

UNIVERSIDAD NACIONAL DE INGENIERÍA

Programación Orientada a Objetos

Arreglos, `ArrayList` y diseño con clases

Muchos objetos a la vez, y cómo repartir el trabajo entre clases

Gerson Garrido

Semana 4

Hoy

1. Sesión A: arreglos vistos desde C, arreglos de referencias, un arreglo que crece
2. `ArrayList`, clases wrapper y autoboxing
3. Una `Playlist` encapsulada, con sus pruebas (`@BeforeEach`)
4. Sesión B: del enunciado al diseño, responsabilidades y tarjetas CRC
5. Diagramas de clases UML, en Mermaid

> **Lectura**
>
> Eck, *Introduction to Programming Using Java* (v. 9): secs. 3.8, 7.1, 7.2.4, 7.3, 5.3.4 y 5.4.1.

SECCIÓN 01

Arreglos de objetos

01

Arreglos: lo que ya saben de C, y lo que cambia

Igual que en C

Índices desde 0. `a[i]` para leer y escribir. Recorrer con `for`. Tamaño fijo al crearlo.

Distinto

- Es un objeto: `new`, heap, referencia.
- Conoce su longitud: `a.length`.
- Cada índice se verifica.
- Empieza en `0`, `false` o `null`.

Literales y for-each

Java

```java
int[] scores = {14, 17, 11};

int sum = 0;
for (int i = 0; i < scores.length; i++) {
    sum += scores[i];
}

int best = 0;
for (int s : scores) {   // para cada s en scores
    best = Math.max(best, s);
}
```

- `{14, 17, 11}`: un literal de arreglo *(array literal)*.
- **for-each**: da cada elemento, sin índice. No se equivoca de posición.
- El `for` clásico, cuando hace falta el índice: modificar `a[i]`, ir al revés, saltar.

`new int[3]` frente a `new Song[3]`

jshell

```
jshell> int[] counts = new int[3]
counts ==> int[3] { 0, 0, 0 }

jshell> Song[] queue = new Song[3]
queue ==> Song[3] { null, null, null }
```

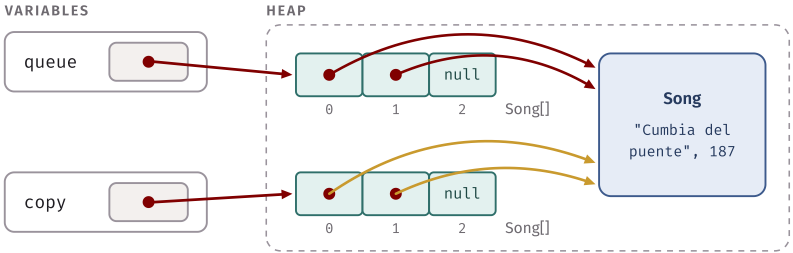
- `new int[3]`: tres enteros.
- `new Song[3]`: ninguna canción. Tres casillas para referencias, en `null`.
- Las canciones se crean aparte, cada una con su `new`.

Song: un record, como Money

`record Song(String title, String artist, int seconds)`, con un constructor compacto que rechaza un título vacío y una duración no positiva.

Un arreglo de objetos guarda referencias

```
Song[] queue = new Song[3];
queue[0] = new Song("Cumbia del puente", ...);
queue[1] = queue[0];
Song[] copy = queue.clone();
```



4. clone() crea otro arreglo, pero sus flechas llegan a la misma canción

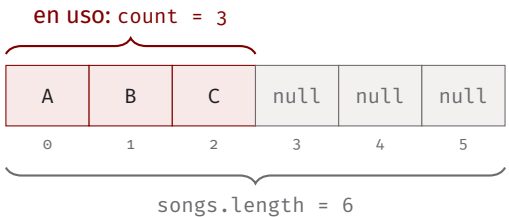
Lo mismo, ejecutado

bash

```
$ java -cp build music.ArraysDemo
counts[0] = 0, queue[0] = null
queue[0] == queue[1]: true
copy == queue: false, copy[0] == queue[0]: true
queue[2].title() ...
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
  "music.Song.title()" because "queue[2]" is null
    at music.ArraysDemo.main(ArraysDemo.java:19)
```

- `clone()` copia el arreglo, no las canciones: copia superficial *(shallow copy)*.
- Una casilla que nadie llenó sigue en `null`. El mensaje dice cuál: `queue[2]`.

Arreglo parcialmente lleno



- Un arreglo más grande de lo necesario y un contador de casillas en uso.
- Invariante: $0 \leq$ `count` $\leq$ `songs.length`, y valen las primeras `count`.
- ¿Y cuando se llena? No se le pueden agregar casillas…

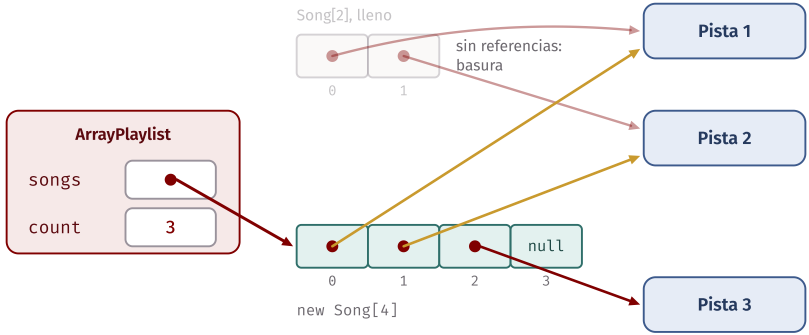
Un arreglo que crece: `ArrayPlaylist`

ArrayPlaylist.java

```java
public void add(Song song) {
    if (count == songs.length) {
        grow();
    }
    songs[count] = song;
    count++;
}
private void grow() {
    Song[] bigger = new Song[songs.length * 2];
    for (int i = 0; i < count; i++) {
        bigger[i] = songs[i];
    }
    songs = bigger;
}
```

- Campos: `Song[] songs` (capacidad 2 al inicio) y `int count`.
- Lleno: crear otro del doble, copiar, olvidar el viejo.
- Se copian referencias, no canciones.
- Duplicar hace raras las copias: en promedio, `add` es barato.
- `get(i)` verifica contra `count`, no contra `songs.length`.

`grow()`, paso a paso



4. add guarda Pista 3 en la casilla count, y count pasa a 3

`size` y `capacity`

PlaylistDemo.java

```java
ArrayPlaylist small = new ArrayPlaylist();
for (int i = 1; i <= 5; i++) {
    small.add(new Song("Pista " + i,
                       "Varios", 60 * i));
    IO.println("size " + small.size()
        + ", capacity " + small.capacity());
}
```

bash

```
size 1, capacity 2
size 2, capacity 2
size 3, capacity 4
size 4, capacity 4
size 5, capacity 8
```

Arreglo dinámico *(dynamic array)*: parcialmente lleno + reemplazo por uno del doble. `size` es lo que le importa a quien usa la clase; `capacity`, un detalle interno.

SECCIÓN 02

ArrayList

02



`ArrayList`: el arreglo dinámico, ya escrito

```java
import java.util.ArrayList;

ArrayList<Song> songs = new ArrayList<>();
songs.add(new Song("Lima gris",
                   "Grupo Garúa", 222));
Song first = songs.get(0);
int n = songs.size();
```

- `ArrayList<Song>`: “lista de `Song`”.
- El compilador no deja agregar otra cosa, y sabe que `get` devuelve un `Song`.
- `<>` vacío (el diamante): el tipo se deduce.
- Cómo se escribe una clase así: **genéricos**, semana 10.

Los métodos que más se usan

Método	Qué hace	Con un arreglo
`add(x)`	agrega al final	`a[count++] = x`, y crecer
`add(i, x)`	inserta en `i`, corre el resto	mover a mano
`get(i)` / `set(i, x)`	lee / reemplaza en `i`	`a[i]`
`remove(i)`	quita el de la posición `i`	mover a mano
`size()`, `isEmpty()`	cuántos hay	`count`
`contains(x)`, `indexOf(x)`	busca uno igual a `x`	recorrer y comparar

Por dentro: arreglo de 10 al primer `add`, y crece 1,5 veces. `new ArrayList<>(1000)` reserva de una vez.

Solo referencias: clases wrapper

jshell

```
jshell> ArrayList<int> bad = new ArrayList<>()
|  Error:
|  unexpected type
|    required: reference
|    found:    int
```

- Entre ángulos van tipos de referencia.
- Cada primitivo tiene su **clase wrapper**: `int` → `Integer`, `double` → `Double`, `char` → `Character`, `boolean` → `Boolean`...
- **Autoboxing**: de `int` a `Integer`. **Unboxing**: al revés. Los hace el compilador.
- Un `Integer` es un objeto en el heap: para muchos números, un `int[]` ocupa mucho menos.

Error común: `remove(1)`

BoxingDemo.java · con lo que imprime en cada paso

```java
ArrayList<Integer> list = new ArrayList<>();
list.add(5);  list.add(1);  list.add(7);  // [5, 1, 7]
list.remove(1);                  // [5, 7]     quita la posición 1
list.add(1);                     // [5, 7, 1]
list.remove(Integer.valueOf(1)); // [5, 7]     quita el valor 1
int first = list.get(0);         // unboxing: first es 5
```

- `list.add(5)`: el compilador escribe `list.add(Integer.valueOf(5))`.
- Hay dos `remove`: `remove(int index)` y `remove(Object o)`. Con un `int`, el compilador elige el de la posición, sin convertir nada.
- Para quitar el valor, pasen un objeto: `Integer.valueOf(1)`. El overloading de la semana 2, en contra.

SECCIÓN 03

Una lista encapsulada

03



`Playlist`: un `ArrayList` que nadie más ve

Playlist.java

```java
private final ArrayList<Song> songs =
        new ArrayList<>();

public void add(Song song) {
    if (song == null) {
        throw new IllegalArgumentException("...");
    }
    songs.add(song);
}
// totalSeconds(): un for-each que suma
// s.seconds() de cada canción
```

- El encapsulamiento de la semana 3, ahora con una colección.
- `final`: la variable apunta siempre a la misma lista; la lista sí cambia.
- Invariante: ninguna canción es `null`. Por eso `totalSeconds` no pregunta.

`longest` y `songs`

Playlist.java

```java
public Song longest() {
    if (songs.isEmpty()) {
        throw new IllegalArgumentException("...");
    }
    Song best = songs.get(0);
    for (Song s : songs) {
        if (s.seconds() > best.seconds()) {
            best = s;
        }
    }
    return best;
}
```

- Lista vacía: no hay respuesta. Devolver `null` pasa el problema a quien llama.
- Nunca `return songs;`: cualquiera podría hacer `.add(null)` o `.clear()`.
- `songs()` entrega una copia: `return new ArrayList<>(songs);` Las canciones se comparten: son inmutables.

`contains` compara con `equals`

PlaylistDemo.java

```
1  Song wanted = new Song("Cumbia del puente", "Los Rieleros", 187);
2  IO.println("¿tiene Cumbia del puente? " + trip.contains(wanted));
```

bash

```
$ java -cp build music.PlaylistDemo
...
¿tiene Cumbia del puente? true
```

- Preguntamos por un objeto nuevo, y responde `true`.
- `contains`, `indexOf` y `remove(Object)` usan `equals`: el del record compara componentes.
- Con una clase sin `equals` (`Thermometer`, `Copy`): igual que `==`. Semana 5.

Error común: quitar dentro de un for-each

RemoveInLoop.java

```java
// names = [alice, bob, carol, dave]
for (String n : names) {
    if (n.startsWith("b")) {
        names.remove(n);
    }
}
```

bash

```
Exception in thread "main"
  java.util.
    ConcurrentModificationException
  at ...checkForComodification(
    ArrayList.java:1096)
  at RemoveInLoop.main(RemoveInLoop.
    java:11)
```

- El for-each usa un **iterador**; si la lista cambia por otro lado, se detiene.
- Y a veces no falla: con `[alice, bob, carol, cindy]`, quitando los que empiezan con “c”, el resultado es `[alice, bob, cindy]`. Sin excepción.

Quitar bien: de atrás hacia adelante

Playlist.java

```java
public int removeShorterThan(int seconds) {
    int removed = 0;
    for (int i = songs.size() - 1; i >= 0; i--)
      {
        if (songs.get(i).seconds() < seconds) {
            songs.remove(i);
            removed++;
        }
    }
    return removed;
}
```

- Al quitar `i`, se corren los de después: esos ya se revisaron.
- En la semana 12: `removeIf`, en una línea.

¿Arreglo o `ArrayList`?

	Arreglo	**`ArrayList`**
Tamaño	fijo al crearlo	crece y se achica
Tipos	primitivos y referencias	solo referencias: `Integer`
Sintaxis	`a[i]`, `a.length`	`get(i)`, `size()`
Insertar/quitar en medio	a mano	`add(i, x)`, `remove(i)`
Buscar	a mano	`contains`, `indexOf`

Si la cantidad cambia mientras el programa corre: `ArrayList`. Si es fija y conocida (los 12 meses) o son muchos números primitivos: un arreglo.

Pruebas: `@BeforeEach` y `assertSame`

PlaylistTest.java

```java
class PlaylistTest {
    private Playlist list;   // y a, b, c
    @BeforeEach
    void setUp() {
        list = new Playlist("Prueba");
    }
    @Test
    void longestOfSeveral() {
        list.add(a); list.add(c); list.add(b);

        assertSame(c, list.longest());
    }
```

- El patrón de la semana 3.
- `@BeforeEach`: corre antes de cada prueba, en un objeto de prueba nuevo. Ninguna ve lo que hizo otra.
- `assertSame`: el mismo objeto (`==`). `assertNull(x)`: `x` es `null`.
- Bordes: vacía, repetida, empate.

SECCIÓN 04

Diseñar con clases

04



Nadie nos da las clases

Enunciado

Una biblioteca de barrio presta libros a sus socios. De cada **libro** se conoce ISBN, título y autor, y puede haber varios **ejemplares** del mismo libro. Un **socio** puede **pedir prestados** hasta tres ejemplares a la vez. Cada **préstamo vence** a los 14 días; al **devolver**, el ejemplar queda disponible. Se quiere saber cuántos ejemplares de un libro están disponibles y qué préstamos están **vencidos**.

- ¿Qué clases escribimos? ¿Qué hace cada una?
- No hay una única respuesta correcta, pero sí diseños que se dejan probar y cambiar, y otros que no.

Sustantivos y verbos

Sustantivos → candidatos a clase

- biblioteca, libro, ejemplar, socio, préstamo → clases
- ISBN, título, autor, código, nombre → campos: solo describen algo
- fecha → ya existe: `LocalDate`

Verbos → candidatos a método

- prestar, devolver, vencer, contar disponibles
- La pregunta difícil: ¿de qué clase?

Libro ≠ ejemplar

Un libro es un título; un ejemplar es el volumen del estante. El socio se lleva un ejemplar.

La responsabilidad va donde están los datos

Loan.java

```java
public boolean isOverdue(LocalDate today) {
    return !returned && today.isAfter(due);
}
```

Member.java

```java
public int overdueCount(LocalDate today) {
    int n = 0;
    for (Loan loan : activeLoans) {
        if (loan.isOverdue(today)) n++;
    }
    return n;
}
```

- ¿Quién decide si un préstamo venció? Quien tiene `due` y `returned`: `Loan`.
- En `Library`, habría que pedirle los datos al préstamo y decidir por él: *Tell, Don't Ask*.
- `Member` pregunta a cada préstamo, sin saber cómo se calcula.

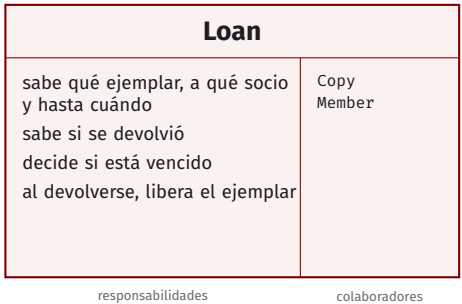
Prestar: coordina quien conoce a los dos

Library.java

```java
public Loan lend(Copy copy, Member member,
                 LocalDate today) {
    // rechaza con IllegalArgumentException si el
    // ejemplar no está disponible o el socio no
    // puede pedir más (canBorrow)
    LocalDate due = today.plusDays(LOAN_DAYS);
    Loan loan = new Loan(copy, member, due);
    copy.markOnLoan();
    member.addLoan(loan);
    return loan;
}
```

- `Copy` sabe si está disponible; `Member`, si puede pedir más.
- `Library`, que conoce a los dos, coordina.
- `new Loan`, `markOnLoan`, `addLoan`: acceso de paquete. Desde afuera, solo `lend` crea préstamos.

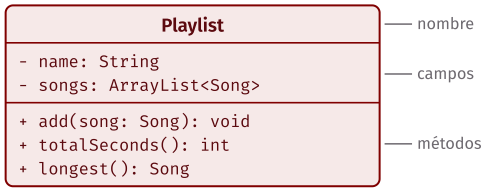
Tarjetas CRC



- Beck y Cunningham, 1989: **C**lase, **R**esponsabilidades, **C**olaboradores.
- Una cartulina por clase. En grupo, cada uno “actúa” una tarjeta en un escenario: *Alice pide un libro*.
- ¿Nadie tiene los datos? Falta algo. ¿Una tarjeta llena y las demás vacías? Esa clase hace demasiado.
- Son baratas: se rompen y se vuelven a escribir.

Diagramas de clases UML

UNA CLASE



`-` private `+` public `~` de paquete

el tipo va después de los dos puntos

RELACIONES

Asociación



A conoce a B (un campo)

Agregación



A tiene B; B existe por su cuenta

Composición



B es parte de un solo A

Herencia (semana 5)



A es un B

Asociación, agregación, composición

- **Asociación**: A conoce a B. En Java, un campo.
- **Multiplicidad**: `1`, `0..1`, `0..*`, `0..3`. “Un ejemplar es de exactamente 1 libro.”
- **Agregación** (rombo vacío): la biblioteca tiene socios; un socio existe por su cuenta.
- **Composición** (rombo lleno): el ejemplar es parte del inventario de una biblioteca.

> **En Java se ven igual**
>
> Las tres son un campo (o un `ArrayList`) con referencias. La diferencia es de diseño: quién crea las partes, quién más las conoce. UML como *boceto* para pensar y conversar, no como plano completo.

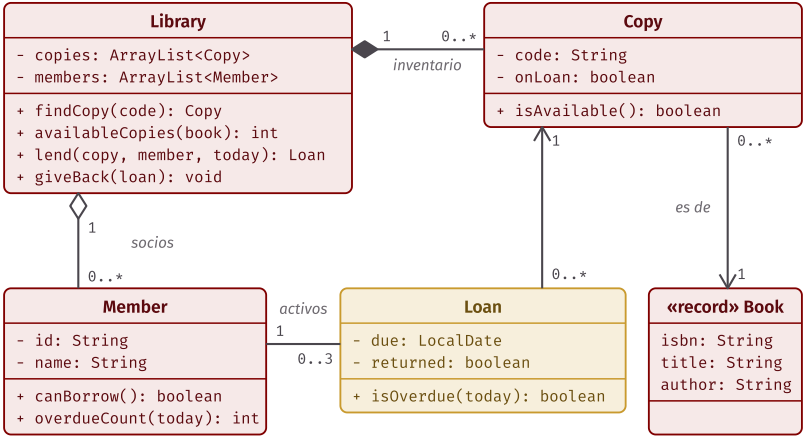
Mermaid: el diagrama como texto

biblioteca.mmd

```
classDiagram
    class Loan {
        -LocalDate due
        +isOverdue(LocalDate today) boolean
    }
    Library "1" *-- "0..*" Copy
    Library "1" o-- "0..*" Member
    Copy "0..*" --> "1" Book
    Loan "0..*" --> "1" Copy
    Member "1" --> "0..3" Loan
```

- Texto: se versiona con el código.
- GitHub y GitLab lo dibujan en cualquier Markdown; `mermaid.live`, mientras se escribe.
- `*--` composición, `o--` agregación, `-->` asociación; multiplicidades entre comillas.

La biblioteca, dibujada



Dos errores de diseño opuestos

God class

Una sola clase con todo: arreglos paralelos (`titles`, `onLoan`, `dueDates`) y cincuenta métodos. El estilo C de la semana 1 con `class` encima.

Clases anémicas

Solo datos, getters y setters; la lógica en `main` o en un "gestor". El cálculo de vencido, copiado en cada lugar.

Síntoma

Un método que solo usa datos de otra clase, pidiéndoselos con getters, quiere vivir en esa otra clase.

SECCIÓN 05

Laboratorio

05

Laboratorio: un gimnasio

Enunciado

Clases grupales con nombre, día, hora y **cupo**. Socios con código y nombre que se **inscriben** hasta en cuatro clases por semana; nunca dos veces en la misma, nunca en una clase llena. Se puede **cancelar**. ¿Quiénes están en una clase, cuántos lugares quedan, en qué clases está un socio?

1. Sustantivos y verbos. ¿Hace falta una clase para la inscripción?
2. Tarjetas CRC; actuar “hay un lugar” y “está llena”.
3. Diagrama de clases en Mermaid, en el `README.md`.
4. Dos o tres clases con `ArrayList` encapsulado y sus pruebas JUnit (`@BeforeEach`).
5. Revisión cruzada: ¿alguien devuelve su lista? ¿algún método vive en la clase equivocada?

make test

- El `Makefile` de la semana 3: paquetes `music` y `library`.
- `ArraysDemo` y `RemoveInLoop` terminan con excepción a propósito.
- 21 pruebas: `Song`, `Playlist`, `ArrayPlaylist`, `Library`.

bash

```
$ make test
== music.PlaylistDemo
Viaje: 4 canciones, 688 s
...
ArrayPlaylistTest, PlaylistTest,
SongTest, LibraryTest
...
[        21 tests successful      ]
[         0 tests failed          ]
```

Resumen

- Un arreglo de objetos guarda **referencias**: `new Song[3]` son tres `null`; `clone()` copia flechas.
- Arreglo **parcialmente lleno** + crecer al doble = **arreglo dinámico**. Eso es `ArrayList`.
- `ArrayList<Song>`: "lista de `Song`". Solo referencias: `Integer`, **autoboxing**; ojo con `remove(1)`.
- `contains` e `indexOf` usan `equals`. No modificar una lista en su propio for-each.
- Una clase con una colección la **encapsula**: lista privada, copias hacia afuera.
- `@BeforeEach` prepara cada prueba desde cero; `assertSame`, `assertNull`.
- Diseño: sustantivos y verbos; la responsabilidad va **donde están los datos**; CRC para pensar, **UML** (en Mermaid) para comunicar.

Para la próxima clase

- Lecturas de la semana 5: Eck, secs. 5.5 (5.5.1 a 5.5.3 y 5.5.6) y 5.6 (5.6.2 y 5.6.3).
- Laboratorio: terminar el gimnasio y sus pruebas.
- **P1** (Inventario): el inventario guarda sus productos en un `ArrayList` encapsulado. El diagrama de clases del laboratorio sirve de primer diseño.

Pregunta para pensar

`contains` encontró la canción porque `Song` es un record. Si `Product` fuera una clase sin `equals`, ¿encontraría `inventory.contains(...)` un producto con el mismo código? ¿Debería?