Java Collections

Bryan Ivan Montiel Ortega

Noviembre 2023

Índice

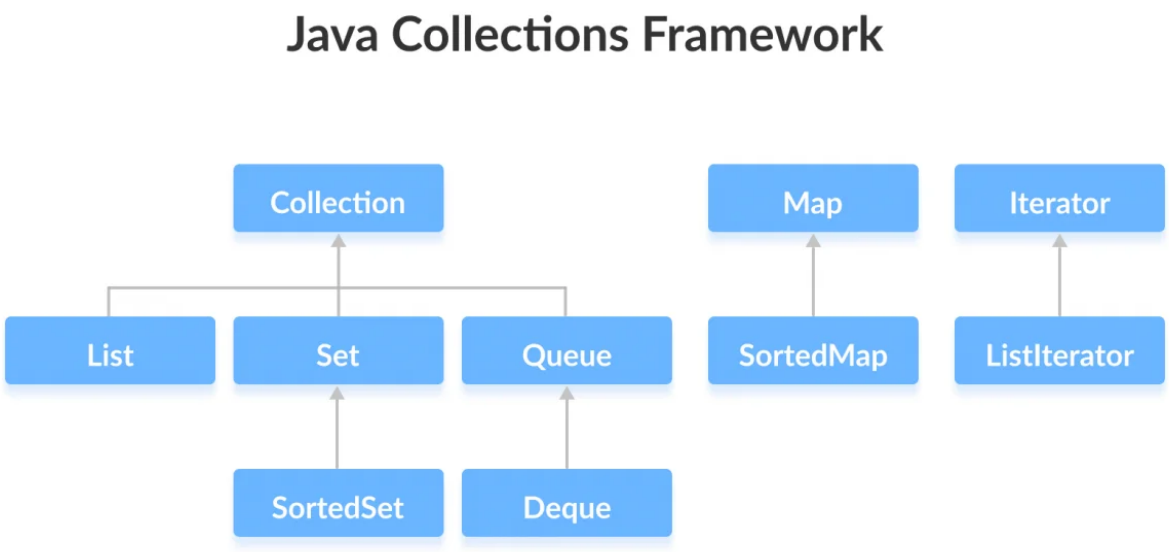
Java Collections Framework

Proporciona un conjunto de interfaces y clases para implementar diversas estructuras de datos y algoritmos.

Por ejemplo, la clase `LinkedList` del marco de colecciones proporciona la implementación de la estructura de datos de lista doblemente vinculada.

Interfaces de Collections FrameWork

El framwork de colecciones proporciona varias interfaces. Estas interfaces incluyen varios métodos para realizar diferentes operaciones en colecciones.



Java Collection Interface

La interfaz `Collection` es la interfaz raíz de la jerarquía del marco de recopilaciones.

Java no proporciona implementaciones directas de la interfaz `Collection`, pero proporciona implementaciones de sus subinterfaces como `List`, `Set` y `Queue`.

Collections Framework Vs. Collection Interface

La gente a menudo se confunde entre el framework de colecciones y la interfaz Collection.

La interfaz `Collection` es la interfaz raíz del marco de recopilaciones. El framework también incluye otras interfaces: `Map` y `Iterator`. Estas interfaces también pueden tener subinterfaces.

Subinterfaces de Collection Interface

Como se mencionó anteriormente, la interfaz `Collection` incluye subinterfaces que son implementadas por clases Java.

Todos los métodos de la interfaz `Collection` también están presentes en sus subinterfaces.

Estas son las subinterfaces de la interfaz `Collection`.

List Interface

La interfaz `List` es una colección ordenada que nos permite agregar y eliminar elementos como una matriz.

La colección más básica de Java. También, es la más usada por los programadores que no han investigado el framework de colecciones a fondo por hacernos pensar que se trata de una especie de '`array hipervitaminado`' ya que hace su trabajo y es fácil de entender.

Sabiendo los requisitos de tu aplicación, seguramente encontrarás otra colección que haga el trabajo de una lista mejor. Procura asegurarte de que cuando vas a usar una lista es porque realmente te hace falta, no porque no conoces el resto. La diferencia de rendimiento puede ser enorme en según qué operaciones.

¿Qué beneficios tienen las listas?

- Están ordenadas (Podemos usar Colections.sort() para ordenar los elementos siguiendo el criterio que queramos).
- Podemos añadir / eliminar elementos sin ninguna restricción.
- Tienen un iterador especial `ListIterator` que permite modificar la lista en cualquier dirección.
- Siguen la notación de los arrays, por lo que son fáciles de comprender.

¿Qué problemas tienen las listas?

- Bajo rendimiento en operaciones especializadas respecto a otras colecciones.

Tipos de Lista

ArrayList: Muy rápida accediendo a elementos, relativamente rápida agregando elementos si su capacidad inicial y de crecimiento están bien configuradas. Es la lista que deberías usar casi siempre.

LinkedList: Una lista que también es una cola. Más rápida que `ArrayList` añadiendo elementos en su principio y eliminando elementos en general. Utilízala en lugar de `ArrayList` si realizas más operaciones de inserción (en posición 0) o de eliminación que de lectura. La diferencia de rendimiento es enorme.

Vector: Terreno peligroso. Vector es una colección `deprecated` (obsoleta), así que usarla únicamente si necesitas un ArrayList concurrente. El rendimiento de Vector es superior al de `Collections.syncronizedList(new ArrayList())`.

CopyOnWriteArrayList: Colección concurrente que es muy poco eficiente en operaciones de escritura, pero muy rápida en operaciones de lectura. Úsala sobre Vector (o `synced ArrayList`) cuando el número de lecturas concurrentes sea mucho mayor al número de escrituras.

Set Interface

La interfaz `Set` nos permite almacenar elementos en diferentes conjuntos similares al conjunto de las matemáticas. No puede tener elementos duplicados.

Los sets, o conjuntos, son colecciones que por norma general no admiten elementos iguales en su interior. Dos elementos `A` y `B` son iguales si `A.equals(B)`.

Podemos añadir y eliminar elementos de un set, así como saber si determinado elemento está en un set, pero no tenemos acceso aleatorio, por lo que hay que tener muy claro cuando queremos usarlos.

Una colección normal, en apariencia, nos da todo lo anterior -excepto el hecho de eliminar duplicados-, así que, ¿por qué usar un Set y no una Lista?, el motivo es simple: **eficiencia**.

¿Qué beneficios tienen los sets?

- No permite elementos duplicados.
- Implementación muy eficiente de `.add` para asegurarnos de que no hay duplicados.

¿Qué desventajas tienen?

- No tienen acceso aleatorio.

Solo algunos tipos de set pueden ordenarse y lo hacen de forma poco eficiente.

Tipos de Set

HashSet: La implementación más equilibrada de la interfaz Set. Es rápida y no permite duplicados, pero no tiene ningún tipo de ordenación. Utilízala si necesitas un control de duplicados pero no ningún tipo de ordenación o acceso aleatorio.

LinkedHashSet: Un HashSet que incluye ordenación de elementos por orden de entrada, una velocidad de iteración mucho mayor y un rendimiento mucho peor a la hora de añadir elementos. Utilízala si necesitas un Set ordenado por orden de inserción o si vas a usar un Set que vaya a realizar solamente operaciones de iteración.

TreeSet: Un set que incluye una implementación de un árbol rojo-negro. Este Set puede ser ordenado, pero su rendimiento es mucho peor en cualquier operación (menos `iteración`) respecto a un HashSet.

Utilízalo solo si necesitas un Set con un criterio de ordenación específico y ten cuidado con las inserciones.

EnumSet: La mejor implementación de Set para tipos enumerados (Enum). Utilízala si quieres crear un Set de Enums.

CopyOnWriteArraySet: Set concurrente que tiene un gran rendimiento de lectura, pero pésimo de escritura, eliminado y contains. Úsalo solo en Sets concurrentes que apenas tengan estas operaciones.

ConcurrentSkipListSet: Set concurrente y ordenable. Utilízalo sólo cuando requieras un Set ordenable (como TreeSet) en entornos de concurrencia. En Sets de tamaños muy grandes su rendimiento empeora notablemente.

Queue Interface

La interfaz `Queue` se utiliza cuando queremos almacenar y acceder a elementos de la manera **First In, First Out**.

Las `queues` son estructuras que ofrecen un gran rendimiento al obtener elementos de su principio o de su final, representando colas `LIFO / FIFO`.

Deberás usar una `queue` cuando vayas a recuperar siempre el primer o último elemento de una serie. Se usan para implementar las mencionadas colas `LIFO / FIFO`, así como colas de prioridades (como puede ser un sistema de tareas o de procesos).

Cabe destacar que hay dos tipos de `queues`, `Queues` y `Deques`. Las primeras sólo proporcionan métodos para acceder al último elemento de la cola, mientras que las `Deques` permiten acceder a cualquiera de los dos extremos.

¿Qué ventajas tienen las colas?

- Ofrecen un gran rendimiento al recuperar el primer o último objeto de la cola.
- Permiten crear estructuras LIFO / FIFO o colas de prioridades con muy buen rendimiento.

¿Qué desventajas tienen las colas?

- La iteración por las colas suele ser muy lenta.
- El acceso aleatorio, de la misma manera, es muy lento.

Tipos de colas

ArrayDeque: Una implementación de Deque de rendimiento excepcional. Implementa tanto cola LIFO como FIFO al ser una Deque y es la Cola que deberías usar si quieres implementar una de estas dos estructuras.

LinkedBlockingDeque: Una Deque concurrente que has de usar cuando quieras usar un ArrayDeque en entornos multihilo.

LinkedList: LinkedList, anteriormente mencionada en la sección de listas, también es una Deque, sin embargo, su rendimiento es muy inferior al de ArrayDeque. No deberías usar LinkedList cuando quieras usar una cola.

PriorityQueue: Una cola que se ordena mediante un Comparator, permitiendo crear una Cola donde el primer elemento no dependerá de su tiempo de inserción, sino de cualquier otro factor (tamaño, prioridad, etc). Deberemos usarlo cuando necesitemos este comparator, ya que ArrayDeque no lo permite.

PriorityBlockingQueue: La versión concurrente de PriorityQueue.

Why the Collections Framework?

El marco de compilaciones de Java proporciona varias estructuras de datos y algoritmos que se pueden utilizar directamente. Esto tiene dos ventajas principales:

- No tenemos que escribir código para implementar estas estructuras de datos y algoritmos manualmente.
- Nuestro código será mucho más eficiente ya que el marco de cobros está altamente optimizado.

Además, el marco de colecciones nos permite utilizar una estructura de datos específica para un tipo particular de datos. Estos son algunos ejemplos,

- Si queremos que nuestros datos sean únicos, podemos usar la interfaz `Set` proporcionada por el marco de colecciones.
- Para almacenar datos en pares **key/value**, podemos usar la interfaz.Map
- La clase `ArrayList` proporciona la funcionalidad de matrices de tamaño variable.

Interface Map de Java

La interfaz `Map` del framework de colecciones de Java proporciona la funcionalidad de la estructura de datos de mapa.

Los maps son colecciones que asocian un valor con una clave. Tanto la clave como el valor pueden ser cualquier tipo de datos de Java: Objetos, primitivos, otras colecciones, etc.

Las implementaciones son muy parecidas a los Sets debido a que, internamente, utilizan un Set (la implementación varía según el tipo de Map) para garantizar que no hay elementos duplicados en las claves.

¿Qué ventajas tienen los sets?

- Asociación clave -> valor.

- Gracias a que utilizan internamente un Set, garantizan que no habrá dos claves iguales.
- Es fácil reconocer cuando necesitamos usar un Map.

¿Qué desventajas tienen los sets?

- Rendimiento no muy elevado comparado con otras colecciones.

Tipos de Map

HashMap: La implementación más genérica de Map. Un array `clave->valor` que `no garantiza el orden de las claves` (de la misma forma que un HashSet). Si necesitas un Map no-concurrente que no requiera ordenación de claves, este es el tuyo. El código de HashSet, utiliza un HashMap internamente.

LinkedHashMap: Implementación de map que garantiza orden de claves por su tiempo de inserción; es decir, las claves que primero se creen serán las primeras. También puede configurarse para que la orden de claves sea por tiempo de acceso (las claves que sean accedidas precederán a las que no son usadas). Itera más rápido que un HashMap, pero inserta y elimina mucho peor. Utilizalo cuando necesites un HashMap ordenado por orden de inserción de clave.

TreeMap: Un Map que permite que sus claves puedan ser ordenadas, de la misma forma que TreeSet. Usalo en lugar de un HashMap solo si necesitas esta ordenación, ya que su rendimiento es mucho peor que el de HashMap en casi todas las operaciones (excepto iteración).

EnumMap: Un Map de alto rendimiento cuyas claves son Enumeraciones (Enum). Muy similar a EnumSet. Usadlo si vais a usar Enums como claves.

WeakHashMap: Un Map que solo guarda referencias blandas de sus claves y valores. Las referencias blandas hacen que cualquier clave o valor sea elegible por el recolector de basura si no hay ninguna otra referencia al mismo desde fuera del WeakHashMap.

Usa este Map si quieres usar esta característica, ya que el resto de operaciones tienen un rendimiento pésimo. Comúnmente usado para crear registros que vayan borrando propiedades a medida que el sistema no las vaya necesitando y vaya borrando sus referencias.

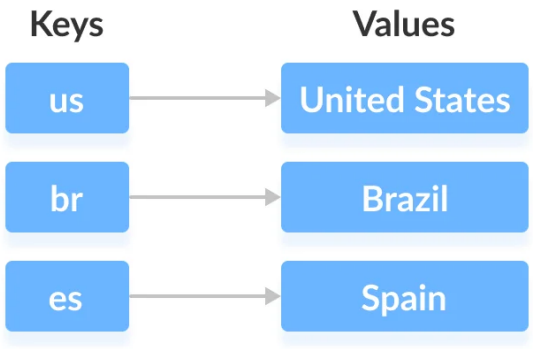
HashTable: Map deprecated y concurrente. Básicamente, es un HashMap concurrente que no debes usar nunca. En su lugar, utiliza ConcurrentHashMap.

ConcurrentHashMap: Un Map concurrente que no permite valores nulos. Sustitución básica de HashTable. Usala si necesitas un HashMap concurrente.

Funcionamiento del Map

En Java, los elementos `Map` de se almacenan en pares clave/valor. Las `keys` son valores únicos asociados a `values` individuales.

Un `Map` no puede contener claves duplicadas. Además, cada clave está asociada a un único valor.



Podemos acceder y modificar valores utilizando las claves asociadas a ellos.

En el diagrama anterior, tenemos los valores: `United States`, `Brazil`, and `Spain`. Y tenemos las claves correspondientes: `us`, `br`, and `es`.

Ahora, podemos acceder a esos valores usando sus claves correspondientes.

Nota: La interfaz `Map` mantiene 3 conjuntos diferentes:

- El set de keys
- El set de values
- El set/value de associations (asignación).

Por lo tanto, podemos acceder a claves, valores y asociaciones individualmente.

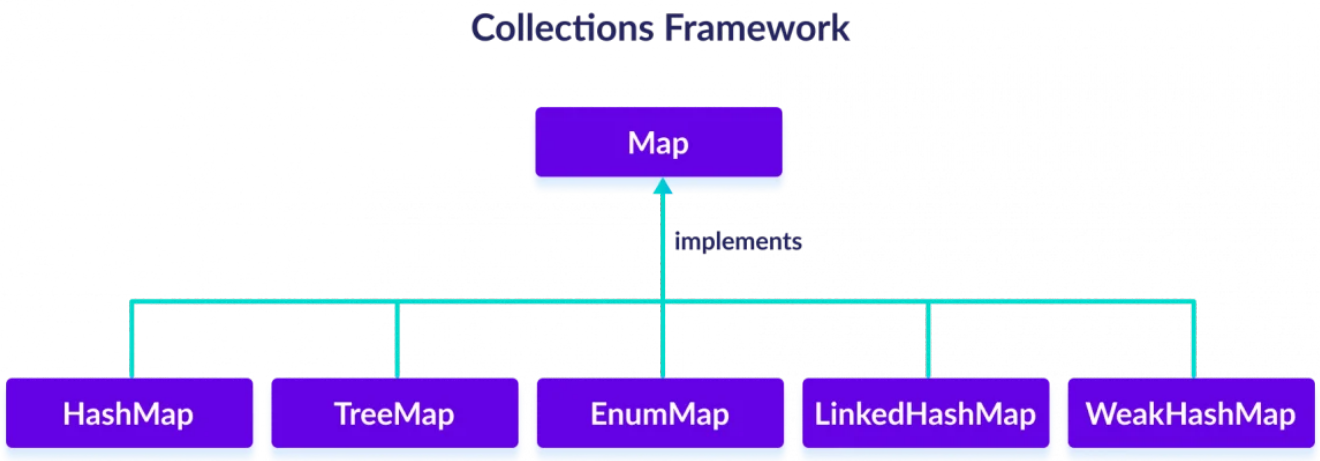
Clases que `'implements'` Map

Dado que `Map` es una interfaz, no podemos crear objetos a partir de ella.

Para utilizar las funcionalidades de la interfaz `Map`, podemos utilizar estas clases:

- HashMap
- EnumMap
- LinkedHashMap
- WeakHashMap
- TreeMap

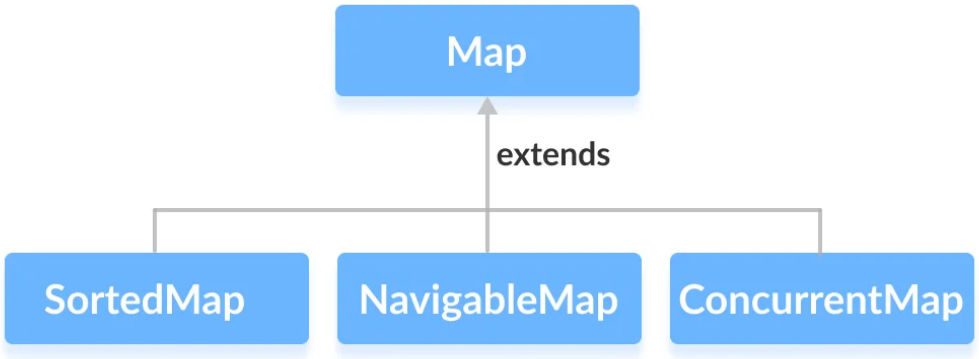
Estas clases se definen en el marco de colecciones e implementan la interfaz `Map`.



Interfaces que `'extends'` Map

La interfaz `Map` también se amplía mediante estas subinterfaces:

- SortedMap
- NavigableMap
- ConcurrentMap



¿Cómo usar Map?

En Java, debemos importar el paquete `'mapa.java.util.Map'` para poder usar `Map`. Una vez que importamos el paquete, así es como podemos crear un map:

```
// Map implementation using HashMap
Map<Key, Value> numbers = new HashMap<>();
```

En el código anterior, hemos creado un `Map` llamado `numbers`. Hemos usado la clase `HashMap` para implementar la interfaz `Map`.

Aquí

- `Key` - Un identificador único utilizado para asociar cada elemento (valor) en un mapa
- `Value` - Elementos asociados por claves en un mapa

Métodos de Map

La interfaz `Map` incluye todos los métodos de la interfaz `Collection`. Esto se debe a que `Collection` es una súper interfaz de `Map`.

Además de los métodos disponibles en la interfaz `Collection`, la interfaz `Map` también incluye los siguientes métodos:

- **put(K, V)** - Inserta la asociación de una clave y un valor en el mapa. Si la clave ya está presente, el nuevo valor reemplaza al valor anterior.KV
- **putAll()** - Inserta todas las entradas del mapa especificado en este mapa.
- **putIfAbsent(K, V)** - Inserta la asociación si la clave aún no está asociada con el valor .KV
- **get(K)** - Devuelve el valor asociado con la clave especificada . Si no se encuentra la clave, devuelve .Knull
- **getOrDefault(K, defaultValue)**: devuelve el valor asociado a la clave especificada. Si no se encuentra la clave, devuelve el archivo .KdefaultValue
- **containsKey(K)** - Comprueba si la clave especificada está presente en el mapa o no.K
- **containsValue(V)** - Comprueba si el valor especificado está presente en el mapa o no.V
- **replace(K, V)** - Reemplaza el valor de la clave con el nuevo valor especificado .KV
- **replace(K, oldValue, newValue)**: reemplaza el valor de la clave con el nuevo valor solo si la clave está asociada con el valor.KnewValueKoldValue
- **remove(K)** - Elimina la entrada del mapa representada por la clave .K
- **remove(K, V)** - Elimina la entrada del mapa que tiene la clave asociada con el valor .KV
- **keySet()** - Devuelve un conjunto de todas las claves presentes en un mapa.
- **values()** - Devuelve un conjunto de todos los valores presentes en un mapa.
- **entrySet()** - Devuelve un conjunto de todas las asignaciones clave/valor presentes en un mapa.

Los siguientes son espacios para colecciones más usadas o que he llegado a ver más durante los ejercicios de los simuladores.

LinkedList

`LinkedList` es una implementación de lista doblemente enlazada de las interfaces `List` y `Deque`. Implementa todas las operaciones de lista opcionales y permite todos los elementos (incluido `null`).

Características

A continuación puede encontrar las propiedades más importantes de LinkedList:

- Las operaciones que se indexan en la lista recorrerán la lista desde el principio o el final, lo que esté más cerca del índice especificado
- No está **sincronizado**
- Sus iteradores `Iterator` y `ListIterator` son `fail-fast` (lo que significa que después de la creación del iterador, si se modifica la lista, se producirá `ConcurrentModificationException`)
- Cada elemento es un nodo, que mantiene una referencia a los siguientes y anteriores
- Mantiene el orden de inserción

Aunque `LinkedList` no está sincronizado, podemos recuperar una versión sincronizada del mismo llamando al método Collections.synchronizedList, como:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

Comparación con ArrayList

Aunque ambos implementan la interfaz List, tienen una semántica diferente, lo que definitivamente afectará la decisión de cuál usar.

Structure

`ArrayList` es una estructura de datos basada en índices respaldada por una matriz. Proporciona acceso aleatorio a sus elementos con un rendimiento igual a O(1).

Por otro lado, una `LinkedList` almacena sus datos como una lista de elementos y cada elemento está vinculado a su elemento anterior y siguiente. En este caso, la operación de búsqueda de un elemento tiene un tiempo de ejecución igual a O(n).

Operaciones

Las operaciones de inserción, adición y eliminación de un elemento son más rápidas en una `LinkedList` porque no es necesario cambiar el tamaño de una matriz o actualizar el índice cuando se

agrega un elemento a alguna posición arbitraria dentro de la colección, solo cambiarán las referencias en los elementos circundantes.

Uso de memoria

Un `LinkedList` consume más memoria que un `ArrayList` porque cada nodo de un `LinkedList` almacena dos referencias, una para su elemento anterior y otra para su elemento siguiente, mientras que `ArrayList` solo contiene datos y su índice.

Uso

Estos son algunos ejemplos de código que muestran cómo puede usar `LinkedList`:

Creación

```
LinkedList<Object> linkedList = new LinkedList<>();
```

Adición de elementos

`LinkedList` implementa la interfaz `List` y `Deque`, además de los métodos estándar `add()` y `addAll()`, puede encontrar `addFirst()` y `addLast()`, que agrega un elemento al principio o al final, respectivamente.

Eliminación del elemento

De manera similar a la adición de elementos, esta implementación de lista ofrece removeFirst() y removeLast().

Además, hay un método conveniente removeFirstOccurence() y removeLastOccurence() que devuelve booleano (true si la colección contenía un elemento especificado).

Operaciones de cola

La interfaz de Deque proporciona comportamientos similares a los de la cola (en realidad, Deque extiende la interfaz de la cola):

- `linkedList.poll();`
- `linkedList.pop();`

Esos métodos recuperan el primer elemento y lo eliminan de la lista.

La diferencia entre poll() y pop() es que pop lanzará NoSuchElementException() en una lista vacía, mientras que poll devuelve null. Las API pollFirst() y pollLast() también están disponibles.

A continuación, se muestra por ejemplo cómo funciona la API push:

- `linkedList.push(Object o);`

Que inserta el elemento como cabecera de la colección.

LinkedList tiene muchos otros métodos, la mayoría de los cuales deberían ser familiares para un usuario que ya usó Listas. Otros que son proporcionados por Deque pueden ser una alternativa conveniente a los métodos "estándar".

ArrayList

ArrayList reside dentro de las bibliotecas principales de Java, por lo que no necesita ninguna biblioteca adicional. Para usarlo, simplemente agregue la siguiente declaración de importación:

```
import java.util.ArrayList;
```

La lista representa una secuencia ordenada de valores en la que algún valor puede aparecer más de una vez.

ArrayList es una de las implementaciones de List creadas sobre una matriz, que puede crecer y reducirse dinámicamente a medida que agrega o elimina elementos. Se puede acceder fácilmente a los elementos mediante sus índices a partir de cero. Esta implementación tiene las siguientes propiedades:

- El acceso aleatorio tarda O(1) tiempo
- La adición del elemento toma un tiempo constante amortizado O(1)
- La inserción/eliminación tarda O(n) tiempo
- La búsqueda tarda O(n) tiempo para una matriz sin ordenar y O(log n) para una ordenada

Crear una ArrayList

`ArrayList` tiene varios constructores.

En primer lugar, ten en cuenta que `ArrayList` es una clase genérica, por lo que puedes parametrizarla con cualquier tipo que desees y el compilador se asegurará de que, por ejemplo, no puedas poner valores Integer dentro de una colección de Strings. Además, no es necesario convertir elementos al recuperarlos de una colección.

En segundo lugar, es una buena práctica utilizar la interfaz genérica `List` como un tipo de variable, ya que la desacopla de una implementación determinada.

Constructor no-arg por defecto

```
List<String> list = new ArrayList<>();
assertTrue(list.isEmpty());
```

Simplemente estamos creando una instancia vacía de `ArrayList`.

Constructor que acepta la capacidad inicial

```
List<String> list = new ArrayList<>(20);
```

Aquí se especifica la longitud inicial de una matriz subyacente. Esto puede ayudarlo a evitar cambios de tamaño innecesarios al agregar nuevos elementos.

Constructor que acepta el cobro

```
Collection<Integer> numbers
  = IntStream.range(0, 10).boxed().collect(toSet());

List<Integer> list = new ArrayList<>(numbers);
assertEquals(10, list.size());
assertTrue(numbers.containsAll(list));
```

Tenga en cuenta que ese elemento de la instancia de Collection se utiliza para rellenar la matriz subyacente.

Agregar elementos a ArrayList

Puede insertar un elemento al final o en la posición específica:

```
List<Long> list = new ArrayList<>();

list.add(1L);
list.add(2L);
list.add(1, 3L);

assertThat(Arrays.asList(1L, 3L, 2L), equalTo(list));
```

También puede insertar una colección o varios elementos a la vez:

```
List<Long> list = new ArrayList<>(Arrays.asList(1L, 2L, 3L));
LongStream.range(4, 10).boxed()
  .collect(collectingAndThen(toCollection(ArrayList::new), ys -> list.addAll(0, ys)));
assertThat(Arrays.asList(4L, 5L, 6L, 7L, 8L, 9L, 1L, 2L, 3L), equalTo(list));
```

Iterar sobre ArrayList

Hay dos tipos de iteradores disponibles: `Iterator` y `ListIterator`.

Mientras que el primero le da la oportunidad de recorrer la lista en una dirección, el segundo le permite atravesarla en ambas direcciones.

Aquí se muestra el `ListIterator`:

```
List<Integer> list = new ArrayList<>(
  IntStream.range(0, 10).boxed().collect(toCollection(ArrayList::new))
);
```

```
ListIterator<Integer> it = list.listIterator(list.size());
List<Integer> result = new ArrayList<>(list.size());
while (it.hasPrevious()) {
    result.add(it.previous());
}

Collections.reverse(list);
assertThat(result, equalTo(list));
```

Buscar en la lista de matrices

Demostraremos cómo funciona la búsqueda utilizando una colección:

```
List<String> list = LongStream.range(0, 16)
  .boxed()
  .map(Long::toHexString)
  .collect(toCollection(ArrayList::new));
List<String> stringsToSearch = new ArrayList<>(list);
stringsToSearch.addAll(list);
```

Búsqueda en una lista sin clasificar

Para encontrar un elemento, puede usar los métodos indexOf() o lastIndexOf(). Ambos aceptan un objeto y devuelven un valor int:

```
assertEquals(10, stringsToSearch.indexOf("a"));
assertEquals(26, stringsToSearch.lastIndexOf("a"));
```

Si desea encontrar todos los elementos que satisfacen un predicado, puede filtrar la colección usando Java 8 Stream API usando Predicate de la siguiente manera:

```
Set<String> matchingStrings = new HashSet<>(Arrays.asList("a", "c", "9"));

List<String> result = stringsToSearch
  .stream()
  .filter(matchingStrings::contains)
  .collect(toCollection(ArrayList::new));

assertEquals(6, result.size());
```

También es posible utilizar un bucle for o un iterador:

```
Iterator<String> it = stringsToSearch.iterator();
Set<String> matchingStrings = new HashSet<>(Arrays.asList("a", "c", "9"));

List<String> result = new ArrayList<>();
while (it.hasNext()) {
    String s = it.next();
    if (matchingStrings.contains(s)) {
```

```
        result.add(s);
    }
}
```

Búsqueda en una lista ordenada

Si tiene una matriz ordenada, puede usar un algoritmo de búsqueda binaria que funciona más rápido que la búsqueda lineal:

```
List<String> copy = new ArrayList<>(stringsToSearch);
Collections.sort(copy);
int index = Collections.binarySearch(copy, "f");
assertThat(index, not(equalTo(-1)));
```

Tenga en cuenta que si no se encuentra un elemento, se devolverá -1.

Eliminar elementos de la lista de matrices

Para eliminar un elemento, debe encontrar su índice y solo entonces realizar la eliminación a través del método `remove()`. Una versión sobrecargada de este método, que acepta un objeto, lo busca y realiza la eliminación de la primera aparición de un elemento igual:

```
List<Integer> list = new ArrayList<>(
  IntStream.range(0, 10).boxed().collect(toCollection(ArrayList::new))
);
Collections.reverse(list);

list.remove(0);
assertThat(list.get(0), equalTo(8));

list.remove(Integer.valueOf(0));
assertFalse(list.contains(0));
```

Pero tenga cuidado al trabajar con tipos en caja como Integer. Para eliminar un elemento en particular, primero debe encuadrar int value o, de lo contrario, un elemento será eliminado por su índice.

También puede usar la API de Stream mencionada anteriormente para eliminar varios elementos, pero no lo mostraremos aquí. Para ello utilizaremos un iterador:

```
Set<String> matchingStrings
 = HashSet<>(Arrays.asList("a", "b", "c", "d", "e", "f"));

Iterator<String> it = stringsToSearch.iterator();
while (it.hasNext()) {
    if (matchingStrings.contains(it.next())) {
        it.remove();
    }
}
```

Iterator

Un iterador es una de las muchas formas en que podemos recorrer una colección y, como toda opción, tiene sus pros y sus contras.

Se introdujo por primera vez en Java 1.2 como reemplazo de las enumeraciones y:

- Se han introducido nombres de métodos mejorados
- Es posible eliminar elementos de una colección sobre la que estamos iterando
- No garantiza el orden de iteración

La interfaz del iterador

Para empezar, necesitamos obtener un Iterador de una Colección; Esto se hace llamando al método iterator().

Para simplificar, obtendremos la instancia de Iterator de una lista:

```
List<String> items = ...
Iterator<String> iter = items.iterator();
```

La interfaz del iterador tiene tres métodos principales:

hasNext()

El método hasNext() se puede usar para verificar si queda al menos un elemento para iterar.

Está diseñado para ser utilizado como condición en bucles while:

```
while (iter.hasNext()) {
    // ...
}
```

next()

El método next() se puede usar para pasar por encima del siguiente elemento y obtenerlo:

```
String next = iter.next();
```

Es una buena práctica usar hasNext() antes de intentar llamar a next().

Los iteradores de colecciones no garantizan la iteración en ningún orden determinado a menos que una implementación concreta lo proporcione.

remove()

Finalmente, si queremos eliminar el elemento actual de la colección, podemos usar el método remove:

```
iter.remove();
```

Esta es una manera segura de quitar elementos mientras se itera sobre una colección sin riesgo de una ConcurrentModificationException.

Ejemplo Full Iterator

Ahora podemos combinarlos todos y echar un vistazo a cómo usamos los tres métodos juntos para el filtrado de colecciones:

```
while (iter.hasNext()) {
    String next = iter.next();
    System.out.println(next);

    if( "TWO".equals(next)) {
        iter.remove();
    }
}
```

Así es como comúnmente usamos un Iterator, verificamos con anticipación si hay otro elemento, lo recuperamos y luego realizamos alguna acción sobre él.

Iteración con expresiones lambda

Como vimos en los ejemplos anteriores, es muy detallado usar un Iterator cuando solo queremos repasar todos los elementos y hacer algo con ellos.

Desde Java 8, tenemos el método `forEachRemaining` que permite el uso de lambdas para procesar los elementos restantes:

```
iter.forEachRemaining(System.out::println);
```

La interface ListIterator

ListIterator es una extensión que agrega una nueva funcionalidad para iterar sobre listas:

```
ListIterator<String> listIterator = items.listIterator(items.size());
```

Podemos proporcionar una posición inicial, que en este caso es el final de la lista.

hasPrevious() y previous()

ListIterator se puede usar para el recorrido hacia atrás, por lo que proporciona equivalentes de hasNext() y next():

```
while(listIterator.hasPrevious()) {
```

```
    String previous = listIterator.previous();
}
```

nextIndex() y previousIndex()

Además, podemos recorrer los índices y no los elementos reales:

```
String nextWithIndex = items.get(listIterator.nextIndex());
String previousWithIndex = items.get(listIterator.previousIndex());
```

Esto podría resultar muy útil en caso de que necesitemos conocer los índices de los objetos que estamos modificando actualmente, o si queremos llevar un registro de los elementos eliminados.

add()

El método add, que, como su nombre indica, nos permite añadir un elemento antes del elemento que sería devuelto por next() y después del devuelto por previous():

```
     listIterator.add("FOUR");
```

set()

El último método que vale la pena mencionar es set(), que nos permite reemplazar el elemento que se devolvió en la llamada a next() o previous():

```
String next = listIterator.next();
if( "ONE".equals(next)) {
    listIterator.set("SWAPPED");
}
```

Es importante tener en cuenta que esto solo se puede ejecutar si no se realizaron llamadas previas a `add()` o `remove()`.

Ejemplo completo de ListIterator

Ahora podemos combinarlos todos para hacer un ejemplo completo:

```
ListIterator<String> listIterator = items.listIterator();
while(listIterator.hasNext()) {
    String nextWithIndex = items.get(listIterator.nextIndex());
    String next = listIterator.next();
    if("REPLACE ME".equals(next)) {
        listIterator.set("REPLACED");
    }
}
listIterator.add("NEW");
while(listIterator.hasPrevious()) {
    String previousWithIndex
```

```
     = items.get(listIterator.previousIndex());
    String previous = listIterator.previous();
    System.out.println(previous);
}
```

En este ejemplo, comenzamos obteniendo el ListIterator de la lista, luego podemos obtener el siguiente elemento ya sea por índice (que no aumenta el elemento actual interno del iterador) o llamando a next.

Luego podemos reemplazar un elemento específico con set e insertar uno nuevo con add.

Después de llegar al final de la iteración, podemos retroceder para modificar elementos adicionales o simplemente imprimirlos de abajo hacia arriba.

Bibliografía

1. baeldung. (2022, julio 28). Guide to the Java ArrayList. Baeldung.com. https://www.baeldung.com/java-arraylist
2. baeldung. (2023, abril 24). A Guide to the Java LinkedList. Baeldung.com. https://www.baeldung.com/java-linkedlist
3. baeldung. (2022 junio 23). A Guide to Iterator in Java. Baeldung.com. https//www.baeldung.com/java-iterator
4. Java collections framework. (s/f). Programiz.com. Recuperado el 15 de noviembre de 2023, de https://www.programiz.com/java-programming/collections
5. Java Map interface. (s/f). Programiz.com. Recuperado el 15 de noviembre de 2023, de https://www.programiz.com/java-programming/map
6. Novo, H. L. (2013, enero 16). Guía de colecciones en Java • Héctor Luaces Novo. Héctor Luaces. https://www.luaces-novo.es/guia-de-colecciones-en-java/