

Índice

Índice

Introducción

Vista general

Operaciones comunes

- Métodos

- Bucle FOR EACH

List

- Características

Set

- Acceder a los elementos de un TreeSet

- TreeSet de Objetos no primitivos (Comparable)

- Características

Map

- Características

¿Cual colección usar?

Resumen de métodos

-  Añadir

-  Eliminar

-  Vaciar

-  Verificar contenido

-  Tamaño de la colección

-  Comprobar si está vacía

-  Obtener elementos

-  Iterar elementos

Resumen características

Introducción

Una **colección en Java** es una estructura que permite almacenar, organizar y manipular un grupo de objetos. Algo similar en la vida real a un álbum de fotos, un carrito de supermercado o una biblioteca de libros.



Se utilizan para organizar los datos y acceder a ellos de una forma rápida (ya que están en memoria), pudiendo insertar, actualizar, obtener y eliminar elementos.

Son similares a los arrays, pero estos tienen las siguientes limitaciones:

- Los arrays tienen un tamaño fijo. No pueden cambiar de forma dinámica su tamaño. Ni aumentan, ni disminuyen.
- No proveen distintas formas de almacenar o acceder a los datos.
- No poseen formas automáticas de ordenación.

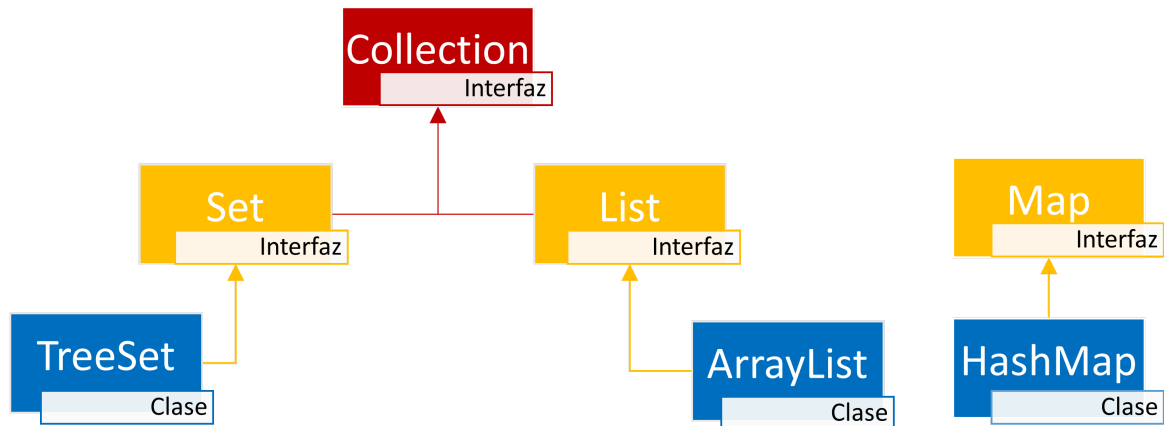
Para superar dichas limitaciones, utilizamos las Colecciones.

Las colecciones nos ofrecen los siguientes beneficios:

- **Menos esfuerzo al manejarlas:** Ya poseen una estructura de datos y algoritmos para agrupar, ordenar, insertar, borrar y buscar sus elementos.
- **Fácil de aprender:** Gastamos menos tiempo para comprender la lógica para agrupar y ordenar una colección, que si lo hiciéramos manualmente de otra forma.
- **Mejoran la calidad del código:** Usamos clases que ya están muy probadas y refinadas en su funcionamiento y eficiencia.
- **Reusabilidad:** Las colecciones se pueden usar en diferentes partes de un programa o en diferentes programas, lo que permite compartir y pasar objetos fácilmente. Por ejemplo, si una parte del programa devuelve una lista de usuarios, otra parte puede usar esa misma lista sin problemas. Esto hace que el código sea más fácil de entender y mantener, ya que no necesitas crear estructuras diferentes cada vez.

Vista general

El esquema de colecciones de Java es muy complejo, y veremos tan solo algunas implementaciones de las interfaces más comúnmente usadas.



En Java, la **interfaz** `Collection` es la base para las colecciones de tipo `Set` y `List`. `Collection` define un conjunto de métodos útiles que las colecciones deben implementar, como agregar, eliminar y verificar elementos.

- `Collection`: Interfaz base de la que heredan:
 - `Set` es implementada por la clase `TreeSet`.
 - `List` es implementada por la clase `ArrayList`.
- `Map` es otra interfaz diferente a `Collection` y es implementada por la clase `HashMap`. La interfaz `Map` no implementa la interfaz `Collection` porque su propósito es un poco diferente. Ahora las veremos todas.

Operaciones comunes

Métodos

Todas las colecciones poseen unas operaciones básicas comunes. Después cada colección posee sus propios métodos particulares. Las operaciones básicas para todas las colecciones son las siguientes:

- Añadir un objeto a la colección. Se realiza con el método `add`.
- Borrar un objeto de la colección. Se realiza con el método `remove`.
- Obtener el tamaño de la colección. Se realiza con el método `size`.
- Comprobar si un objeto se encuentra en la colección. Se realiza con el método `contains`.
- Comprobar si una colección está vacía. Se realiza con el método `isEmpty`.
- Obtener un elemento de la colección. Se realiza con el método `get`.
- Eliminar todos los elementos de la colección. Se realiza con el método `clear`.

Cada colección sobrescribirá sus propios métodos, adaptándolos a sus necesidades, pudiendo existir diferencias entre una colección y otra, **por lo que siempre consultaremos la documentación de los mismos**, para comprobar que hace cada método, sus valores devueltos, posibles excepciones lanzadas, etc.

Bucle FOR EACH

Es un bucle especial que no vimos en el momento de las estructuras repetitivas y ahora es el momento. Está especialmente diseñado para recorrer colecciones y arrays, y con el que no necesitamos manejar ningún índice, tal y como hacemos con el `for` tradicional. Su sintaxis es la siguiente:

```
1  for (TipoDato objeto : nombreColección ) {  
2      instrucciones...  
3  }
```

Veamos un ejemplo concreto para mostrar todos los elementos de un array de Strings.

```
1  String[] nombres = {"Tony", "Bruce", "Peter", "Natasha", "Steve", "Clint"};  
2  for (String nombre : nombres) {  
3      System.out.println(nombre)  
4  }
```

El bucle `for...each` anterior imprimiría en cada vuelta los nombres del array en el mismo orden en el que han sido definidos. No tendríamos que manejar un índice ni incrementarlo para poder acceder a los elementos por su posición, lo cual nos puede ahorrar muchos errores de acceso a índices erróneos.

Si necesitamos recorrer el array o colección de una forma que no sea estrictamente secuencial, por ejemplo, recorrerla al revés o de dos en dos, como necesitamos acceder a posiciones concretas del array, no podríamos usar el `for...each` y necesitaríamos un `for` tradicional y para poder manejar el índice.

List

Son colecciones basadas en índices, como los arrays (de ahí su nombre), que permiten duplicados y se almacenan en base a una posición.

La implementación más común de la Interfaz **List**, es la clase [ArrayList](#). También existe [LinkedList](#) la cual ofrece mejor rendimiento en determinadas circunstancias.

Su sintaxis es la siguiente:

```
1 List<Objeto> nombreColección = new ArrayList<>();
```

Creamos un nuevo objeto del tipo `ArrayList` y debemos parametrizar el tipo de dato que albergará la colección.

Note

Si no parametrizamos la colección con `<>`, estaremos creando una colección de `Object`, lo cual nos permitiría crear una colección de cualquier tipo de objeto todos mezclados. Se recomienda encarecidamente siempre crear una colección de un tipo de objeto determinado.

Veamos un ejemplo para crear un `ArrayList` de Strings.

```
1 List<String> lista = new ArrayList<>();
```

Important

🧐 Se considera una buena práctica para el uso de colecciones, el declarar la variable del tipo de la interfaz (`List`) que es más genérico, e inicializarla con el nombre de la especialización (`ArrayList`). De esta forma, en la misma variable `lista`, podremos guardar un `ArrayList`, o cualquier otro objeto que implemente la interfaz `List`, como un `LinkedList`. **Haremos lo mismo para todas las demás colecciones.**

🧐 Además, en la inicialización, podemos omitir la clase parametrizada, pudiendo dejar `new Colección<>()`, siendo `Colección` la clase que implementa la interfaz, sea `ArrayList`, `TreeSet` o `HashMap`.

Así estamos creando el objeto `lista`, que será un `ArrayList` que contendrá `String`. Inicialmente se crea un `ArrayList` vacío, y deberemos añadirle elementos con el método `add`. Como le hemos indicado que será una colección de `String`, el método `add` sólo admitirá objetos de esa clase.

```

1 // Creamos la lista vacía inicialmente
2 List<String> lista = new ArrayList<>();
3
4 // Añadimos objetos a la lista
5 lista.add("Tony");
6 lista.add("Bruce");
7 lista.add("Clint");
8 lista.add("Steve");
9 lista.add("Natasha");
10 lista.add("Peter");
11 lista.add(2, "Thanos"); // Se inserta en la posición 2, moviendo todos los
    índices siguientes

```

Cada vez que insertamos un elemento con `add(objeto)`, este se posiciona al final de la lista. También podemos usar el método sobrecargado `add(índice, objeto)`, dónde le decimos el índice donde insertar el objeto, y lo que hace es sumarle un 1 a todos los índices siguientes.

Para acceder a los elementos de la lista, usaremos el método `get(índice)`, el cual recibe un entero que representa el índice (posición) del elemento dentro del `ArrayList`. Sus índices empiezan en 0, igual que los arrays.

```

1 System.out.println(lista.get(2)); // Thanos

```

Para borrar los elementos, podemos hacerlo de dos formas, gracias a la sobrecarga del método `remove()`.

```

1 // Borramos por índice
2 lista.remove(2); // Borrará a Thanos
3 // Borramos por objeto
4 lista.remove("Natasha"); // Borrará a la primera "Natasha". Da igual en que
    posición estuviese
5 // Borramos todo
6 lista.clear(); // Borra a todos los elementos

```

La colección `List`, posee todos los métodos comunes que hemos descrito con anterioridad, como `contains`, `isEmpty`, `clear`, `remove`, etc. Lo mejor siempre para comprobar los métodos de una colección será crear el objeto y escribir su nombre y un punto para que el IDE nos muestre la lista de todos los métodos disponibles, así como la documentación de como usarlos.

Características

- Son de propósito general y muy versátiles en todos los aspectos.
- Están basadas en índices.
- Admiten duplicados.
- No tienen un orden natural por defecto.
- Acceso directo por posición.
- Son muy parecidos a los arrays, y por eso familiares para todos los programadores.



Tip

Encontrarás un [resumen de todos los métodos comunes](#) de las colecciones al final del documento.

Set

La colección Set tiene como principal característica que no admite duplicados y que algunas de sus implementaciones mantienen sus elementos ordenados. Su implementación más común es la clase [TreeSet](#). También existe la clase [HashSet](#), que es más rápida que la clase TreeSet pero no es muy versátil.

Su sintaxis es la siguiente:

```
1 Set<Objeto> nombreColección = new TreeSet<>();
```

Veamos el ejemplo para crear un `TreeSet` de enteros.

```
1 Set<Integer> numeros = new TreeSet<>();
2 numeros.add(8);
3 numeros.add(12);
4 numeros.add(12); // No lo añade. add devuelve false.
5 numeros.add(-4);
6 int elementos = numeros.size(); // elementos = 3
7 System.out.println(numeros); // [-4, 8, 12]
```

Note

Al parametrizar una colección siempre deberemos indicar una clase, por lo que si queremos hacer una colección de tipos primitivos, deberemos usar sus clases envoltorio.

Important

La mayoría de las colecciones tienen sobrescrito el método `toString()`, de forma que podemos imprimir directamente el objeto y veríamos el contenido de las mismas de una forma rápida sin tener que recorrerlas manualmente.

La gran ventaja que presentan los `TreeSet` frente a los `ArrayList`, es que **no permiten duplicados**, por lo que la colección `numeros`, no tendría 2 veces el valor `12`. Y otra ventaja es que **mantiene el orden natural de los elementos**. Es decir, sus elementos estarán ordenados de menor a mayor si son números y alfabéticamente si son String. Da igual el orden de inserción.

Otro ejemplo con un `TreeSet` de Strings:

```
1 Set<String> frutas = new TreeSet<>();
2 frutas.add("Plátano");
3 frutas.add("Calabaza");
4 frutas.add("Zanahoria");
5 frutas.add("Arándano");
6 frutas.add("Plátano"); // Elemento repetido, no lo añade
7 System.out.println(frutas); // [Arándano, Calabaza, Plátano, Zanahoria]
```

Acceder a los elementos de un TreeSet

No están basando en índices como los `List` y no podemos acceder directamente a un elemento concreto de un `TreeSet`. Tendremos que recorrer el conjunto de forma secuencial.

- ★ Usando un bucle `for-each`:

```
1 Set<Integer> numeros = new TreeSet<>();
2 numeros.add(1);
3 numeros.add(2);
4 numeros.add(3);
5
6 for (Integer num : numeros) {
7     System.out.println(num); // Accede a los elementos uno a uno
8 }
```

Esta forma es más simple que la siguiente, ya que usamos un bucle que “se mueve solo” y no necesita ni índices ni objetos ni métodos adicionales (como el siguiente). Lo usaremos cuando tengamos que recorrer completamente el `Set`.

⚠ **Warning**

No podrás eliminar elementos de una colección (de cualquier tipo) mientras la recorres con un `for-each`. Si intentas hacerlo directamente, obtendrás una excepción de tipo `ConcurrentModificationException`.

- Usando un `iterator`:

```
1 Iterator<Integer> iterador = numeros.iterator();
2 while (iterador.hasNext()) {
3     System.out.println(iterador.next()); // Accede a los elementos usando
    el iterador
4 }
```

El objeto `iterator` es como un puntero que está posicionado antes del primer elemento.

- El método `next()` del `Iterator` **devuelve el siguiente elemento** de la colección y **avanza el puntero** del iterador al elemento siguiente. Si lo llamas repetidamente, irás obteniendo los elementos uno por uno hasta que llegues al final de la colección.
- El método `hasNext()` del `Iterator` comprueba si **hay más elementos** en la colección que se está recorriendo. **Devuelve true si hay un siguiente elemento** disponible y `false` si no hay más. Es como preguntar: “¿Queda algo más por recorrer?”. Se usa comúnmente como condición en un `while` para asegurarse de que no intentamos acceder a elementos cuando ya hemos llegado al final de la colección.

💡 **Tip**

Con un iterador si puedes eliminar elementos de la colección mientras lo recorres, usando el método `iterator.remove()`.

TreeSet de Objetos no primitivos (Comparable)

Expongamos un problema. Tengo la siguiente clase:

```

1 public class Persona {
2     private String nombre;
3     private int edad;
4     ...
5 }

```

Y ahora hago una colección TreeSet de `Persona`. Hasta aquí todo normal. Es lo mismo que antes pero en lugar de primitivos usamos una clase nuestra.

```

1 Set<Persona> lista = new TreeSet<>();
2 lista.add(new Persona("Tobey", 46));
3 lista.add(new Persona("Tom", 25));
4 lista.add(new Persona("Andrew", 38));

```

Hemos dicho que la colección TreeSet mantiene sus elementos ordenados, pero nos asaltan varias preguntas.

- ¿Como sabe Java si una persona es igual a otra para añadirla o no a la lista?
- ¿Qué criterios usará para ordenarla?
- ¿Ordena alfabéticamente por nombre? ¿Ordena numéricamente por edad?
- ¿Qué Spiderman te gusta más?

La respuesta a la primera pregunta es que **NO SABE HACERLO**, por lo que dará un fallo y no hará nada de lo siguiente.

Para poder hacer una colección TreeSet, el objeto deberá implementar la interfaz

`Comparable`. Al implementarla, nos obligará a desarrollar un método llamado `compareTo()`, el cual decidirá si un objeto de su misma clase es **menor, igual o mayor** que otro recibido. Nosotros deberemos escribir la lógica de la comparación eligiendo el criterio que queramos. Por ejemplo, podremos decidir si *Andrew* es menor que *Tom* porque alfabéticamente va antes, o si *Tom* debería ir el primero en la lista, por ser el ~~mejor~~ más joven.

El método `compareTo` recibirá una objeto de la misma clase (`Persona` en nuestro ejemplo) y retornará un entero. Ese entero deberá seguir los siguientes criterios:

- **Un número negativo:** significa que el objeto actual (`this`) es "menor" que el objeto comparado (recibido en el método).
- **Cero (0):** significa que ambos objetos son "iguales" en cuanto al criterio de orden.
- **Un número positivo:** significa que el objeto actual (`this`) es "mayor" que el objeto comparado (recibido en el método).

Solo tenemos que sobrescribir el método. Las llamadas y las comparaciones las hará de forma interna el `TreeSet` al hacer una inserción con `add`.

Supongamos que lo queremos es ordenar las personas por edad. Primero el más joven. Pues compararemos los atributos edad de los objetos en el `compareTo`.

Ya terminado sería:

```

1 public class Persona implements Comparable<Persona> {
2     private String nombre;
3     private int edad;
4     ...
5     @Override
6     public int compareTo(Persona otraPersona) {
7         return this.edad - otraPersona.getEdad();
8     }
9 }
10

```

Al restar las edades, ya tenemos justo los valores que queremos.

- Si ESTA clase es menor que la recibida por parámetros, nos dará un número negativo (da igual cual).
- Si ESTA clase es mayor que la recibida por parámetros, dará un número positivo.
- Y si tienen las mismas edades, el resultado será 0.

De esta forma, el `TreeSet` podrá comparar los objetos entre si y saber el orden (que hemos decidido nosotros) para insertar los elementos en unas posiciones u otras, o bien no insertarlos porque son iguales.

⚠ Warning

`TreeSet` para comprobar la igualdad de dos objetos no usa el `.equals()` ni el `hashCode()`, si no el `compareTo()`. Por lo que hay que tener en cuenta que dos personas distintas, con distintos nombres PERO CON LA MISMA EDAD, el `TreeSet` **entendería que son iguales, por lo que no la insertaría** en la colección.

Todavía nos queda una duda... ¿Por qué en las colecciones que hicimos de ejemplo inicialmente con `Integer` o `String` no tuvimos que hacer nada? La respuesta es simple. Es que esas clases ya tienen la interfaz `Comparable` implementada, lo que lleva a que tienen el método `compareTo` sobrescrito y así el `TreeSet` si sabe como compararlos. Ese trabajo que nos ahorramos.

Características

- No admiten duplicados
- Están ordenados. Deben implementar la interfaz `Comparable`.
- No permiten acceso aleatorio. Sólo secuencial.
- No permiten acceso por índice.

Map

Las colecciones que implementan la interfaz `Map`, están basadas en pares de `clave-valor`. Cada par de clave-valor se llama **entrada**.

- La *clave* es un objeto que identifica a una entrada.
- El *valor* es un objeto que contiene el valor asociado a la clave.

Podríamos decir que en un `ArrayList<Persona>`, el valor sería el objeto `Persona`, y la clave sería el índice que es un entero SIEMPRE. La ventaja de los `Map` es que en lugar de un entero para el índice, podemos usar otro objeto cualquiera para poder acceder a la `Persona`, pudiendo asociar como clave un `char`, un `String` u otro objeto más complejo.

La implementación más común de la interfaz `Map` es `HashMap`.

Su sintaxis es la siguiente:

```
1 Map<Clave, Valor> nombreColección = new HashMap<>();
```

Donde `Clave` y `Valor` son las clases que representarán a la clave y al valor dentro del `Map`. Veamos un ejemplo de código:


Vamos a crear una colección con los personajes de [Reservoir Dogs](#).

Creemos primero la clase `Actor`:

```
1 public class Actor {
2     private String nombre;
3     private String personaje;
4     ...
5 }
```

```
1 Map<String, Actor> reparto = new HashMap<>();
2 reparto.put("BLANCO", new Actor("Harvey Keitel", "Larry"));
3 reparto.put("NARANJA", new Actor("Tim Roth", "Freddy"));
4 reparto.put("ROSA", new Actor("Steve Buscemi", "Michael"));
5 reparto.put("RUBIO", new Actor("Michael Madsen", "Vic Vega"));
6 reparto.put("AZUL", new Actor("Edward Bunker", "Jack"));
7 reparto.put("MARRON", new Actor("Quentin Tarantino", "Tommy"));
8 reparto.put("MARRON", new Actor("Tarantinooor", "Tommy")); // Se inserta,
    sustituyendo al anterior.
```

Important

 Para añadir elementos a un `HashMap`, utilizamos el método `put(clave, valor)`, que recibe dos objetos: uno como clave y otro como valor. Este método devuelve `null` si la clave es nueva, o bien el valor anterior asociado a esa clave si ya existía (y será reemplazado por el nuevo valor). Debido a esto, nunca habrá claves duplicadas en el mapa.

Otros ejemplos de métodos útiles serían:

```

1 // Acceder a elementos por su clave
2 reparto.get("ROSA"); // retorna -> Actor{nombre=Steve Buscemi,
   personaje=Michael}
3 // Borrar elementos
4 reparto.remove("RUBIO"); // retorna -> Actor{nombre=Michael Madsen,
   personaje=Vic Vega}
5 // Buscar elementos
6 reparto.containsKey("VERDE"); // retorna -> false

```

Internamente los HashMap utilizan dos tablas, una para las claves y otra para los valores. Se puede acceder a ambas tablas a través de los métodos `keySet()` y `values()`, respectivamente. Y podemos recorrer la colección iterando sobre las claves o los valores, según nos convenga.

Ejemplo de recorrer un `HashMap` con un bucle `for each`:

```

1 // Iterando sobre sus claves
2 for (String clave : reparto.keySet()) {
3     Actor actor = reparto.get(clave);
4     System.out.println(actor.getPersonaje() + " (interpretado por " +
   actor.getNombre() + ")");
5 }

```

```

1 // Iterando sobre sus valores
2 for (Actor actor : reparto.values()) {
3     System.out.println(actor.getPersonaje() + " (interpretado por " +
   actor.getNombre() + ")");
4 }

```

Ya es decisión de usar la forma que mejor se adapte a la lógica de nuestra aplicación. Por ejemplo, si sólo queremos mostrar los actores, en nuestro ejemplo sería más simple iterar sobre sus valores, ya que no usamos sus claves, pero si hubiésemos querido mostrar su clave (para mostrar el apodo que tenían en la película), en el segundo `for` no tenemos la clave por ningún sitio. Es importante conocer las distintas formas y después elegir.

Características

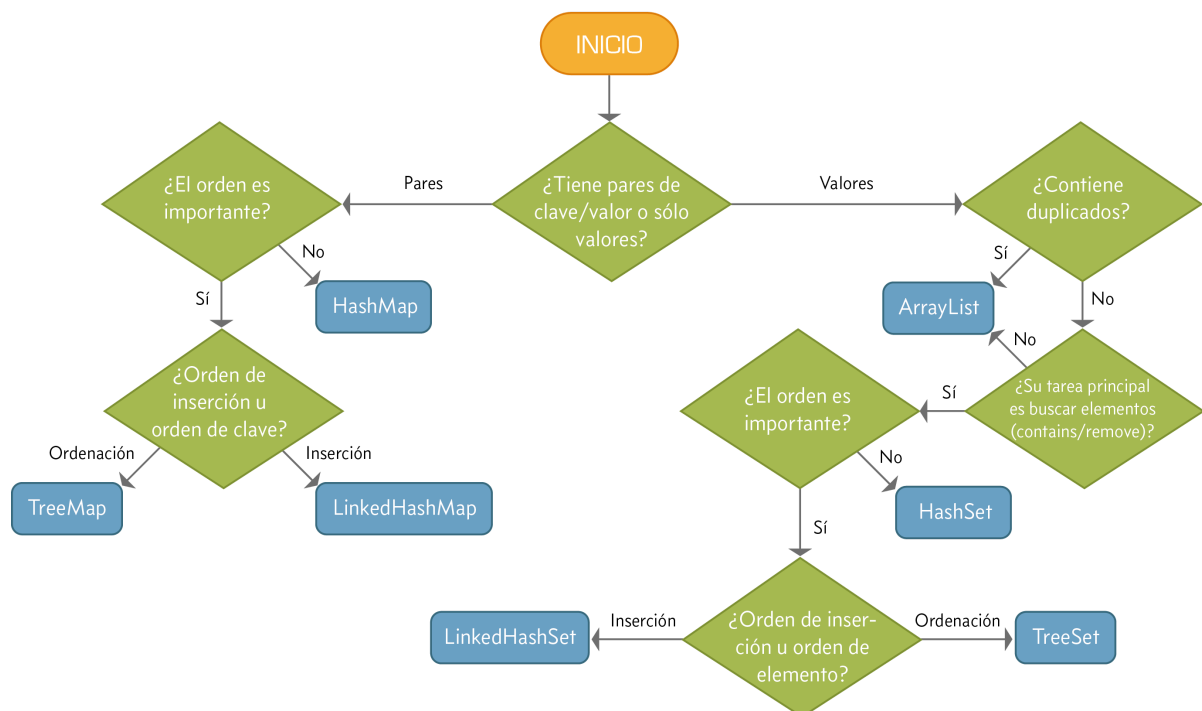
- No admiten claves duplicadas. Valores duplicados si.
- Permiten búsqueda rápida por clave.
- Flexibilidad al tener claves y valores de distintos tipos.

¿Cual colección usar?

Entre tantas colecciones, es posible que a veces no tengamos claro cual usar que se adapte mejor a las necesidades, tanto en funcionamiento como en eficiencia. El siguiente cuadro te ayudará a escoger la más adecuada.

Algunas clases o interfaces no se han visto en el curso. Pero si se ajustan a lo que necesitas, y conociendo las bases de las colecciones, verás que no es nada difícil aprender a manejar una nueva, ya que seguro que comparten muchos métodos y comportamientos con otra que si conoces.

Diagrama de decisión para uso de colecciones Java



Resumen de métodos

Un resumen de los métodos más comunes en las colecciones de Java, cada uno con su ejemplo.

Important

Se recomienda encarecidamente leer la documentación de cada método, para conocer exactamente como se comporta cada uno (qué valores recibe, devuelve, posibles excepciones, etc.). En este resumen no están contemplados todos los casos que se pueden dar. Por ejemplo, ¿qué pasa si intentamos borrar el elemento 4 de una lista de 2 elementos?. Aquí no se explica, en la documentación de `ArrayList.remove()` si está explicado.

+ Añadir

- `add(elemento)`: Añade el elemento a la lista. En algunos casos puede retornar `true` si lo insertó o `false` en caso contrario.

```
1 List<String> amigos = new ArrayList<>();
2 amigos.add("Juan"); // Agrega "Juan" a la lista de amigos.
```

✗ Eliminar

- `remove(elemento)`: Elimina el primer objeto que coincida con el indicado de la lista. Da igual qué posición ocupe.

```
1 amigos.remove("Juan"); // Elimina "Juan" de la lista de amigos.
```

- `remove(indice)`: Elimina el objeto con la posición indicada. Da igual qué objeto sea.

```
1 amigos.remove(0); // Elimina el primer amigo de la lista.
```

Vaciar

- `clear()`: Borra todos los elementos de la lista y la deja vacía.

```
1 amigos.clear(); // Elimina todos los amigos de la lista.
```

Verificar contenido

- `contains(elemento)`: Devuelve `true` si la lista contiene el elemento indicado, o `false` en caso contrario

```
1 boolean tieneJuan = amigos.contains("Juan"); // Verifica si "Juan" está en la lista.
```

Tamaño de la colección

- `size()`: Devuelve un entero con el número de elementos de la lista.


```
1 | int cantidadAmigos = amigos.size(); // Devuelve el número de amigos en la lista.
```

⚠ Comprobar si está vacía

- `isEmpty()`: Devuelve `true` si está vacía (tiene 0 elementos) o `false` en caso contrario.

```
1 | boolean listaVacía = amigos.isEmpty(); // Verifica si la lista de amigos está vacía.
```

📁 Obtener elementos

- `get(index)`: Devuelve el objeto en la posición indicada.

```
1 | String primerAmigo = amigos.get(0); // Obtiene el primer amigo de la lista.
```

🔄 Iterar elementos

- `iterator()`: Devuelve un objeto `iterator` para poder movernos por la lista con sus métodos.

```
1 | Iterator<String> it = amigos.iterator();
2 | while (it.hasNext()) {
3 |     System.out.println(it.next()); // Imprime cada amigo en la lista.
4 | }
```

Resumen características

Esta tabla te ayudará a comparar las principales diferencias entre estos tres tipos de colecciones y sus implementaciones más comunes.

Característica	List	Set	Map
Implementación común	<code>ArrayList</code>	<code>TreeSet</code>	<code>HashMap</code>
Permite duplicados	Sí	No	No (claves no, valores si)
Están ordenados	Sin orden natural	Según el orden natural de sus elementos	No
Método de acceso	Por índice	Secuencial	Pares clave-valor
Caso de uso común	Lista de elementos accesibles por índice	Conjunto único de elementos secuenciales ordenados	Asociación de claves y valores