

# Refactorización, Testing y Collections

**UNRN**

Universidad Nacional  
de Río Negro

**XV**

**I**

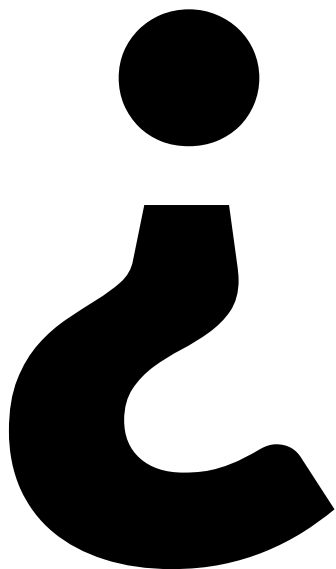
**2025**



---

# Refactorización

**Una de las  
formas**



# Como pagar deuda técnica



# Refactorizar es

el proceso de reestructurar el código existente sin cambiar su comportamiento externo.

**Los tests son  
esenciales**

**Ayudan a  
garantizar que los  
cambios no alteran  
el funcionamiento**

# Renombrar

Cambiar nombres de variables, métodos y clases para que sean más descriptivos

# Cambiar firma del método

Modificar los parámetros de una función para que sea más intuitiva o flexible.



# Extraer método

Dividir un bloque de código en una función aparte con un nombre claro.

*Fraccionando sus responsabilidades y consolida código repetido.*

# Incorporar función

Cuando un método se usa en un único lugar, puede integrarse directamente en el lugar donde se usa.

# Reemplazar primitivo

Cuando un atributo empieza a tener comportamiento asociado, conviene transformarlo en una clase.

# Aplicar polimorfismo

Sustituir `if` o `switch` con clases y métodos polimórficos.

# Pull up method

Si varias subclases comparten un método, este se mueve a su superclase.

# Pull down method

Si solo algunas subclases usan un método, se mueve fuera de la superclase.  
*Potencialmente introduciendo una nueva subclase.*

# Descomponer condicionales

Dividir condiciones largas en funciones  
descriptivas

# Consolidar expresiones condicionales

Agrupar condiciones repetidas en una sola.



# Aplicar estilo de código

Aplicar formato consistente en todo el proyecto (indentación, nombres, espaciado).



**¿Preguntas?**

---

# Testing II

## Cobertura



**¿Preguntas?**

---

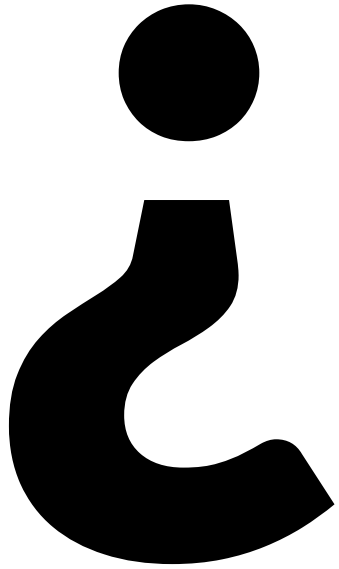
# Optional<T>

# El fantasma de NullPointerException

o el problema del billón de dólares







Que es  
null



# La clásica

```
if (usuario != null) {  
    String nombre = usuario.getNombre();
```

**Pero suma un montón de código repetido...**





NUEVO y  
MEJORADO

# Optional<T>

Resuelve todos sus problemas de  
NullPointerException



**Un Optional es una caja que puede estar vacía**

```
Optional<Usuario> sinUsuario = Optional.empty();
```

**Cuando explicitamente sabemos que no hay nada**

# Se puede crear de varias formas

```
Optional<String> nombreOpcional = Optional.of(nombre);
```

Para cuando es *literalmente* opcional y no sabemos si va a

# **Pero la más útil es la que podemos usar desde algo externo**

```
Usuario actual = basededatos.obtenerUsuarioActual();  
Optional<Usuario> usuario = Optional.ofNullable(nombre);
```

**Este o no, recibimos un Optional**

---

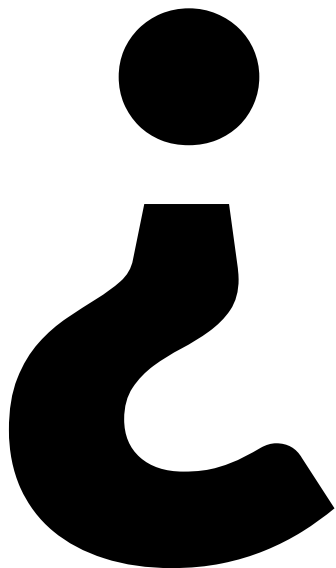
# Métodos

`T get()`

`T orElse(T otro)`

Obtiene el valor con excepcion

Obtiene el valor u 'otro'



**Cuando  
usarlo**



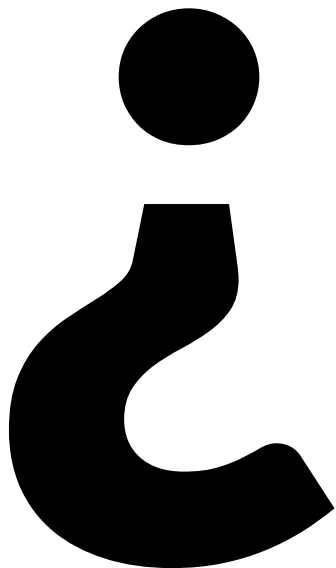
# 1

# Retornos de valores que pueden no estar

**2**

# **Atributos que pueden no estar presentes** (información opcional)





**Cuando no  
usarlo**



# 1

# Atributos base

# 2

## Parámetros de métodos\*

Ver si el código no queda mas complicado...

**En general, es mejor usar sobrecarga**



**¿Preguntas?**

---

# Java Collections

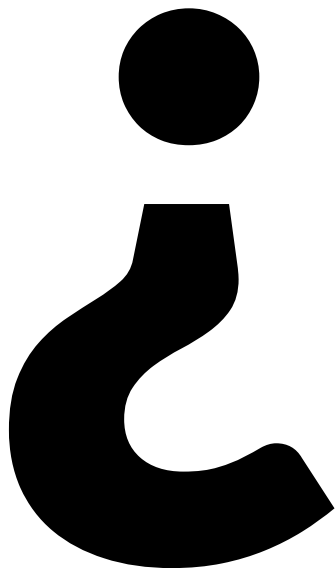
**UNRN**

Universidad Nacional  
de Río Negro

**JCF**

**Porque no todo  
son  en la vida**

**Y guardar conjuntos  
de objetos es  
*bastante* habitual**

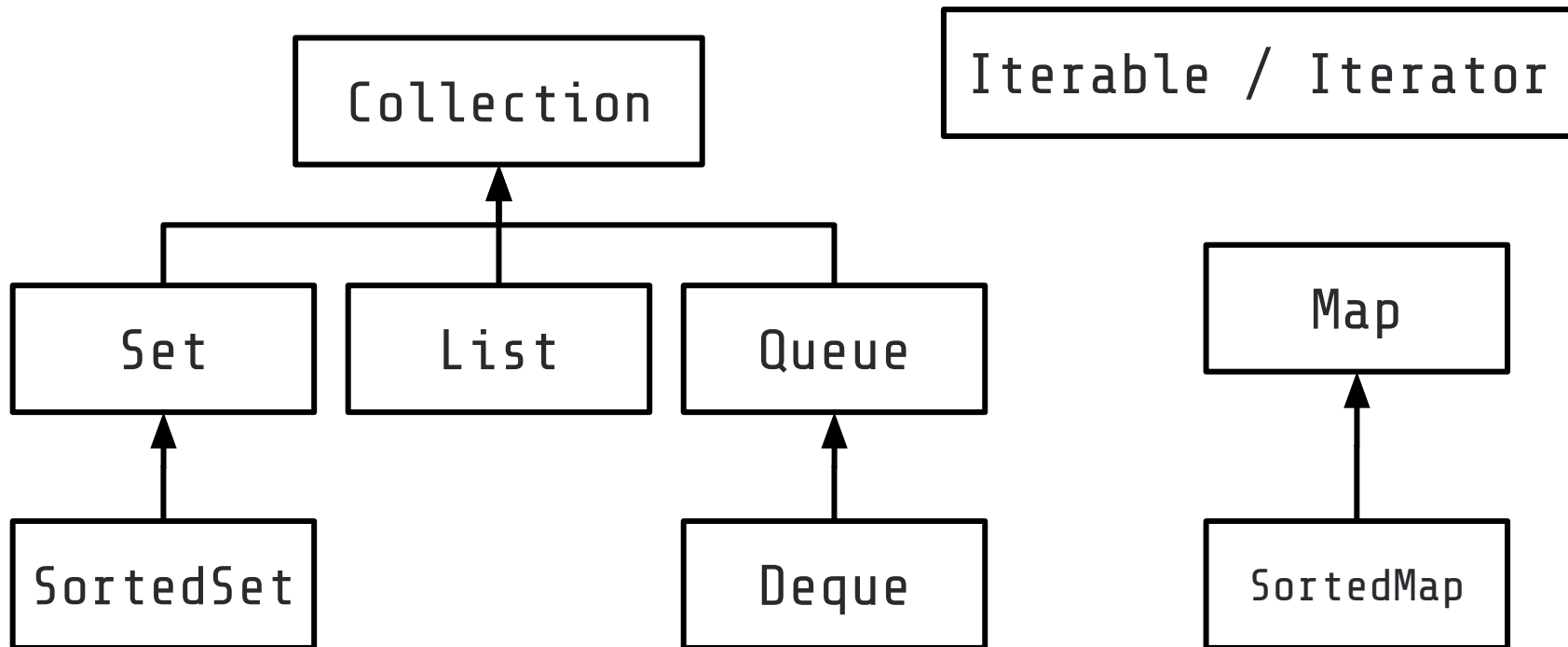


**Qué son**





# La familia de interfaces\*

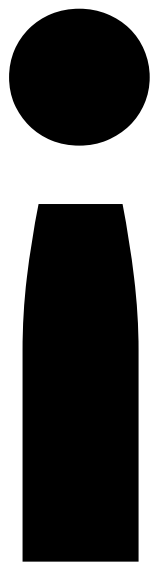


# Collection<E> - colección

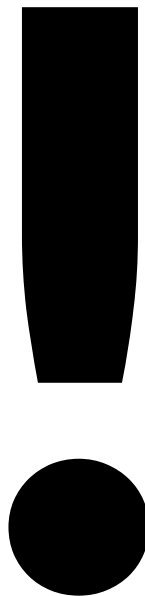
Esta interfaz define el comportamiento mínimo común denominador para grupos de objetos llamados *elementos* de un tipo E

# Operaciones definidas en la interfaz

```
add(E e)  
remove(Object e)  
contains(Object o)  
clear()  
iterator()  
toArray()  
isEmpty()  
size()  
equals / hashCode
```



**<? extends E>**



Algún tipo desconocido que es una subclase de E o E mismo.

# Set<E> - conjunto

Un grupo de elementos de un tipo E sin repetidos y sin orden.

**Ojo con usar  
valores  
mutables**

# List<E> - conjunto

Un grupo de elementos de un tipo E  
con orden.

(en líneas generales, como un arreglo)

```
addFirst(E e)
addLast(E e)
removeFirst(E e)
removeLast(E e)
E get(int index)
E getFirst()
E getLast()
reversed
set(int posicion, E e)
List<E> subList(int fromIndex, int toIndex)
```



# Queue<E> - fila

Un grupo de elementos de un tipo E con orden, en donde lo que entra primero sale primero.

# Operaciones en Queue \*

`add(E elemento)`

`remove()`

`element()`

Piensen en una  
fila del  
supermercado

**Las operaciones lanzan excepciones si hay problemas**

# Operaciones en Queue

`offer(E elemento)`

`E poll()`

`E peek()`

**Estas devuelven un valor especial ( `null` ) si hay problemas**

# Deque<E> - fila doble

Un grupo de elementos de un tipo E con orden, en donde se puede agregar y remover elementos desde los extremos.

# Operaciones en Deque

`addFirst(E)`

`addLast(E)`

`E removeFirst()`

`E removeLast()`

`E getFirst()`

`E getLast()`

**Piensen en una  
fila de un  
restaurant (con  
prioridad)**

# Map<K, E> - Diccionarios

Un grupo de elementos sin un orden en particular de un tipo E que puede ser ubicado con una llave de tipo K.

# Operaciones en Map

`containsKey(K)`

`put(K, V)`

`V get(K)`

`V remove(K)`

`V replace(K, V)`



**Recuerden igual  
que son interfaces**

***No hay código***



**Vamos a ver  
detalles sobre estas  
estructuras más  
adelante**

# 2

# Implementaciones

# List

ArrayList  
LinkedList

# Set

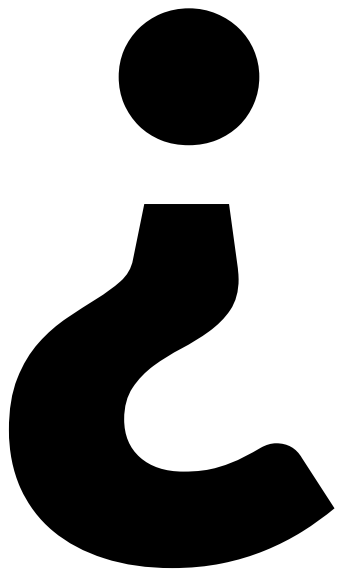
HashSet  
TreeSet

# Map

HashMap  
TreeMap

# Queue

PriorityQueue  
ArrayDeque  
Queue  
Deque



**¿¿Por qué hay  
dos de cada??**



# Complejidad

en tiempo y espacio

0x0000

*Cuestiones de estilo*

**Al usar una  
Collection, la  
variable\* del tipo  
de la Interfaz**

```
List lista = new LinkedList()
```

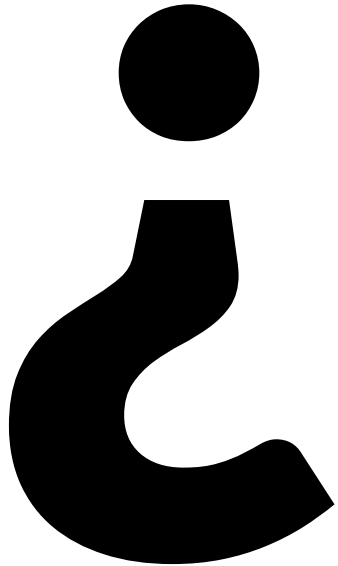


**¿Preguntas?**



**que son los**

# **Patrones de diseño**



# 1

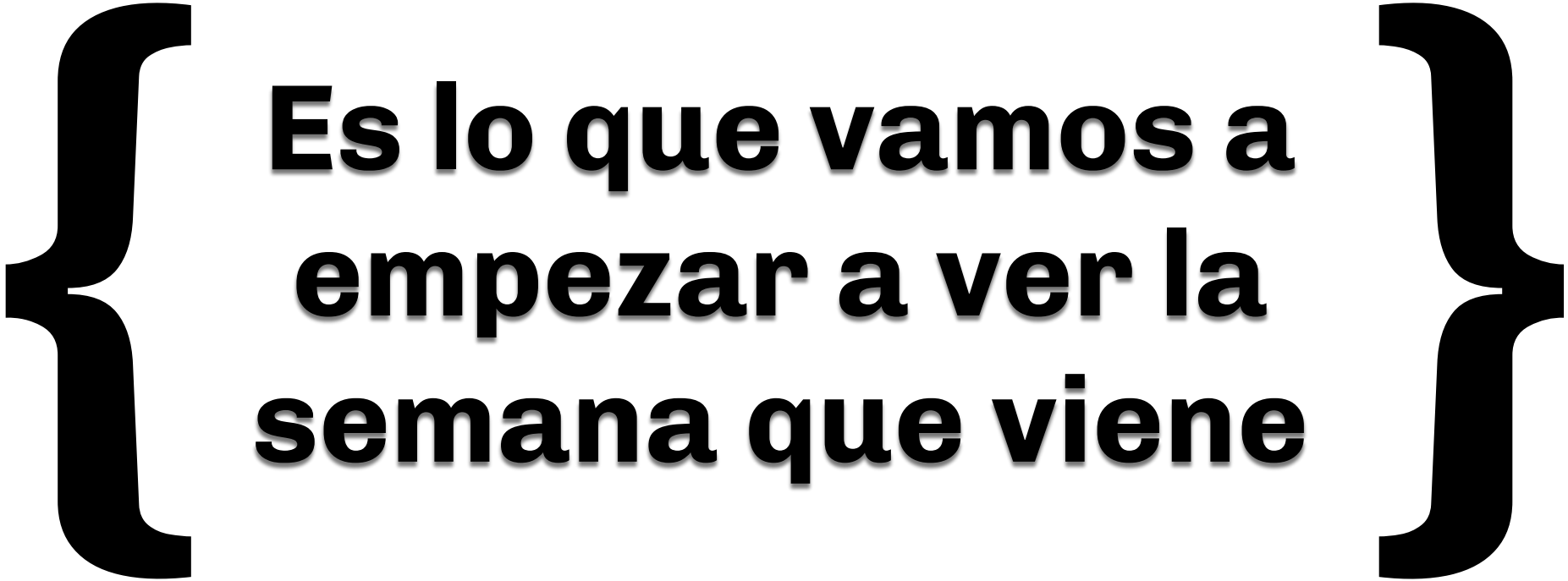
# Vocabulario

**2**

# **Soluciones comunes a problemas comunes**



**¿Preguntas?**



**Es lo que vamos a  
empezar a ver la  
semana que viene**

**Hasta la  
próxima**

