

UNIVERSIDAD NACIONAL DE INGENIERÍA

Programación Orientada a Objetos

Colecciones

`List`, `Set`, `Map`, `Comparable`, `Comparator` e iteradores

Gerson Garrido

Semana 11

Hoy

1. El Collections Framework: interfaces e implementaciones
2. `List`, `Set` y `Map`, y lo que cuesta cada operación
3. Colecciones inmutables y vistas para no romper el encapsulamiento
4. Ordenar: `Comparable` y `Comparator`
5. Recorrer: iteradores y una colección propia que funciona en un for-each

> **Lectura**
> Eck (v. 9): subsecs. 10.1.4–10.1.6, secs. 10.2 y 10.3, subsecs. 10.4.1–10.4.3 y 5.8.2.

SECCIÓN 01

Sesión A: List, Set, Map

01

La semana pasada, esta semana

Semana 10: genéricos

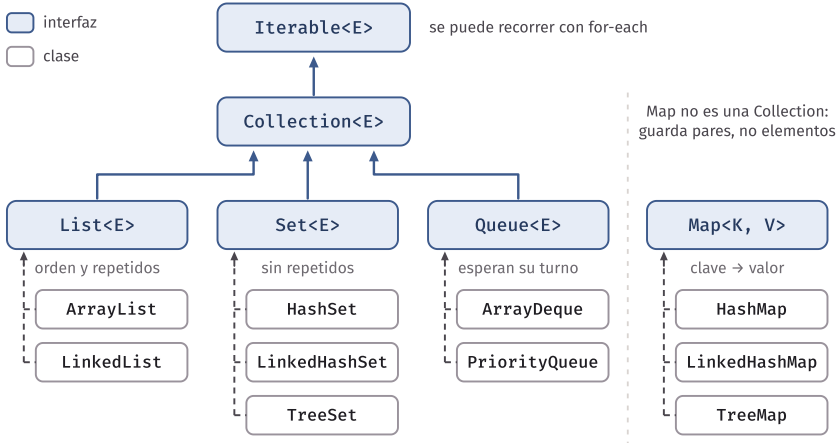
`Box<T>`, `BoundedStack<E>`: el compilador revisa qué entra y qué sale de una colección.

Semana 11: el Collections Framework

Interfaces y clases genéricas ya escritas y probadas para guardar muchos objetos: en orden, sin repetidos, por clave.

La mitad del trabajo es elegir bien la colección.

La jerarquía



Programar contra la interfaz

Ejemplos

```
1  List<Song> songs = new ArrayList<>();
2  Map<String, Integer> counts =
3          new TreeMap<>();
4
5  double average(List<Integer> marks) {
6      // sirve con cualquier List
7  }
```

- La variable es de tipo interfaz; la clase concreta solo aparece en el `new`.
- Cambiar de implementación es cambiar una línea.
- En parámetros y retornos: pedir `List`, no `ArrayList`.
- Es la idea de la semana 7, ahora en la biblioteca estándar.

Las canciones de los ejemplos

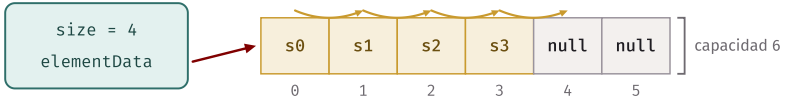
Song.java

```java
public record Song(String title, String artist, int seconds) implements
    Comparable<Song> {
    // compareTo: más adelante
}
```

- Un `record`: `equals` y `hashCode` ya comparan los tres campos (semana 5).
- `seconds`: la duración en segundos. `Songs.sample()`: seis canciones inventadas, tres artistas.
- Eso va a importar mucho en `Set` y `Map`.

List: ArrayList frente a LinkedList

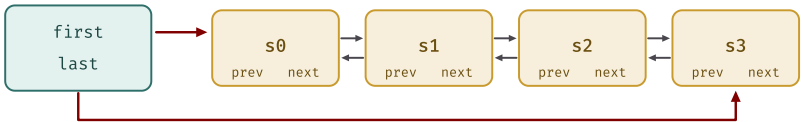
ARRAYLIST · UN ARREGLO DE REFERENCIAS



get(2): se calcula la posición y se va directo a la casilla 2

add(0, x): corre los 4 elementos una casilla (arcos dorados)

LINKEDLIST · NODOS ENCADENADOS



add(0, x): un nodo nuevo y dos flechas. get(2): recorrer s0 → s1 → s2

Lo que cuesta cada operación

Operación	`ArrayList`	`LinkedList`
`get(i)`, `set(i, x)`	constante	proporcional a n
`add(x)` al final	constante (en promedio)	constante
`add(0, x)`, `remove(0)`	proporcional a n	constante
`contains(x)`	proporcional a n	proporcional a n

- En la práctica, `ArrayList` gana casi siempre: memoria contigua, nodos dispersos.
- Para una cola: `ArrayDeque`, no `LinkedList`.
- ¿Mucho `contains`? Entonces no es una lista: es un conjunto.

`Set`: sin repetidos

SetDemo.java · visits: 7 ciudades

```java
Set<String> hash = new HashSet<>();
Set<String> tree = new TreeSet<>();
for (String city : visits) {
    hash.add(city);
    tree.add(city);
}
```

- 7 visitas, 5 ciudades: los repetidos no entran; `add` devuelve `false`.
- Iguales: en `HashSet` y `LinkedHashSet`, según `equals` y `hashCode` (semana 5).
- En `TreeSet`, si la comparación da 0 (más adelante).

bash · con LinkedHashSet también

```
HashSet:       [Puno, Lima, Ica, Arequipa, Cusco]
LinkedHashSet: [Lima, Cusco, Arequipa, Puno, Ica]
TreeSet:       [Arequipa, Cusco, Ica, Lima, Puno]
```

Tres conjuntos, tres órdenes

Clase	Orden del recorrido	`add`, `contains`
`HashSet`	no especificado	constante en promedio
`LinkedHashSet`	orden de inserción	constante en promedio
`TreeSet`	ordenado (necesita comparar)	proporcional a $\log n$

No dependan del orden de un `HashSet`

Depende de los `hashCode` y del tamaño de la tabla, y puede cambiar al agregar elementos.

`Map<K, V>`: de una clave a un valor

Método	Qué hace
`put(k, v)`	asocia; devuelve el valor anterior
`get(k)`	el valor, o `null`
`getOrDefault(k, d)`	el valor, o `d`
`containsKey(k)`	¿tiene valor?
`remove(k)`	quita la clave
`keySet()`, `values()`	vistas
`entrySet()`	los pares

- Cada clave, una vez, con un valor.
- `HashMap`: rápido, sin orden.
- `LinkedHashMap`: orden de inserción.
- `TreeMap`: claves ordenadas.
- `HashSet` está hecho con un `HashMap` por dentro.

Contar palabras

WordCount.java

```java
Map<String, Integer> counts = new TreeMap<>();
for (String word : text.toLowerCase().split("[^\\p{L}]+")) {
    if (!word.isEmpty()) {
        counts.put(word, counts.getOrDefault(word, 0) + 1);
    }
}
```

bash

```
{cae=2, en=2, garúa=3, la=3, lima=2, moja=1, no=1, pero=1}
garúa: 3, sol: null, sol con getOrDefault: 0
```

Ojo: `int x = counts.get("sol")` lanza NullPointerException.

Recorrer un mapa: `entrySet()`

WordDemo.java

```java
for (Map.Entry<String, Integer> e
        : counts.entrySet()) {
    if (e.getValue() >= 2) {
        IO.println("  " + e.getKey()
                + " x" + e.getValue());
    }
}
```

bash

```
cae x2
en x2
garúa x3
la x3
lima x2
```

- Cada par es un `Map.Entry<K, V>` con `getKey()` y `getValue()`.
- Mejor que recorrer `keySet()` y hacer `get` por cada clave.

Un índice: `Map<String, Set<Integer>`

WordIndex.java · index es un TreeMap

```java
for (int i = 0; i < lines.size(); i++) {
    for (String w : words(lines.get(i))) {
        Set<Integer> where = index.get(w);
        if (where == null) {
            where = new TreeSet<>();
            index.put(w, where);
        }
        where.add(i + 1);
    }
}
```

- Buscar; si no está, crear y guardar; después usar.
- `where.add` cambia el conjunto que ya está en el mapa.
- Semana 12: `computeIfAbsent`, en una línea.

bash

```
río -> [1, 2]
puente -> [1, 3]
el -> [1, 2]
```

Cómo encuentra un `HashMap` una clave

1. Un arreglo de buckets: casillas donde se guardan los pares.
2. `put(k, v)`: mezcla `k.hashCode()` con sus bits altos y lo reduce a un índice (con 16 buckets, los últimos 4 bits del resultado). El par va a ese bucket.
3. `get(k)`: calcula el mismo bucket y busca ahí una clave igual según `equals`.
4. No recorre todo el mapa: va directo a un bucket. Por eso es rápido.

Pregunta

¿Qué pasa si la clave cambia *después* del `put`?

Error común: una clave mutable

MutableKeyDemo.java

```java
Map<MutableKey, String> seats =
        new HashMap<>();
MutableKey seat = new MutableKey('A', 7);
seats.put(seat, "alice");
seats.put(new MutableKey('B', 3), "bob");

seat.moveTo('C', 2);   // ¡la clave cambia!
seats.get(new MutableKey('C', 2));
seats.get(new MutableKey('A', 7));
seats.containsKey(seat);
```

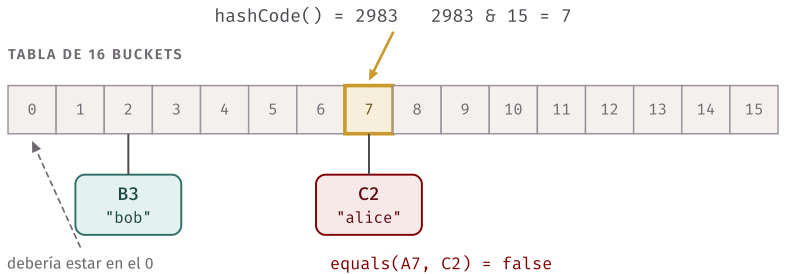
bash

```
get(A7) = alice
mapa = {B3=bob, C2=alice}
get(C2) = null
get(A7) = null
containsKey(seat) = false
size = 2
```

El mapa la imprime, pero nadie la encuentra.

La entrada se queda en el bucket viejo

```
seats.get(new MutableKey('A', 7))   // null
```



5. A7 busca en el 7: hay una entrada, pero C2 no es igual a A7

la entrada sigue en el mapa (size = 2), pero ya no se puede encontrar

Colecciones inmutables

Ejemplos

```
1  List<String> notes =
2          List.of("do", "re", "mi");
3  Set<Integer> primes = Set.of(2, 3, 5, 7);
4  Map<String, Integer> stock =
5          Map.of("pan", 40, "leche", 12);
```

jshell

```
jshell> nums.set(0, 9)
|  Exception java.lang.
     UnsupportedOperationException
```

- `add`, `set`, `put`, `remove`: lanzan excepción.
- Sin `null`; `Set.of` y `Map.of` rechazan repetidos.
- Ideales para constantes y datos de prueba.
- Y como claves: lo inmutable no se pierde en un `HashMap`.

Devolver una colección sin romper el encapsulamiento

Playlist.java

```java
public class Playlist
        implements Iterable<Song> {
    private final List<Song> songs =
            new ArrayList<>();

    // add(song): rechaza repetidos

    public List<Song> songs() {
        return Collections
            .unmodifiableList(songs);
    }
}
```

- `return songs;` deja hacer `add` o `clear` sobre la lista interna.
- `unmodifiableList`: una vista. No copia; deja leer; no deja modificar.
- Ve los cambios posteriores. Para una foto fija: `List.copyOf`.

Vista frente a copia

ViewDemo.java

```java
List<Song> view = playlist.songs();
List<Song> copy = List.copyOf(view);
view.add(new Song("Arena", "Duna", 185));  // UnsupportedOperationException
playlist.add(new Song("Brisa", "Duna", 210));
IO.println("vista: " + view.size() + " canciones, copia: " + copy.size());
```

bash

```
vista: 2 canciones, copia: 1
```

La vista protege la lista, no los elementos: si son mutables, quien recibe la vista puede cambiar cada uno. Con un record inmutable, no.

Elegir la colección

Necesito...	Interfaz	Clase
una secuencia con posiciones, con repetidos	`List`	`ArrayList`
saber rápido si algo está, sin repetidos	`Set`	`HashSet`
lo anterior, en orden de llegada	`Set`	`LinkedHashSet`
lo anterior, ordenado	`Set`	`TreeSet`
buscar un valor por su clave	`Map`	`HashMap`
lo anterior, con las claves ordenadas	`Map`	`TreeMap`
sacar en el orden en que llegaron	`Queue`	`ArrayDeque`

La pregunta no es “¿cuál me gusta?” sino “¿qué operaciones se repiten?”

SECCIÓN 02

Sesión B: Comparable, Comparator, iteradores

02



Orden natural: `Comparable<T>`

`int compareTo(T other)`: se lee el signo. Negativo, `this` va antes; cero, empatan; positivo, va después. `String`: alfabético; `Integer`: numérico.

Song.java · por título, luego artista, luego duración

```java
public int compareTo(Song other) {
    int byTitle = title.compareTo(other.title);
    if (byTitle != 0) {
        return byTitle;
    }
    int byArtist = artist.compareTo(other.artist);
    if (byArtist != 0) {
        return byArtist;
    }
    return Integer.compare(seconds, other.seconds);
}
```

El contrato de `compareTo`

1. **Antisimetría**: `a.compareTo(b)` y `b.compareTo(a)` tienen signos contrarios.
2. **Transitividad**: si *a* va antes que *b* y *b* antes que *c*, *a* va antes que *c*.
3. **Empates coherentes**: si *a* y *b* empatan, se comparan igual con cualquier *c*.

> **Recomendado: consistente con `equals`**
>
> `a.compareTo(b) == 0` exactamente cuando `a.equals(b)`. `Song` lo cumple: compara los mismos tres campos que su `equals`.

Error común: comparar restando

SubtractionOrder.java · MAL

```java
public int compare(Integer a,
                   Integer b) {
    return a - b;
}
```

OverflowDemo.java

```java
int a = -2_000_000_000;
int b = 2_000_000_000;
amounts.addAll(List.of(b, a, 0));
amounts.sort(new SubtractionOrder());
```

bash

```
a - b = 294967296
con resta:
  [2000000000, -2000000000, 0]
con Integer.compare:
  [-2000000000, 0, 2000000000]
```

$-4\,000\,000\,000$ no cabe en un `int`: hay overflow y el signo se invierte. Usen `Integer.compare`.

Otros órdenes: `Comparator<T>`

ByDuration.java

```java
public class ByDuration
        implements Comparator<Song> {
    @Override
    public int compare(Song a, Song b) {
        return Integer.compare(a.seconds(),
                               b.seconds());
    }
}
```

- `int compare(T a, T b)`: el mismo signo que `compareTo`.
- Un orden externo: otro objeto, aparte de la clase que se ordena.
- Tantos como hagan falta, sin tocar `Song`.
- Hoy, como clases. Semana 12: con lambdas.

Varios criterios

ByArtistThenTitle.java

```java
public int compare(Song a, Song b) {
    int byArtist =
            a.artist().compareTo(b.artist());
    if (byArtist != 0) {
        return byArtist;
    }
    return a.title().compareTo(b.title());
}
```

- Comparar por el primero y, solo si empata, por el siguiente.
- El mismo esquema que el `compareTo` de `Song`.
- Las pruebas de un comparador deben incluir empates: ahí se esconden los errores.

Pasar el comparador a quien ordena

SortDemo.java

```java
List<Song> songs = Songs.sample();
Collections.sort(songs);  // compareTo
songs.sort(new ByDuration());
songs.sort(new ByArtistThenTitle());
```

bash

```
Por duración (ByDuration):
  185  Duna - Arena
  190  Los Andinos - Cumbre
  190  Costa Verde - Malecón
  210  Duna - Brisa
  ...
```

También: `new TreeSet<>(cmp)`, `new TreeMap<>(cmp)`, `Collections.max(c, cmp)`.

Cumbre antes que *Malecón*: empatan en 190 y `List.sort` es estable (no reordena los que empatan; venían por título).

`Comparable` o `Comparator`

`Comparable<T>`

El orden de la clase. Uno solo, escrito dentro de ella: `compareTo`.

`Comparator<T>`

Un orden externo. Muchos, en objetos aparte; sirven también para clases que no son nuestras.

Un adelanto de la semana 14

Un comparador es un algoritmo (cómo comparar) encapsulado en un objeto que se le pasa a otro (el que ordena): el patrón **Strategy**.

Empates en un `TreeSet`

SortDemo.java

```java
TreeSet<Song> byDuration = new TreeSet<>(new ByDuration());
byDuration.addAll(songs);
IO.println("Lista: " + songs.size() + " canciones; TreeSet(ByDuration): "
        + byDuration.size());
```

bash

```
Lista: 6 canciones; TreeSet(ByDuration): 5
```

- Para un `TreeSet`, “iguales” = el comparador da 0. No usa `equals`.
- *Malecón* y *Cumbre* son distintas, pero duran lo mismo: entra solo la primera.
- Orden inconsistente con `equals`. Solución: desempatar con más criterios.

Qué hace realmente un for-each

Lo que escribimos

```java
for (Song s : songs) {
    // cuerpo
}
```

Lo que compila

```java
Iterator<Song> it = songs.iterator();
while (it.hasNext()) {
    Song s = it.next();
    // cuerpo
}
```

- `Iterable<E>`: un método, `iterator()`, que fabrica un iterador nuevo.
- `Iterator<E>`: `hasNext()`, `next()` y `remove()` (opcional).
- Un cursor entre dos elementos: `next()` salta uno y lo devuelve.
- Funciona con cualquier `Iterable`, también con las clases que escribamos.

Error común: borrar dentro de un for-each

RemoveDemo.java

```java
for (Song s : songs) {
    if (s.seconds() < 200) {
        songs.remove(s);   // ¡por fuera!
    }
}
```

bash

```
for-each + remove:
  java.util.
    ConcurrentModificationException
```

- El for-each tiene un iterador escondido.
- `songs.remove(s)` cambia la lista sin que el iterador se entere.
- El siguiente `next()` lo detecta y falla de inmediato.
- No hace falta ningún hilo: el nombre confunde.

Borrar bien: a través del iterador

RemoveDemo.java

```java
Iterator<Song> it = songs.iterator();
while (it.hasNext()) {
    Song s = it.next();
    if (s.seconds() < 200) {
        it.remove();
    }
}
```

bash

```
Quedan 3:
  220  Garúa
  245  Puente de piedra
  210  Brisa
```

`it.remove()` borra el último elemento que devolvió `next()`. Semana 12: `songs.removeIf(...)`.

El cursor no se pierde

```
Iterator<Song> it = songs.iterator();
while (it.hasNext()) {
    Song s = it.next();
    if (s.seconds() < 200) { it.remove(); }
}
```

it.remove() borra el último elemento que devolvió next()

el cursor está entre dos elementos



hasNext() es false: quedan 3 canciones, sin excepción

con for-each y songs.remove(s) la lista cambia por fuera del iterador: ConcurrentModificationException

Una colección propia en un for-each

Delegar, si ya hay una colección adentro:

Playlist.java

```java
@Override
public Iterator<Song> iterator() {
    return songs().iterator();
}
```

El iterador de la vista: su `remove()` no deja saltarse el invariante.

Escribir el iterador, si los datos se guardan de otra forma:

- `NumberRange(1, 10, 2)`: 1, 3, 5, 7, 9.
- No guarda los números: el iterador los calcula.

`NumberRange`: una clase interna

NumberRange.java

```
1  private class RangeIterator
2          implements Iterator<Integer> {
3      private int next = from;
4      public boolean hasNext() { return next < to; }
5      public Integer next() {
6          if (!hasNext()) {
7              throw new NoSuchElementException();
8          }
9          int current = next;
10         next += step; return current;
11     }
12 }
```

- Clase interna, sin `static`: ve `from`, `to` y `step`.
- `private`: afuera solo se ve un `Iterator<Integer>`.
- Sin elementos: una excepción, nunca `null`.

Cada iterador, su propio cursor

RangeDemo.java

```
1  NumberRange odds =
2          new NumberRange(1, 10, 2);
3  int pairs = 0;
4  for (int a : odds) {
5      for (int b : odds) {
6          if (a < b) {
7              pairs++;
8          }
9      }
10 }
```

bash

```
$ java -cp build RangeDemo
for-each: 1 3 5 7 9
iterador: 1 3 5 7 9
pares a < b: 10
```

- Cada `iterator()` crea un cursor nuevo: dos recorridos a la vez.
- Con un solo cursor compartido, el bucle interno lo agotaría.

SECCIÓN 03

Laboratorio

03

Laboratorio: elegir la colección adecuada

1. Cinco problemas cortos (`lab/Choices.java`): elegir interfaz y clase, justificar con el costo, implementar y probar.
2. Medir `contains` en `ArrayList` y `HashSet` con 10^5 elementos (`lab/ContainsTiming.java`).
3. Dos comparadores como clases, con pruebas que incluyan empates.
4. Hacer iterable el `BoundedStack<E>` de la semana 10: copien su versión en `lab/BoundedStack.java` (mismas firmas).

P3 en curso: entrega la próxima semana

Inventario en un `Map` por código, categorías con `Set`, orden con `Comparable` o `Comparator`.

Resumen

- Interfaces (`List`, `Set`, `Queue`, `Map`) e implementaciones; se programa contra la interfaz.
- `ArrayList` casi siempre. `HashSet`/`HashMap`: rápidos, sin orden; `Linked...`: orden de inserción; `Tree...`: ordenados.
- Las claves de un `HashMap` deben ser **inmutables**.
- `List.of` y compañía: inmutables. `unmodifiableList`: una vista para el getter.
- `Comparable`: el orden natural. `Comparator`: órdenes externos. Nunca restar. Un `TreeSet` trata como iguales a los que empatan.
- For-each = `Iterator`. Se borra con `it.remove()`, no con `list.remove`.
- `Iterable<E>` propio: delegando o con una clase interna.

Para la próxima clase

- Lecturas de la semana 12: Eck, sec. 4.5 (lambdas), subsecs. 5.8.3, 5.8.4 y 10.6.1, y 10.4.3 completa.
- Terminar P3 (Ventas): entrega la próxima semana.

Pregunta para pensar

`ByDuration` es una clase entera para escribir *una* línea: `Integer.compare(a.seconds(), b.seconds())`. ¿No habrá una forma más corta de pasarle esa línea a `sort`?