y no permite elementos duplicados. Garantiza que cada elemento en el conjunto sea único.

Queue: Extiende de Collection y representa una colección que sigue el principio FIFO (First In, First Out), es decir, el primer elemento en entrar es el primero en salir.

Map: No extiende de Collection, pero es parte del framework. Representa un conjunto de pares clave-valor, donde cada clave es única.

2.3. ¿Qué diferencias fundamentales existen entre List, Set y Map?

List: Mantiene el orden y permite duplicados. Se usa cuando el orden y los elementos repetidos son importantes.
Set: No mantiene el orden (aunque algunas implementaciones sí), y no permite duplicados. Se usa cuando los elementos deben ser únicos.
Map: Guarda pares clave-valor, con claves únicas y valores repetibles. Se usa para asociar una clave a un valor.

3. IMPLEMENTACIONES COMUNES Y SU FUNCIONAMIENTO INTERNO

3.1. Explica brevemente qué es la complejidad temporal, qué significa Big O.

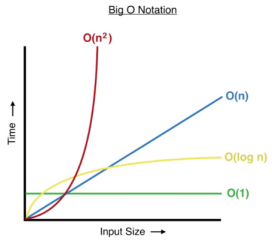
La complejidad temporal mide cuánto tiempo tarda un algoritmo en ejecutarse según el tamaño de los datos de entrada.

Big O es una forma de expresar cómo crece el tiempo de ejecución cuando aumentan los datos, usando notación matemática. Ejemplos:

O(1) → Tiempo constante, no cambia con el tamaño de los datos.

O(n) → Crece proporcionalmente al número de elementos.

O(n²) → Crece mucho más rápido a medida que los datos aumentan.



3.2. ArrayList:

Implementación interna ¿Qué significa que se base en un array dinámico?

Un ArrayList en Java está basado en un array dinámico, lo que significa que su tamaño puede crecer automáticamente cuando se llena.

¿Cuáles son sus tiempos de acceso y en qué casos es ideal utilizarlo?

Tiempo de acceso: Rápido (O(1)) al leer un elemento, pero lento (O(n)) al insertar o eliminar en el medio.
Cuándo usarlo: Cuando necesitas acceder rápido a los datos y no vas a modificar mucho la lista.

3.3. LinkedList:

Investiga la estructura de una LinkedList (lista doblemente enlazada). ¿Qué ventajas tiene en términos de inserciones y eliminaciones?

Una **LinkedList** es una lista doblemente enlazada, donde cada elemento está conectado con el anterior y el siguiente.

Ventajas en inserciones y eliminaciones: añadir o eliminar en cualquier posición es rápido, ya que solo cambia los enlaces y además no es necesario mover elementos, al contrario que un arraylist.

3.4. HashSet y TreeSet:

¿Cómo funcionan internamente un HashSet (tabla hash) y un TreeSet (árbol rojonegro)? ¿Por qué TreeSet puede ordenar sus elementos automáticamente?

HashSet usa una tabla hash para almacenar los elementos. Es muy rápido para buscar, añadir y eliminar (O(1)), pero no mantiene un orden específico.

TreeSet usa un árbol rojo-negro, una estructura ordenada. Gracias a esto, los elementos siempre están en orden y las operaciones tardan O(log n).

3.5. HashMap y TreeMap:

¿Cómo se organizan internamente los elementos en un HashMap y en un TreeMap? ¿Qué diferencias de rendimiento existen entre ambos.

El **HashMap** es rápido (O(1)) y no ordena los elementos. Es ideal si no te importa el orden.

El **TreeMap** es más lento (O(log n)) pero mantiene los elementos ordenados por clave. Es útil si necesitas orden.

3.6. PriorityQueue y ArrayDeque:

¿Cómo funcionan internamente PriorityQueue y ArrayDeque?.¿Cuándo conviene usar una PriorityQueue en lugar de una LinkedList o una ArrayDeque? Indica ventajas y desventajas concretas según casos de uso.

La **PriorityQueue** organiza los elementos por prioridad usando un heap, lo que la hace ideal para tareas donde el orden es importante, pero es más lenta al insertar o eliminar (O(log n)).

ArrayDeque, en cambio, usa un array dinámico y es muy eficiente para añadir y quitar elementos de ambos extremos (O(1)), pero no mantiene ningún orden de prioridad.

3.7. Colecciones Obsoletas:

¿Por qué se recomienda evitar el uso de Vector y Stack? ¿Qué alternativas se usan hoy en día?

Vector y Stack son obsoletos debido a su falta de eficiencia y sincronización innecesaria.
Alternativas:

ArrayList para listas dinámicas (sincronización opcional) y ArrayDeque para pilas y colas, ya que es más rápido y flexible.

3.8. Compara las complejidades de operaciones comunes (acceso, inserción, eliminación, búsqueda) entre distintas colecciones

Operación	ArrayList	LinkedList	HashMap	TreeMap	HashSet	TreeSet
Acceso	O(1)	O(n)	O(1)	O(log n)	O(1)	O(log n)
Inserción	O(1) (final)	O(1) (extremos)	O(1)	O(log n)	O(1)	O(log n)
Eliminación	O(1) (final)	O(1) (extremos)	O(1)	O(log n)	O(1)	O(log n)
Búsqueda	O(n)	O(n)	O(1)	O(log n)	O(1)	O(log n)

3.9. Investiga y explica brevemente cómo las distintas colecciones manejan los valores null.

HashMap y HashSet permiten null (una clave null en HashMap y un solo valor null en HashSet).

TreeMap y TreeSet no permiten null como claves, ya que no se pueden ordenar.

ArrayList y LinkedList permiten múltiples valores null.

PriorityQueue permite null si su comparador lo soporta; de lo contrario, lanza una excepción.

Vector y Stack permiten valores null sin restricciones.

3.10 Identifica algunas excepciones típicas al trabajar con colecciones. En qué consisten y como se podrían evitar.

ConcurrentModificationException: Se produce si se modifica una colección mientras se recorre. Hay que evitarlo usando métodos seguros como Iterator.remove() o colecciones concurrentes.

NullPointerException: Ocurre cuando se usa un objeto null. Hay que evitarlo verificando que no sea null antes de usarlo.

ClassCastException: Se lanza al intentar convertir un objeto a un tipo incorrecto. Hay que evitarlo usando instanceof o generics.

IndexOutOfBoundsException: Aparece al acceder a un índice fuera de rango. Hay que evitarlo asegurándose de que el índice sea válido.

3.11 Uso de equals() y hashCode() en colecciones:

Investiga qué papel tiene el método equals() en colecciones como ArrayList, LinkedList y PriorityQueue cuando se usan métodos como contains(), remove() o indexOf(). Investiga qué papel adicional juega el método hashCode() junto a equals() en colecciones basadas en tablas hash como HashSet y HashMap. ¿Qué diferencia hay respecto al primer grupo?

ArrayList, LinkedList y PriorityQueue:

Usan **equals()** en métodos como contains(), remove() y indexOf() para comparar elementos.
No usan **hashCode()** ya que no están basadas en tablas hash.

HashSet y HashMap:

Usan **hashCode()** para determinar dónde se almacenan los elementos en la tabla hash.
Si hay colisión (mismo **hashCode()**), usan **equals()** para confirmar si los elementos son iguales.
hashCode() mejora la eficiencia de operaciones como búsqueda, inserción y eliminación.

Diferencia:

ArrayList, LinkedList, y PriorityQueue solo dependen de equals().

HashSet y HashMap usan tanto hashCode() como equals() para mejorar el rendimiento.

3.12. Buenas Prácticas:

¿Por qué es recomendable declarar variables utilizando las interfaces (por ejemplo, List lista = new ArrayList<>();)?

Es mejor usar interfaces al declarar variables porque así el código es más flexible y fácil de modificar.

4. ITERACIÓN Y ELIMINACIÓN SEGURA

4.1. Iterable

¿Qué es la interfaz Iterable en Java y cuál es su papel fundamental en el recorrido de colecciones?

Iterable es una interfaz en Java que permite recorrer una colección de elementos de forma sencilla. Su papel fundamental es proporcionar el método **iterator()**, que devuelve un **Iterator**, permitiendo recorrer la colección con un bucle for-each. Es la base para estructuras como List, Set y Queue.

¿Qué método debe implementar cualquier clase que declare que es iterable y cuál es su función?

Cualquier clase que implemente Iterable debe definir el método: **Iterator<T> iterator()**

Su función es devolver un Iterator, que permite recorrer los elementos de la colección uno a uno. Gracias a este método, la colección se puede recorrer con un bucle **for-each.**

¿Qué ventajas tiene implementar Iterable en tus propias clases?

Implementar Iterable en una clase permite recorrer sus objetos con for-each, lo que hace el código más limpio. También facilita la compatibilidad con APIs de Java que trabajan con colecciones y permite personalizar cómo se recorren los elementos.

4.2. For y For-each:

¿Cuáles son las diferencias entre recorrer una colección con un bucle for tradicional(basado en índices) y un for-each?

El **for** tradicional permite acceder por índice y modificar posiciones, pero solo en estructuras indexadas.

El **for-each** es más limpio y seguro, pero no permite acceder a índices ni modificar la colección directamente.

¿Qué limitaciones tiene el for-each en comparación con el for tradicional, especialmente en lo que respecta a modificar la colección durante el recorrido? ¿Ventajas e inconvenientes tiene cada uno?

El **for-each** no permite modificar la colección mientras se recorre ni acceder a los índices, lo que puede causar errores como **ConcurrentModificationException.**

El **for** tradicional sí permite modificar elementos y recorrer en distintos órdenes, pero es más propenso a errores de índice.

For-each es más limpio y seguro; for tradicional es más flexible pero más propenso a fallos.

4.3. Iterator:

¿Qué es un Iterator y cuáles son sus métodos principales? Investiga por qué usar Iterator.remove() es la forma segura de eliminar elementos durante la iteración, evitando errores como ConcurrentModificationException.

Un Iterator es un objeto que permite recorrer una colección de forma segura. Sus métodos principales son:

hasNext(): Indica si hay más elementos.

next(): Devuelve el siguiente elemento.

remove(): Elimina el último elemento devuelto por next().

Usar **Iterator.remove()** evita **ConcurrentModificationException**, ya que elimina elementos de forma segura mientras se recorre la colección.

4.4. ListIterator:

¿Qué ventajas ofrece el ListIterator en comparación con el Iterator convencional? ¿Cómo permite recorrer una lista en ambas direcciones?

ListIterator mejora a **Iterator** porque permite recorrer la lista en ambas direcciones con **next()** y **previous()**, además de modificar elementos con **set()**. Esto lo hace más flexible para trabajar con listas.

4.5. Iterable vs Iterator:

Compara en una tabla, ¿Qué es? Método clave ¿Qué hace? Ejemplo uso para recorrer una colección ¿Dónde se usa?

Característica	Iterable	Iterator
¿Qué es?	Es una interfaz que permite recorrer una colección.	Es una interfaz que facilita la iteración sobre una colección.
Método clave	`iterator()`	hasNext(), next(), remove()

5. ORDENACIÓN Y COMPARACIÓN: COMPARABLE Y COMPARATOR

5.1. Comparable:

¿Qué hace la interfaz Comparable y cómo se utiliza el método compareTo()? ¿En qué se diferencia este método de equals()?

La interfaz Comparable permite definir el orden de los objetos en una clase. Su método **compareTo()** compara dos objetos y devuelve un valor **negativo, cero o positivo**, dependiendo de si el objeto es menor, igual o mayor que el otro.

La diferencia con equals() es que equals() verifica si los objetos son exactamente iguales, mientras que **compareTo()** establece un orden relativo entre ellos.

5.2. Comparator:

¿Cómo permite Comparator definir un orden personalizado para objetos? ¿Por qué podría ser necesario tener múltiples criterios de ordenación?

Comparator permite definir un orden personalizado implementando el método **compare()**. Es útil tener múltiples criterios de ordenación cuando se necesita ordenar los objetos de diferentes maneras, como por edad o por nombre.

5.3. Métodos de ordenación:

¿Cómo se utiliza Collections.sort() para ordenar una lista? ¿Qué rol juegan Comparable y Comparator en este proceso?

Collections.sort() ordena una lista en Java. Si los objetos implementan **Comparable**, se ordenan según **compareTo()**. Si no, se puede usar un **Comparator** para definir un orden personalizado.

5.4. Comparable vs Comparator:

Compara en una tabla, ¿Qué es? Método clave ¿Qué hace? Ejemplo uso con Collection.sort() ¿Dónde se usa?

Característica	Comparable	Comparator
¿Qué es?	Interfaz que define el orden natural de los objetos.	Interfaz que permite definir un orden personalizado.
Método clave	compareTo()	compare()
¿Qué hace?	Compara dos objetos para establecer su orden natural.	Compara dos objetos según un criterio personalizado.
Ejemplo con Collection.sort()	Collections.sort(list); (requiere que los objetos implementen Comparable)	Collections.sort(list, comparator); (requiere un Comparator)
¿Dónde se usa?	Se usa cuando se quiere un orden natural para los objetos.	Se usa cuando se necesita un orden diferente o personalizado.

6. COLECCIONES INMUTABLES

6.1. Métodos of()

¿Qué hacen métodos como List.of(), Set.of() y Map.of()?

List.of() crea una lista inmutable con los elementos que se le pasen.
Set.of() crea un conjunto inmutable sin elementos duplicados.
Map.of() crea un mapa inmutable con pares clave-valor.

¿Cuáles son las ventajas de trabajar con colecciones inmutables?

Ofrecen seguridad, seguridad en entornos concurrentes, mantenimiento más fácil y mejor rendimiento, ya que no se pueden modificar después de su creación.

6.2. Colecciones no modificables:

¿Cómo se crean usando Collections.unmodifiableList() y similares?

Las colecciones no modificables se crean usando métodos como **Collections.unmodifiableList()**, **unmodifiableSet()** y **unmodifiableMap()**. Estos métodos envuelven una colección existente y la hacen inmutable, es decir, no se pueden añadir, eliminar ni modificar elementos, aunque la colección original aún pueda ser modificable.

¿En qué se diferencian de las colecciones inmutables?

Colecciones no modificables: La colección original puede cambiar, pero la **copia inmutable** no permite modificaciones.
Colecciones inmutables: Ni la colección original ni la copia pueden modificarse.

7. USO DE GENÉRICOS EN COLECCIONES

7.1 Importancia de los Genéricos:

¿Por qué es fundamental usar genéricos en las colecciones?

Los genéricos son importantes porque aseguran tipos, eliminan castings y mejoran la legibilidad del código.

¿Qué ventajas ofrece en términos de seguridad en tiempo de compilación y legibilidad del código?

Seguridad en tiempo de compilación: Detectan errores de tipo antes de ejecutar el código, evitando errores en tiempo de ejecución.
Mejor legibilidad: Hace que el código sea más claro, ya que los tipos de datos son explícitos y no necesitan conversiones.

¿Qué problemas comunes evitan los genéricos en colecciones (por ejemplo, la excepción)

Los genéricos evitan errores de tipo, eliminan castings innecesarios y aumentan la seguridad en tiempo de compilación.

8. ENTREVISTA DE TRABAJO

Imagina que ya estás trabajando en una empresa y tu equipo necesita incorporar a una nueva persona. Tu tarea es diseñar una parte de la entrevista técnica centrada en colecciones en Java, ya que es un tema fundamental en vuestro día a día. Prepara una lista con las 25 preguntas más importantes que harías a esas personas para comprobar si realmente domina las colecciones en Java. Piensa en qué conceptos, usos prácticos, diferencias y buenas prácticas deberían conocer

1. ¿Qué es una colección en Java y por qué son importantes?
2. Explica la diferencia entre List, Set y Map en Java.
3. ¿Qué diferencias existen entre HashMap y TreeMap?
4. ¿Cómo se garantiza la unicidad de elementos en un Set?
5. ¿Cuáles son las diferencias entre ArrayList y LinkedList?
6. ¿Qué es un HashSet y cuándo sería más adecuado usarlo?
7. ¿Qué es un LinkedHashSet y en qué se diferencia de un HashSet?
8. ¿Qué tipo de ordenación mantiene un TreeSet?
9. ¿Cómo funciona el método compareTo() y en qué situaciones se utiliza?
10. ¿Qué es un Iterator y cómo se utiliza para recorrer colecciones?
11. ¿Cómo se puede eliminar un elemento de una colección mientras se está iterando sobre ella de manera segura?
12. ¿Qué diferencias existen entre el uso de un for-each y un Iterator para recorrer colecciones?
13. ¿Cómo se define un orden personalizado para una colección de objetos?
14. ¿Cuál es la diferencia entre Comparable y Comparator en Java?
15. ¿Cómo utilizarías Collections.sort() para ordenar una lista?
16. ¿Qué son las colecciones inmutables y cuáles son sus ventajas?
17. ¿Cómo se crean colecciones no modificables usando Collections.unmodifiableList()?
18. ¿Qué es la interfaz Iterable y por qué es importante en el contexto de colecciones?
19. ¿Cómo funcionan los genéricos en colecciones y por qué son útiles?
20. ¿Cuáles son las ventajas de usar colecciones genéricas?
21. ¿Qué es la excepción ConcurrentModificationException y cómo se puede evitar al trabajar con colecciones?

22. ¿Qué son las colecciones sincronizadas y cuándo se deben utilizar?
23. Explica cómo se manejan los valores null en las colecciones de Java.
24. ¿Cuándo es recomendable usar un ArrayDeque en lugar de un LinkedList?
25. ¿Cómo funcionan los métodos List.of(), Set.of() y Map.of()?