12 Programación Funcional

1. Introducción
2. Interfaces funcionales
3. Implementación de interfaces funcionales mediante clases inline anónimas
4. Interfaces funcionales puras predefinidas
5. Interfaces funcionales para tipos primitivos
6. Expresiones lambda
    6.1 Sintaxis de la expresión lambda
    6.2 Ámbito de una expresión lambda
    6.3 Limitaciones de las expresiones lambda
7. Referencias a método
8. Consumer y BiConsumer
9. Predicate y BiPredicate
10. Function
11. Comparator
12. Optional
    12.1 Construcción
    12.2 Igualdad
    12.3 Procesamiento
    12.4 Dónde no usar Optional

1. Introducción

La **programación imperativa** es uno de los paradigmas de programación de computadoras más utilizados. Bajo este paradigma, la programación se describe en términos del estado del programa y de sentencias que cambian dicho estado. Java es un lenguaje imperativo, lo que implica que un programa Java está compuesto por una secuencia de instrucciones, que son ejecutadas en el mismo orden en el que se escriben, de manera que al ejecutarla se produce cambios en el estado del programa.

Por su parte, la **programación funcional** es un paradigma de programación alternativo, en el que el resultado de un programa deriva de la aplicación de distintas funciones a la entrada, sin cambiar el estado interno del programa. En la programación funcional los bloques principales de construcción de nuestros programas son las funciones y no los objetos.

Al aplicar programación funcional se produce normalmente un código más corto y más sencillo de entender que aplicando programación imperativa, ya que es más fácil crear abstracciones a través de funciones que a través de interfaces.

Java siempre fue un lenguaje para programación imperativa y de hecho las funciones en Java NO son objetos, por lo que una función no puede pasarse directamente como argumento de otra función para que se ejecute su código. Sin embargo, gracias a las interfaces funcionales y a las clases inline anónimas podíamos superar esta limitación. Pero ¿qué es una interfaz funcional?

2. Interfaces funcionales

Una interfaz funcional es una interfaz que contiene **un único método abstracto**. Esto no quiere decir que no pueda contener otros métodos. De hecho, puede contener:

- Otros métodos `static` (Java 8+).
- Otros métodos `default` (Java 8+).
- Otros métodos `private` (Java 9+)
- Métodos que sobrescriban métodos de la clase `Object`.

A la hora de definir una interfaz funcional, Java 8 proporciona la anotación `@FunctionInterface`, que informa al compilador de que dicha interfaz es funcional y por tanto tiene un único método abstracto. El objetivo de esta anotación es que se produzca un error de compilación si le añadimos un segundo método abstracto a la interfaz. El uso de esta anotación no se ha establecido como obligatoria para mantener la compatibilidad con el código ya existente, pero sí que está recomendada.

Una **interfaz funcional pura** es aquella en la que las clases que la implementan no almacenan ningún estado, como por ejemplo `Comparator`. Veamos el método `sort` de la interfaz `List<E>` que recibe un objeto de una clase que implementa la interfaz `Comparator`: `default void sort(Comparator<? super E> c)`. El método utiliza el `Comparator` para ordenar la lista llamando al método `compare` de dicho objeto cada vez que debe comparar dos objetos de la lista. Por lo tanto, debemos crear una clase que implemente `Comparator` para determinar cómo se comparan dos elementos:

```java
package tema12_ProgramacionFuncional.comparator;

import java.util.Comparator;

public class ListOrder implements Comparator<Integer> {

    @Override
    public int compare(Integer o1, Integer o2) {
        return Integer.compare(o1, o2);
    }

}
```

Así, cuando queramos ordenar una lista de enteros haríamos:

```java
package tema12_ProgramacionFuncional.comparator;

import java.util.Arrays;
import java.util.List;

public class Main{

    public void show() {
        List<Integer> list = Arrays.asList(3, 2, 6, 1, 5, 4);
        list.sort(new ListOrder());
        for (Integer i : list) {// 1  2  3  4  5  6
            System.out.printf(" %d ", i);
        }
    }

    public static void main(String[] args) {

        new Main().show();

    }

}
```

3. Implementación de interfaces funcionales mediante clases inline anónimas

El problema del código anterior es que si esta ordenación se hace solamente en dicha ocasión, se ha creado la clase `ListOrder` para un único uso. En ese caso, es más conveniente utilizar una clase inline anónima:

```java
package tema12_ProgramacionFuncional;
```

```java
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class InlineAnonymousClass{

    public void show() {

        List<Integer> list = Arrays.asList(3, 2, 6, 1, 5, 4);
        list.sort(new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return Integer.compare(o1, o2);
            }
        });
        for (Integer i : list) {// 1  2  3  4  5  6
            System.out.printf(" %d ", i);

        }

    }

    public static void main(String[] args) {

        new InlineAnonymousClass().show();

    }

}
```

Lo que estamos haciendo es indicarle al método `sort()` el código que debe ejecutar para comparar dos objetos. Entonces, ¿no sería más fácil que al método `sort()` le pudiéramos pasar directamente el código que debe ejecutar? El problema es que en Java las funciones no son objetos, por lo que no pueden ser referenciadas mediante una variable o pasadas directamente como argumento.

En realidad, lo que nos interesa es poder establecer *tipos función*, es decir, tipos que representen una función que reciba unos determinados parámetros de algún tipo y que devuelva un valor de retorno de algún tipo. Si existieran los *tipos función*, podríamos definir variables o parámetros de dichos tipos. De hecho hay lenguajes de programación que tienen *tipos función*. En Java, debido a la necesidad de mantener la compatibilidad con versiones anteriores, no existe ninguna sintaxis especial para definir *tipos función* sino que se utilizan las interfaces funcionales para representarlos. Dado que una interfaz funcional solo puede tener un único método abstracto, la firma de dicho método puede ser usado como *tipo función*.

4. Interfaces funcionales puras predefinidas

Java incorpora, a partir de la versión 8, una serie de interfaces funcionales puras predefinidas en el paquete `java.util.function` para permitir la programación funcional en Java:

- `Function<T,R>` : su método abstracto es `R apply(T t)` .
- `UnaryOperator<T>` : es un caso específico de la interfaz funcional *Function<T,T>*, es decir, coinciden el tipo del argumento y el tipo de retorno, por lo que está parametrizada con un único tipo .
- `BiFunction<T,U,R>` : su método abstracto es `R apply(T t,U u)` .
- `BinaryOperator<T>` : es un caso específico de la interfaz funcional *BiFunction<T,T,T>*en el que coinciden el tipo de los dos argumentos recibidos por el método *apply* y el tipo de retorno del mismo. Es por tanto similar a *BiFunction<T,T,T>*. La interfaz funcional *BinaryOperator* está, por tanto, parametrizada con un único tipo .

- `Predicate<T>` : su método abstracto es `boolean test(T t)` .
- `BiPredicate<T>` : su método abstracto es `boolean test(T t,U u)` .
- `Consumer<T>` : su método abstracto es `void accept(T t)` .
- `BiConsumer<T,U>` : su método abstracto es `void accept(T t,U u)` .
- `Supplier<T>` : su método abstracto es `T get()` .

Veamos un ejemplo utilizando la interfaz funcional `BinaryOperator<T>` :

```
package tema12_ProgramacionFuncional.binaryOperator;

import java.util.function.BinaryOperator;

public class ShowBinaryOperator{

    public Integer calculate(Integer value1,Integer
value2,BinaryOperator<Integer> binaryOperation) {
        return binaryOperation.apply(value1,value2);
    }

}
```

```
package tema12_ProgramacionFuncional.binaryOperator;

import java.util.function.BinaryOperator;

public class Main{

    public void show() {
        ShowBinaryOperator binOper = new ShowBinaryOperator();
```

```
        System.out.printf("12 + 6 = %d\n", binOper.calculate(12, 6,
new BinaryOperator<Integer>() {

            @Override
            public Integer apply(Integer t, Integer u) {
                return t + u;
            }

        }));
        System.out.printf("12 - 6 = %d\n", binOper.calculate(12, 6,
new BinaryOperator<Integer>() {

            @Override
            public Integer apply(Integer t, Integer u) {
                return t - u;
            }

        }));
        System.out.printf("12 / 6 = %d\n", binOper.calculate(12, 6,
new BinaryOperator<Integer>() {

            @Override
            public Integer apply(Integer t, Integer u) {
                return t / u;
            }

        }));
        System.out.printf("12 * 6 = %d\n", binOper.calculate(12, 6,
new BinaryOperator<Integer>() {

            @Override
            public Integer apply(Integer t, Integer u) {
                return t * u;
            }

        }));

    }

    public static void main(String[] args) {

        new Main().show();

    }

}
```

Salida por consola:

```
12 + 6 = 18
12 - 6 = 6
12 / 6 = 2
12 * 6 = 72
```

5. Interfaces funcionales para tipos primitivos

Como no podemos usar la parametrización de clases e interfaces con los tipos primitivos (limitación de *generics*), el paquete `java.util.function` define también una serie de interfaces funcionales similares a las explicadas anteriormente pero específicas para los tipos primitivos:

- Para el tipo primitivo **boolean**: `BooleanSupplier`
- Para el tipo primitivo **double**: `DoubleBinaryOperator`, `DoubleConsumer`, `DoubleFunction`, `DoublePredicate`, `DoubleSupplier`, `DoubleToIntFunction`, `DoubleToLongFunction`, `DoubleUnaryOperator`, `ToDoubleBiFunction`, `ToDoubleFunction`, `ObjDoubleConsumer`.
- Para el tipo primitivo **int**: `IntBinaryOperator`, `IntConsumer`, `IntFunction`, `IntPredicate`, `IntSupplier`, `IntToDoubleFunction`, `IntToLongFunction`, `IntUnaryOperator`, `ToIntBiFunction`, `ToIntFunction`, `ObjIntConsumer`.
- Para el tipo primitivo **long**: `LongBinaryOperator`, `LongConsumer`, `LongFunction`, `LongPredicate`, `LongSupplier`, `LongToDoubleFunction`, `LongToIntFunction`, `LongUnaryOperator`, `ToLongBiFunction`, `ToLongFunction`, `ObjLongConsumer`.

Además, la mayoría de las interfaces vistas hasta ahora incluyen métodos cuyo nombre incluye `ToTipo` que retornan objetos de interfaces funcionales para tipos primitivos.

6. Expresiones lambda

Con objeto de incorporar a Java funcionalidades propias de la programación funcional, Java 8 trajo consigo dos nuevas sintaxis para representar interfaces funcionales: expresiones lambda (*lambda expressions*) y referencias a método (*method references*).

Una **expresión lambda** es una nueva sintaxis con la que representar la implementación del método abstracto de interfaces funcionales indicando además la lista de parámetros con sus tipos y el tipo de retorno. De esta manera, podemos escribir el código de apartados anteriores mediante una expresión lambda, haciéndolo mucho más legible.

El nuevo operador para las expresiones lambda se denomina **operador lambda** y tiene la forma de flecha `->`. Divide la expresión lambda en dos partes: la parte izquierda especifica los parámetros necesarios y la parte derecha contiene el cuerpo de la expresión. Este cuerpo puede estar compuesto por una única expresión o puede ser un

bloque de código. Cuando es una única expresión se denomina **lambda de expresión** y cuando es un bloque de código se denomina **lambda de bloque**.

Debemos tener en cuenta que cuando se especifica una expresión *lambda*, no indicamos nada sobre la interfaz funcional a la queremos aplicarla, es decir, dependiendo de donde se esté usando la expresión *lambda*, el compilador deberá determinar si la firma de la expresión *lambda* coincide con la firma del método abstracto de la correspondiente interfaz funcional. Si la expresión *lambda* no incluye los tipos de los parámetros, el compilador tratará de inferirlos a partir de los tipos de los parámetros del método abstracto de la interfaz funcional.

El ejemplo anterior de la interfaz `BinaryOperator<Integer>`, si lo realizamos con una expresión lambda, resultaría de la siguiente manera:

```
package tema12_ProgramacionFuncional.binaryOperator;

public class Main2 {

    public void show() {
        ShowBinaryOperator binOper = new ShowBinaryOperator();
        System.out.printf("12 + 6 = %d\n", binOper.calculate(12, 6,
(t, u) -> t + u));
        System.out.printf("12 - 6 = %d\n", binOper.calculate(12, 6,
(t, u) -> t - u));
        System.out.printf("12 / 6 = %d\n", binOper.calculate(12, 6,
(t, u) -> t / u));
        System.out.printf("12 * 6 = %d\n", binOper.calculate(12, 6,
(t, u) -> t * u));

    }

    public static void main(String[] args) {

        new Main2().show();

    }

}
```

Salida por consola:

```
12 + 6 = 18
12 - 6 = 6
12 / 6 = 2
12 * 6 = 72
```

Veamos el ejemplo de ordenación de la lista hecho de las dos maneras, con una clase inline anónima y con una expresión lambda:

```
package tema12_ProgramacionFuncional;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class LambdaExpression {

    public void show() {

        List<Integer> list = Arrays.asList(3, 2, 6, 1, 5, 4);
        list.sort(new Comparator<Integer>() {//Clase inline anónima
            @Override
            public int compare(Integer o1, Integer o2) {
                return Integer.compare(o1, o2);
            }
        });
        for (Integer i : list) {// 1  2  3  4  5  6
            System.out.printf(" %d ", i);

        }
        System.out.println();
        list.sort((o1, o2) -> Integer.compare(o1, o2));//Expresión
lambda
        for (Integer i : list) {// 1  2  3  4  5  6
            System.out.printf(" %d ", i);

        }

    }

    public static void main(String[] args) {

        new LambdaExpression().show();

    }

}
```

Aún así, debemos tener en cuenta que un objeto de una clase inline anónima y una expresión *lambda* no son lo mismo, porque una expresión *lambda* no crea ninguna clase adicional y además en una expresión *lambda* no se puede almacenar ningún estado, mientras que en una clase anónima inline sí. Un objeto de una clase anónima inline genera un archivo de clase independiente durante la compilación que aumenta el tamaño del archivo *jar*. Sin embargo, una expresión lambda se convierte en un simple

método privado. En una lambda, *this* representa la clase actual desde la que se está usando la lambda. En el caso de una clase anónima, *this* representa ese objeto de la clase anónima en particular.

Como vemos, las expresiones *lambda* son muy útiles para simplificar el código, pero presentan un problema: para que podamos usar esta nueva sintaxis de expresión *lambda* es necesario que la interfaz contra la que la usemos sea una interfaz funcional, es decir, que sólo contenga un único método abstracto, porque si tuviera por ejemplo dos métodos, ¿el código proporcionado mediante la expresión *lambda* cuándo se debería ejecutar, cuando se ejecute uno o cuando se ejecute el otro? Por este motivo las expresiones *lambda* solo se pueden usar con interfaces funcionales.

Se puede almacenar una expresión *lambda* en una variable cuyo tipo corresponda a una interfaz funcional compatible con dicha lambda, es decir, cuya firma del método abstracto de la interfaz sea compatible con la expresión *lambda*. Por ejemplo:

```
Comparator<Integer> comparador = (o1, o2) -> Integer.compare(o1, o2);
BinaryOperator<Integer> operacionBinaria = (o1, o2) -> o1 + o2;
```

6.1 Sintaxis de la expresión lambda

Veamos la **sintaxis** de la expresión *lambda*:

```
(tipo param1, tipo param2, ...) -> {
    cuerpoExpresionLambda
    return valorRetorno
}
```

Podemos omitir el tipo de dato de cada parámetro, siempre y cuando el compilador pueda inferirlos (deducirlos) a partir del contexto, es decir, a partir de los tipos de los parámetros del método abstracto de la interfaz funcional para la que se está usando.

El cuerpo de las expresiones *lambda* puede contener los mismos tipos de sentencias que cualquier otra función, como sentencias condicionales, iterativas o `try catch`.

Si se especifica un único parámetro y no se especifica el tipo de éste sino que es inferido, podemos omitir los paréntesis. Por ejemplo:

```
x -> x + x;
```

Si no se especifica ningún parámetro, es obligatorio poner los paréntesis. Por ejemplo:

```
() -> System.out.println("Hola mundo");
```

En el cuerpo podemos omitir las llaves si éste contiene una única expresión o una única sentencia que no retorna valor. Si el cuerpo contiene una única expresión, ésta será evaluada y la expresión *lambda* retornará el valor obtenido.

Si el cuerpo contiene más de una sentencia y la expresión *lambda* debe retornar un valor, entonces debemos usar una sentencia `return valor`.

Si el cuerpo contiene una sentencia `return valor`, forzosamente debemos poner las llaves, incluso si el cuerpo contiene una única sentencia, ya que `return` no es una expresión.

Una expresión *lambda* se puede usar como argumento de un parámetro de tipo interfaz funcional y como valor de retorno de una función cuyo tipo de retorno sea una interfaz funcional. Sin embargo, habrá ocasiones donde debamos realizar un *cast* explícitamente para indicar la interfaz funcional a la que queremos aplicar una determinada expresión *lambda*.

6.2 Ámbito de una expresión lambda

Una expresión lambda puede acceder a las variables `static` definidas en el ámbito en el que la expresión *lambda* es usada. También puede acceder a las variables locales pero que sean eficazmente finales, es decir, variables cuyo valor no cambia una vez asignado. Estas variables no tienen necesariamente que estar definidas como `final`. Una expresión lambda también tiene acceso a `this`, lo que hace referencia a la instancia de invocación de la clase contenedora de la expresión lambda.

Si en el cuerpo de una expresión *lambda* con más de una sentencia definimos una variable local, debemos tener en cuenta que dicha variable tendrá como ámbito el correspondiente a donde se ha definido la expresión lambda, ya que la expresión *lambda* no define su propio ámbito independiente. Si ya existiera una variable con el mismo nombre en dicho ámbito se produciría un error de compilación. Por ejemplo:

```
int z = 2;
BinaryOperator<Integer> operacion = (x, y) -> {
    // ERROR: z ya está definida en el ámbito
    int z = 4;
    System.out.println(x + z);
};
```

Una expresión lambda puede generar una excepción. No obstante, si genera una excepción comprobada, esta tendrá que ser compatible con la excepción (o excepciones) indicadas en la cláusula `throws` del método abstracto de la interfaz funcional. Veamos un ejemplo:

```java
package tema12_ProgramacionFuncional;

public interface FunctionalInterface {
    int ioAction() throws Exception;
}
```

```java
package tema12_ProgramacionFuncional;

import java.util.Scanner;

public class LambdaException {

    @SuppressWarnings("resource")
    public void show() {

        FunctionalInterface fi = () -> {
            Scanner keyboard = new Scanner(System.in);
            int num = keyboard.nextInt();
            return num;
        };
        try {
            System.out.printf("Introduce un número:");
            System.out.println(method(fi));
        } catch (Exception e) {
            System.out.println("Error en la lectura");
        }

    }

    public int method(FunctionalInterface fi) throws Exception {
        return fi.ioAction();
    }

    public static void main(String[] args) {

        new LambdaException().show();

    }

}
```

6.3 Limitaciones de las expresiones lambda

El código de una expresión *lambda* se convierte en el código del método abstracto de la interfaz funcional que implementa. Por tanto, las expresiones *lambda* no sirven para sobrescribir la implementación por defecto de un método *default* de la interfaz, sino que debe tratarse de un método abstracto. De hecho si la interfaz solo tiene un método *default*, no será considerada una interfaz funcional. Si nos vemos en la

obligación de sobrescribir un método *default* de una interfaz, entonces tendremos que usar una clase anónima *inline*.

Por otro lado, una expresión *lambda* no es consciente de qué interfaz funcional concreta está implementando, por lo que no puede llamar a su vez a métodos privados ni *default* de la interfaz.

Finalmente, las expresiones *lambda* no pueden usarse con clases abstractas que tengan un único método abstracto, solo se pueden usar con interfaces funcionales.

7. Referencias a método

Una referencia de método (*method reference*) permite hacer referencia a un método sin ejecutarlo. Al evaluar una referencia de método, también se crea una instancia de una interfaz funcional.

- Sintaxis para métodos estáticos: `NombreClase::nombreMétodo`

  Ejemplos:

  - `v -> Math.sqrt(v)` equivaldría a `Math::sqrt`
  - `(o1, o2) -> Integer.compare(o1, o2)` equivaldría a `Integer::compare`
- Sintaxis para métodos de instancia: `refObj::nombreMétodo`

  Ejemplos:

  - `persona -> persona.getNombre()` equivaldría a `Persona::getNombre`
  - `n -> System.out.println(n)` equivaldría a `System.out::println`
  - `(cadena1, cadena2) -> cadena1.compareToIgnoreCase(cadena2)` equivaldría a `String::compareToIgnoreCase`: en este caso, el primer parámetro de la expresión lambda es quien ejecuta el método y el resto de parámetros se pasan como argumentos en la llamada.
  - `empleado -> jefe.comparaSalarioCon(empleado)` equivaldría a `jefe::comparaSalarioCon`: en este caso, un objeto ajeno a la expresión lambda es quien ejecuta el método y dicho método recibe como argumento el (o los) parámetro(s) de la expresión lambda.
  - `() -> new TreeMap<>()` equivaldría a `TreeMap::new`: este tipo se conoce como referencia a constructor, que emplearemos cuando queramos que se llame al método constructor de una clase.
  - `i -> new int[i]` equivaldría a `int[]::new`: en este caso, lo que queremos es que se llame al constructor de un array.
- Sintaxis para métodos genéricos:
  - Estáticos: `NombreClase::<T>nombreMétodo`
  - Métodos de instancia: `refObj::<T>nombreMétodo`

8. Consumer y BiConsumer

- `Consumer<T>` : su método abstracto es `void accept(T t)` .
- `BiConsumer<T,U>` : su método abstracto es `void accept(T t,U u)` .

La interfaz `Consumer` es empleada por el método `forEach(Consumer<T> action)` de la interfaz `Iterable` , que ejecuta la acción indicada sobre cada elemento del iterable.

```java
package tema12_ProgramacionFuncional;

import java.util.List;

public class InterfaceConsumer {

    public void show() {

        List<Integer> list = List.of(3, 2, 6, 1, 5, 4);
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new InterfaceConsumer().show();

    }

}
```

Salida por consola:

```
3
2
6
1
5
4
```

En este ejemplo, se ejecuta la acción de mostrar por consola una línea con cada elemento de la lista.

La interfaz funcional `BiConsumer` es similar a `Consumer` pero su método recibe dos argumentos, uno de tipo `T` y otro de tipo `U` y no retorna nada: `void accept(T t, U u)`

```java
package tema12_ProgramacionFuncional;
```

```
import java.util.HashMap;
import java.util.Map;

public class InterfaceBiConsumer {

    public void show() {

        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < 5; i++) {
            map.put(i, i * i);
        }
        map.forEach((k, v) -> System.out.printf("Clave:%d Valor:%d\n",
k, v));

    }

    public static void main(String[] args) {

        new InterfaceBiConsumer().show();

    }

}
```

Salida por consola:

```
Clave:0 Valor:0
Clave:1 Valor:1
Clave:2 Valor:4
Clave:3 Valor:9
Clave:4 Valor:16
```

La interfaz funcional `Consumer` posee un método *default* llamado `andThen(Consumer<T> after)` que llama al método `accept` del consumidor recibido después de haber llamado a su propio `accept`. Gracias a este método, podemos tener una serie de objetos `Consumer` predefinidos y encadenarlos de la forma que nos interese.Veamos un ejemplo:

```
package tema12_ProgramacionFuncional;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerAndThen {

    public void show() {
```

```
        List<Integer> list = List.of(3, 2, 6, 1, 5, 4);
        List<Integer> listCopy = new ArrayList<>();
        Consumer<Integer> copy = listCopy::add;
        Consumer<Integer> show = System.out::println;
        list.forEach(copy.andThen(show));
        listCopy.forEach(show);

    }

    public static void main(String[] args) {

        new ConsumerAndThen().show();

    }

}
```

Salida por consola:

```
3
2
6
1
5
4
3
2
6
1
5
4
```

9. Predicate y BiPredicate

- `Predicate<T>`: su método abstracto es `boolean test(T t)`.
- `BiPredicate<T>`: su método abstracto es `boolean test(T t,U u)`.

Las interfaces funcionales `Predicate` y `BiPredicate` poseen métodos *default* que retornan un nuevo objeto que implementa la misma interfaz y que permiten componer predicados mediante operaciones lógicas, como `or(otherPredicate)`, `and(otherPredicate)` o `negate(otherPredicate)`. El orden en el que se ejecutarán serán el orden en el que aparecen en la composición, es decir, no existe una prioridad preestablecida como con los operadores lógicos.

Veamos un ejemplo utilizando el método `removeIf` de la interfaz Collection que recibe un *Predicate* por parámetro: `default boolean removeIf(Predicate<? super E> filter)`. Dicho método elimina de la colección aquellos elementos que cumplan el *Predicate*:

```
package tema12_ProgramacionFuncional;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateComposition {

    public void show() {

        List<Integer> list = new ArrayList<>(List.of(3, 2, 6, 1, 5,
4));
        Predicate<Integer> esPar = n -> n % 2 == 0;
        Predicate<Integer> mayorQue3 = n -> n > 3;
        list.removeIf(esPar.and(mayorQue3));
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new PredicateComposition().show();

    }

}
```

Salida por consola:

```
3
2
1
5
```

La interfaz `Predicate` tiene también un método estático factoría `Predicate.isEqual(Object o)` que retorna el predicado correspondiente a comprobar si un elemento es igual a otro objeto. Internamente simplemente se llamará al método `equals`:

```
package tema12_ProgramacionFuncional;
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateIsEqual {

    public void show() {

        List<Integer> list = new ArrayList<>(List.of(5, 3, 2, 5, 6, 1,
5, 4));
        list.removeIf(Predicate.isEqual(5));
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new PredicateIsEqual().show();

    }

}
```

Salida por consola:

```
3
2
6
1
4
```

10. Function

`Function<T,R>`: su método abstracto es `R apply(T t)`. Veamos un ejemplo de utilización en el método `computeIfAbsent` de los mapas:

```
package tema12_ProgramacionFuncional;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class InterfaceFunction {

    public void show() {

        Function<Integer, Integer> elevarAlCuadrado = n -> n * n;
```

```
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < 10; i++) {
            map.computeIfAbsent(i, elevarAlCuadrado);
        }
        map.forEach((k, v) -> System.out.printf("Clave:%d Valor:%d\n",
k, v));

    }

    public static void main(String[] args) {

        new InterfaceFunction().show();

    }

}
```

Salida por consola:

```
Clave:0 Valor:0
Clave:1 Valor:1
Clave:2 Valor:4
Clave:3 Valor:9
Clave:4 Valor:16
Clave:5 Valor:25
Clave:6 Valor:36
Clave:7 Valor:49
Clave:8 Valor:64
Clave:9 Valor:81
```

La interfaz funcional `Function` posee un método *default* llamado `andThen(afterFunction)` que permite que después de la ejecución del método `apply` de la *function* original, su resultado se pase como valor de entrada del método `apply` del objeto `Function` pasado como argumento, retornando el objeto `Function` correspondiente a la cadena de operaciones de transformación. Veamos un ejemplo:

```
package tema12_ProgramacionFuncional;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class FunctionAndThen {

    public void show() {

        Function<Integer, Integer> elevarAlCuadrado = n -> n * n;
```

```
        Function<Integer, String> aCadena = String::valueOf;
        Map<Integer, String> map = new HashMap<>();
        for (int i = 0; i < 10; i++) {
            map.computeIfAbsent(i, elevarAlCuadrado.andThen(aCadena));
        }
        map.forEach((k, v) -> System.out.printf("Clave:%d Valor:%s\n",
k, v));

    }

    public static void main(String[] args) {

        new FunctionAndThen().show();

    }

}
```

Salida por consola:

```
Clave:0 Valor:0
Clave:1 Valor:1
Clave:2 Valor:4
Clave:3 Valor:9
Clave:4 Valor:16
Clave:5 Valor:25
Clave:6 Valor:36
Clave:7 Valor:49
Clave:8 Valor:64
Clave:9 Valor:81
```

Muy parecido al anterior es el método *default* `compose(beforeFunction)` que primero ejecuta el método `apply` del objeto `Function` recibido y después el método `apply` del objeto `Function` original, es decir, se ejecutan en orden inverso al de `andThen`. Gracias a estos dos métodos, podemos tener una serie de objetos `Function` predefinidos y encadenarlos de la forma que nos interese.

```
package tema12_ProgramacionFuncional;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

public class FunctionCompose {

    public void show() {

        Function<Integer, Integer> elevarAlCuadrado = n -> n * n;
```

```
        Function<Integer, String> aCadena = String::valueOf;
        Map<Integer, String> map = new HashMap<>();
        for (int i = 0; i < 10; i++) {
            map.computeIfAbsent(i, aCadena.compose(elevarAlCuadrado));
        }
        map.forEach((k, v) -> System.out.printf("Clave:%d Valor:%s\n",
k, v));

    }

    public static void main(String[] args) {

        new FunctionCompose().show();

    }

}
```

Salida por consola:

```
Clave:0 Valor:0
Clave:1 Valor:1
Clave:2 Valor:4
Clave:3 Valor:9
Clave:4 Valor:16
Clave:5 Valor:25
Clave:6 Valor:36
Clave:7 Valor:49
Clave:8 Valor:64
Clave:9 Valor:81
```

11. Comparator

La interfaz funcional `Comparator` (comparador) contiene el método abstracto `int compare(T o1, T o2)` que recibe dos valores de tipo `T` y retorna un entero que vale:

- 0 si o1 es igual a o2.
- Menor que 0 si o1 es menor que o2.
- Mayor que 0 si o1 es mayor que o2.

La interfaz `Comparator` ya existía en Java 7 y de hecho no se encuentra en el paquete `java.util.function`, sino directamente en `java.util`. Esta interfaz posee una serie de métodos estáticos factoría que retornan objetos `Comparator` correspondientes a los casos más habituales de comparación:

- `static <T extends Comparable<? super T>> Comparator<T> naturalOrder()`: devuelve un objeto `Comparator` para ordenar por el orden natural. Pero, ¿qué considera Java como orden natural? El orden indicado en la implementación de `Comparable`.

```java
package tema12_ProgramacionFuncional.comparator;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class NaturalOrder {

    public void show() {

        List<Integer> list = Arrays.asList(3, 2, 6, 1, 5, 4);
        list.sort(Comparator.naturalOrder());
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new NaturalOrder().show();

    }

}
```

Salida por consola:

```
1
2
3
4
5
6
```

- `static <T extends Comparable<? super T>>Comparator<T> reverseOrder()`: ordena por el orden natural inverso.

```java
package tema12_ProgramacionFuncional.comparator;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class ReverseOrder {
```

```
    public void show() {

        List<Integer> list = Arrays.asList(3, 2, 6, 1, 5, 4);
        list.sort(Comparator.reverseOrder());
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new ReverseOrder().show();

    }

}
```

Salida por consola:

```
6
5
4
3
2
1
```

- `static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)`: recibe un comparador y retorna un nuevo comparador según el cual los elementos `null` precederán a los que no lo sean, que serán ordenados atendiendo al comparador recibido. También tenemos `nullsLast`, similar al anterior pero los elementos que sean `null` se sitúan al final.

```
package tema12_ProgramacionFuncional.comparator;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class NullsFirst {

    public void show() {

        List<Integer> list = Arrays.asList(3, 2, null, 6, 1, 5, 4,
null);

list.sort(Comparator.nullsFirst(Comparator.naturalOrder()));
        System.out.println("nullsFirst:");
        list.forEach(System.out::println);

list.sort(Comparator.nullsLast(Comparator.naturalOrder()));
```

```
        System.out.println("nullsLast:");
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new NullsFirst().show();

    }

}
```

Salida por consola:

```
nullsFirst:
null
null
1
2
3
4
5
6
nullsLast:
1
2
3
4
5
6
null
null
```

- `static <T, U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)`: recibe una función que debe retornar el valor por el que debe comparar el comparador retornado. Este método es muy útil si queremos ordenar una lista de objetos por un determinado campo.

```
package tema12_ProgramacionFuncional.comparator;

public class Vehicle {

    private String registration;
    private int wheelCount;
    private double speed;
    private String colour;
```

```
    public Vehicle(String registration, int wheelCount, String
colour) {
        this.registration = registration;
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public String getRegistration() {
        return registration;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return "Vehicle [registration=" + registration + ",
wheelCount=" + wheelCount + ", speed=" + speed + ", colour="
                + colour + "]";
    }

}
```

```
package tema12_ProgramacionFuncional.comparator;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
```

```
public class Comparing {

    public void show() {

        List<Vehicle> list = new ArrayList<>();
        list.add(new Vehicle("1705UBG", 4, "blanco"));
        list.add(new Vehicle("9685KMX", 4, "azul"));
        list.add(new Vehicle("1235GTR", 2, "rojo"));
        list.add(new Vehicle("7314QWE", 4, "verde"));
        list.add(new Vehicle("3495JZA", 2, "blanco"));
        list.add(new Vehicle("5930POI", 2, "negro"));
        list.sort(Comparator.comparing(Vehicle::getColour));
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Comparing().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0,
colour=azul]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0,
colour=blanco]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0,
colour=blanco]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0,
colour=negro]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0,
colour=rojo]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0,
colour=verde]
```

Hay versiones de este método estático factoría para cuando el valor por el que se debe ordenar es de un tipo primitivo, como `comparingInt`, `comparingLong` o `comparingDouble`.

La interfaz funcional `Comparator` también incorpora una serie de métodos *default* que nos permiten encadenar comparadores:

- `default Comparator<T> reversed()`: sirve para obtener el orden inverso al del comparador original.

```java
package tema12_ProgramacionFuncional.comparator;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Reversed {

    public void show() {

        List<Vehicle> list = new ArrayList<>();
        list.add(new Vehicle("1705UBG", 4, "blanco"));
        list.add(new Vehicle("9685KMX", 4, "azul"));
        list.add(new Vehicle("1235GTR", 2, "rojo"));
        list.add(new Vehicle("7314QWE", 4, "verde"));
        list.add(new Vehicle("3495JZA", 2, "blanco"));
        list.add(new Vehicle("5930POI", 2, "negro"));

list.sort(Comparator.comparing(Vehicle::getColour).reversed());
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new Reversed().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0,
colour=verde]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0,
colour=rojo]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0,
colour=negro]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0,
colour=blanco]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0,
colour=blanco]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0,
colour=azul]
```

- `default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor)`: recibe una función con la que indicar el valor por el que comparar si con el comparador original los elementos son iguales.

```
package tema12_ProgramacionFuncional.comparator;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class ThenComparingFunction {

    public void show() {

        List<Vehicle> list = new ArrayList<>();
        list.add(new Vehicle("1705UBG", 4, "blanco"));
        list.add(new Vehicle("9685KMX", 4, "azul"));
        list.add(new Vehicle("1235GTR", 2, "rojo"));
        list.add(new Vehicle("7314QWE", 4, "verde"));
        list.add(new Vehicle("3495JZA", 2, "blanco"));
        list.add(new Vehicle("5930POI", 2, "negro"));

list.sort(Comparator.comparing(Vehicle::getColour).thenComparing(V
ehicle::getWheelCount));
        list.forEach(System.out::println);

    }

    public static void main(String[] args) {

        new ThenComparingFunction().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0,
colour=azul]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0,
colour=blanco]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0,
colour=blanco]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0,
colour=negro]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0,
colour=rojo]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0,
colour=verde]
```

Existen versiones específicas de este método para cuando la función retorna un tipo primitivo, como `thenComparingInt`, `thenComparingLong` y `thenComparingDouble`.

12. Optional

La clase `Optional<T>` es un *wrapper* alrededor de un valor que puede estar presente o no. El objetivo de esta clase es servir básicamente como tipo de retorno de aquellos métodos que pueden retornar un valor o no.

Por ejemplo, supongamos que queremos realizar un método que retorne la posición en la que se encuentra un determinado elemento de una lista de enteros. La pregunta que inmediatamente nos haríamos como desarrolladores sería ¿qué debo hacer si el elemento a buscar, que ha sido pasado como argumento del método, no se encuentra en la lista? ¿Debo lanzar una excepción? No puedo retornar `false` porque el método retorna un entero. ¿Debo retornar un valor especial como por ejemplo `-1`? ¿Retorno como valor especial el valor `null`?

Tradicionalmente los desarrolladores han resuelto esta situación de distintas maneras y todas tienen sus inconvenientes. Por un lado, lanzar una excepción parece excesivo porque no es un error del programa y al cliente del método simplemente hay que informarle de alguna manera de que no se ha encontrado el elemento.

Por otra parte, retornar un valor especial tiene el inconveniente de que obliga a que el cliente conozca dicho valor especial y además debe acordarse de comprobar que el valor retornado no es el valor especial si quiere usarlo. En cierta manera estamos dándole al cliente la responsabilidad de la comprobación pero no estamos obligándole a ello. La consecuencia es que si el desarrollador del código cliente olvida realizar la comprobación, estará usando un valor no válido. Este hecho se convierte en más peligroso aún si el valor retornado es `null`, porque si olvida realizar la comprobación y más adelante en el código se trata de acceder a una propiedad del objeto retornado por el método, se produciría una excepción `NullPointerExcepcion`.

Entonces, ¿cuál es la solución? La solución propuesta por Java 8 es que el método no retorne directamente un objeto de la clase `T`, sino un `Optional<T>`. La ventaja de esta solución es que si el código cliente quiere acceder al objeto real debe forzosamente comprobar si el *Optional* tiene un valor presente o no, de manera que pueda extraerse.

Es muy importante resaltar que **no es posible tener un Optional cuyo valor contenido sea null**.

12.1 Construcción

La clase `Optional<T>` es una clase inmutable, por lo que no proporciona métodos *setter* que permitan cambiar el valor que contiene. Además, el proceso de construcción de un *Optional* no se realiza a través de un constructor, sino usando alguno de los métodos estáticos factoría que se indican a continuación:

- `static <T> Optional<T> empty()`: retorna un *Optional* que no contiene valor.
- `static <T> Optional<T> of(T value)`: retorna un *Optional* que contiene el valor pasado como argumento. Si tratamos de pasar el valor `null` a dicho método, se producirá una excepción.
- `static <T> Optional<T> ofNullable(T value)`: retorna un *Optional* que contiene el valor pasado como argumento. A diferencia del método anterior, si tratamos de pasar el valor `null`, el método retorna un *Optional* vacío.

Ejemplo:

```
package tema12_ProgramacionFuncional.optional;

import java.util.Optional;

public class Construction {

    public void show() {

        Optional<Integer> optional1 = Optional.empty(); // Crea un
optional sin valor.
        System.out.println(optional1);
        Optional<Integer> optional2 = Optional.of(1000); // Crea un
optional con valor 1000
        System.out.println(optional2);

        // Retorna un Optional vacío si el valor pasado como argumento
es null:
        Optional<Integer> optional3 =
Optional.ofNullable(methodWhichCanReturnNull(5));
        System.out.println(optional3);
```

```
        Optional<Integer> optional4 =
Optional.ofNullable(methodWhichCanReturnNull(2));
        System.out.println(optional4);

        Optional<Integer> optional5;
        try {
            optional5 = Optional.of(null);
            System.out.println(optional5);
        } catch (NullPointerException e) {
            System.out.println("Se ha lanzado la excepción
NullPointerException");
        }

    }

    public static Integer methodWhichCanReturnNull(int num) {

        if (num >= 5) {
            return num;
        } else {
            return null;
        }

    }

    public static void main(String[] args) {

        new Construction().show();

    }

}
```

Salida por consola:

```
Optional.empty
Optional[1000]
Optional[5]
Optional.empty
Se ha lanzado la excepción NullPointerException
```

12.2 Igualdad

Podemos comparar los valores de dos *Optional* con el método `boolean equals(Object obj)`: compara los valores de los dos *Optional*. Dos *Optional* vacíos de distinto tipo son considerados iguales.

Ejemplo:

```java
package tema12_ProgramacionFuncional.optional;

import java.util.Optional;

public class Equality {

    public void show() {

        Optional<Integer> optional1 = Optional.of(1000); //Crea un
optional con valor 1000
        Optional<Integer> optional2 = Optional.of(1000);
        Optional<Integer> optional3 = Optional.empty();
        Optional<String> optional4 = Optional.empty();

        System.out.println(optional1.equals(optional2)); //Comprueba
si tienen el mismo valor: true
        System.out.println(optional1 == optional2); //Comprueba si son
el mismo objeto: false
        System.out.println(optional3.equals(optional4));//Dos Optional
vacíos de distinto tipo son considerados iguales: true
        System.out.println(optional3.equals(null));//False

    }

    public static void main(String[] args) {

        new Equality().show();

    }

}
```

12.3 Procesamiento

Supongamos que queremos usar el método estático `Collections.max(collection)` que retorna el valor máximo contenido en una colección:

```java
// Suponiendo que list es una colección de enteros.
Integer maximo = Collections.max(list);
```

Pero ¿qué ocurre si la colección está vacía? En este caso el método `max()` lanzará la excepción `NoSuchElementException`. ¿Cuál es el problema? Que el cliente debe mirar la documentación para enterarse de ello y capturar la excepción o de lo contrario se producirá un error en tiempo de ejecución. Está obligado a darse cuenta de que debe hacer:

```java
try {
        Integer maximo = Collections.max(list);
    // ...
} catch (NoSuchElementException e) {
    // ...
}
```

¿Y si existiera una forma de indicar que el valor de retorno de una función es opcional? Pues bien, Java 8 introdujo para este problema la clase `Optional<T>`.

La clase `Optional<T>` es una clase parametrizada que representa la abstracción de un valor de retorno opcional. Así, podríamos codificar nuestro método anterior haciendo que retorne un `Optional<T>` en vez de lanzar una excepción en el caso de que la colección esté vacía:

```java
public static <T extends Comparable<T>> Optional<T> max(Collection<?
extends T> coll) {
        try {
            return Optional.of(Collections.max(coll));
    } catch (NoSuchElementException e) {
            return Optional.empty();
    }
}
```

En ese caso, cuando el cliente llame a este método estará obligado a tratar el hecho de que puede que no se retorne valor. No tiene que mirar la documentación ni recordar hacer ningún tipo de comprobación porque el tipo de retorno es `Optional<T>`, de manera que si de verdad quiere obtener el valor, va a tener que comprobar si el *Optional* tiene o no valor.

```java
Optional<Integer> maximoOpt = max(list);
```

Veamos más métodos de *Optional*:

- `T get()`: si un valor está presente, devuelve el valor, de lo contrario lanza la excepción *NoSuchElementException*.
- `boolean isPresent()`: si un valor está presente, devuelve *true*, en caso contrario *false*.

```java
package tema12_ProgramacionFuncional.optional;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
```

```
import java.util.NoSuchElementException;
import java.util.Optional;

public class GetAndIsPresent {

    public void show() {

        List<Integer> list1 = List.of(3, 2, 6, 1, 5, 4);
        List<Integer> list2 = new ArrayList<>();
        Optional<Integer> maximumOptional1 = max(list1);
        Optional<Integer> maximumOptional2 = max(list2);
        Integer maximum1 = 0, maximum2;

        if (maximumOptional1.isPresent()) {
            maximum1 = maximumOptional1.get();
        }
        System.out.println(maximumOptional1);//Optional[6]
        System.out.println(maximum1);//6

        maximum2 = maximumOptional2.isPresent() ?
maximumOptional2.get() : 0;
        System.out.println(maximumOptional2);//Optional.empty
        System.out.println(maximum2);//0

    }

    public static <T extends Comparable<T>> Optional<T>
max(Collection<? extends T> coll)     {
        try {
            return Optional.of(Collections.max(coll));
        } catch (NoSuchElementException e) {
            return Optional.empty();
        }
    }

    public static void main(String[] args) {

        new GetAndIsPresent().show();

    }

}
```

- `T orElse(T other)`: si un valor está presente, devuelve el valor, de lo contrario devuelve el valor suministrado como argumento.
- `T orElseGet(Supplier<? extends T> supplier)`: si hay un valor, devuelve el valor, en caso contrario devuelve el resultado producido por la función suministradora.

- `T orElseThrow()`: si un valor está presente, devuelve el valor, de lo contrario lanza la excepción *NoSuchElementException*.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X`: si hay un valor, devuelve el valor, en caso contrario lanza una excepción producida por la función suministradora.

```
package tema12_ProgramacionFuncional.optional;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Optional;
import java.util.Random;

public class OrElse {

    public void show() {

        List<Integer> list1 = List.of(3, 2, 6, 1, 5, 4);
        List<Integer> list2 = new ArrayList<>();
        Optional<Integer> maximumOptional1 = max(list1);
        Optional<Integer> maximumOptional2 = max(list2);
        Integer maximum1, maximum2;

        maximum1 = maximumOptional1.orElse(0);
        maximum2 = maximumOptional2.orElse(0);
        System.out.println(maximum1);//6
        System.out.println(maximum2);//0

        maximum1 = maximumOptional1.orElseGet(() ->
obtainRandomNumber());
        maximum2 = maximumOptional2.orElseGet(() ->
obtainRandomNumber());
        System.out.println(maximum1);//6
        System.out.println(maximum2);//Número aleatorio entre 1 y
10

        maximum1 = maximumOptional1.orElseThrow();
        System.out.println(maximum1);//6
        try {
            maximum2 = maximumOptional2.orElseThrow();
        } catch (NoSuchElementException e) {
            System.out.println("Valor no presente");
        }
```

```
            maximum1 =
    maximumOptional1.orElseThrow(IllegalStateException::new);
            System.out.println(maximum1);//6
            try {
                maximum2 =
    maximumOptional2.orElseThrow(IllegalStateException::new);
            } catch (IllegalStateException e) {
                System.out.println("Valor no presente");
            }

        }

        public Integer obtainRandomNumber() {
            return new Random().nextInt(10) + 1;//Devuelve un número
    aleatorio entre 1 y 10
        }

        public static <T extends Comparable<T>> Optional<T>
    max(Collection<? extends T> coll)     {
            try {
                return Optional.of(Collections.max(coll));
            } catch (NoSuchElementException e) {
                return Optional.empty();
            }
        }

        public static void main(String[] args) {

            new OrElse().show();

        }

    }
```

- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)`: se introdujo en Java 9 para encadenar *Optionals*. Si el *Optional* original contiene un valor, el método retorna un nuevo *Optional* con dicho valor. Si el *Optional* original no contiene valor, el método retornará el *Optional* producido por la función suministradora.
- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`: sirve para transformar el valor contenido en un *Optional*. Si el *Optional* original está vacío, devuelve un nuevo *Optional* vacío y si tiene valor, devolverá un *Optional* que contendrá como valor el resultado de aplicar la función de transformación al valor contenido en el *Optional* original. Si dicha función devuelve un resultado nulo, entonces se devuelve un *Optional* vacío. Ejemplo:

- `Optional<T> filter(Predicate<? super T> predicate)`: retorna un nuevo *Optional* que estará vacío si el *Optional* original estaba vacío o si no se cumple el predicado recibido como argumento (su método `test()` retorna `false`). Si el *Optional* original contenía un valor y dicho valor cumple con el predicado, el nuevo *Optional* retornado contendrá el valor del *Optional* original. Ejemplo:

```
package tema12_ProgramacionFuncional.optional;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Optional;
import java.util.Random;

public class OrMapFilter {

    public void show() {

        List<Integer> list1 = List.of(3, 2, 6, 1, 5, 4);
        List<Integer> list2 = new ArrayList<>();
        Optional<Integer> maximumOptional1 = max(list1);
        Optional<Integer> maximumOptional2 = max(list2);
        Optional<Integer> maximumOptional3, maximumOptional4,
maximumOptional5;

        maximumOptional3 = maximumOptional1.or(() ->
Optional.of(obtainRandomNumber()));
        maximumOptional4 = maximumOptional2.or(() ->
Optional.of(obtainRandomNumber()));
        System.out.println(maximumOptional3);//Optional[6]
        System.out.println(maximumOptional4);//Optional de un
número aleatorio

        maximumOptional3 = maximumOptional1.map(n -> n * 2);
        maximumOptional4 = maximumOptional2.map(n -> n * 2);
        System.out.println(maximumOptional3);//Optional[12]
        System.out.println(maximumOptional4);//Optional.empty

        maximumOptional3 = maximumOptional1.filter(n -> n % 2 ==
0);
        maximumOptional4 = maximumOptional1.filter(n -> n % 2 !=
0);
        maximumOptional5 = maximumOptional2.filter(n -> n % 2 ==
0);
        System.out.println(maximumOptional3);//Optional[6]
        System.out.println(maximumOptional4);//Optional.empty
        System.out.println(maximumOptional5);//Optional.empty
```

```
    }

    public Integer obtainRandomNumber() {
        return new Random().nextInt(10) + 1;//Devuelve un número
aleatorio entre 1 y 10
    }

    public static <T extends Comparable<T>> Optional<T>
max(Collection<? extends T> coll)     {
        try {
            return Optional.of(Collections.max(coll));
        } catch (NoSuchElementException e) {
            return Optional.empty();
        }
    }

    public static void main(String[] args) {

        new OrMapFilter().show();

    }

}
```

- `void ifPresent(Consumer<? super T> action)`: permite consumir (usar) directamente el valor contenido en un *Optional* si es que éste contiene un valor. Si el *Optional* está vacío, no hace nada.
- `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`: se incorporó este método en Java 9. Ejecuta el consumidor pasado como argumento si el *Optional* posee un valor o ejecuta el *Runnable* pasado como argumento si el *Optional* no contiene ningún valor. Ejemplo:

```
package tema12_ProgramacionFuncional.optional;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.NoSuchElementException;
import java.util.Optional;

public class IfPresent {

    public void show() {

        List<Integer> list1 = List.of(3, 2, 6, 1, 5, 4);
        List<Integer> list2 = new ArrayList<>();
```

```
        Optional<Integer> maximumOptional1 = max(list1);
        Optional<Integer> maximumOptional2 = max(list2);
        maximumOptional1.ifPresent(System.out::println);//6
        maximumOptional2.ifPresent(System.out::println);//No hace
nada

        maximumOptional1.ifPresentOrElse(System.out::println,
                () -> System.out.println("No hay máximo porque la
lista está vacía"));//6
        maximumOptional2.ifPresentOrElse(System.out::println,
                () -> System.out.println("No hay máximo porque la
lista está vacía"));//No hay máximo porque la lista está vacía

    }

    public static <T extends Comparable<T>> Optional<T>
max(Collection<? extends T> coll)     {
        try {
            return Optional.of(Collections.max(coll));
        } catch (NoSuchElementException e) {
            return Optional.empty();
        }
    }

    public static void main(String[] args) {

        new IfPresent().show();

    }

}
```

Desafortunadamente, **no podemos garantizar que la referencia al objeto *Optional* en sí no sea `null`**. De hecho, en todos los ejemplos anteriores estamos confiando en que el método `max()` nunca va a retornar un `null`. Otros lenguajes de programación más modernos, como Kotlin o Swift, incorporan el concepto de *Optional* al sistema de tipos, de manera que existirán dos tipos distintos, uno que incorpora la posibilidad de tener el valor `null` y otro tipo que no lo permite. Por ejemplo, en Kotlin existe el tipo `String`, que no puede contener el valor `null` y el tipo `String?`, que sí puede contenerlo. Al establecer el tipo de retorno de una función tendremos que decidir si el tipo es el que puede incluir `null` o el que no. La ventaja de esta técnica es que si el tipo de retorno del método es `String?`, el cliente deberá comprobar si realmente se ha retornado un valor o no, pero siempre podrá realizar la comprobación. El problema de Java es que al encapsular un valor en un objeto *Optional*, el propio objeto *Optional* podría ser `null` y deberíamos fiarnos de que eso nunca puede suceder o realizar la comprobación cada vez. Evidentemente, cualquier desarrollador que realiza un método

que retorna un *Optional* debería asegurarse y documentar que dicho método nunca retornará `null`, sino siempre un *Optional*.

12.4 Dónde no usar Optional

Debemos tener en cuenta que la clase `Optional<T>` ha sido diseñada específicamente para ser usada como tipo de retorno de los métodos. No se recomienda su uso en los siguientes casos:

- No se recomienda usar *Optional* como tipo de los atributos de una clase.
- No se recomienda usar *Optional* como tipo de los parámetros de un método porque ensucian mucho el código y realmente no hacen que el parámetro sea opcional.
- No se recomienda usar *Optional* como tipo de una colección, como por ejemplo en una lista.