Interfaces funcionales

¿Dónde se usan las expresiones lambda?

- **Interface funcional**. Es una interface que especifica exactamente un solo método abstracto.

```
public interface Adder {
    int add(int a, int b);
}
```

- Las expresiones lambda permiten implementar el método abstracto en una línea y *tratar la expresión completa como una instancia de la interface funcional*.

Principales interfaces funcionales

- El paquete java.util.function define varias interfaces funcionales.
- Las 6 interfaces funcionales genéricas básicas son: BinaryFunction, Consumer, Function, Predicate, Supplier y UnaryOperator.

Interfaces funcionales genéricas básicas

Interface funcional	Descripción
BinaryOperator<T>	Contiene el método apply que recibe dos argumentos T, hace una operación con ellos y devuelve un valor T.
Consumer<T>	Contiene el método accept que recibe un argumento T, hace alguna tarea con el y regresa void.
Function<T, R>	Contiene el método apply que recibe un argumento T, hace alguna tarea con el y regresa un valor de tipo R.
Predicate<T>	Contiene el método test que recibe un argumento T, checa una condición y regresa un boolean.
Supplier<T>	Contiene el método get que no recibe argumentos y devuelve un valor de tipo T. Se puede usar para crear objetos o una colección.
UnaryOperator<T>	Contiene el método apply que recibe un argumento T y devuelve un valor de tipo T. Es una especialización de Function para cuando T y R son el mismo tipo.

Interface BinaryOperator<T>

- Define un método abstracto apply.
- apply recibe dos argumentos genéricos de tipo T y regresa un objeto de tipo T.
- Ejemplo:

```
BinaryOperator<Integer> f = (x1, x2) -> x1 + x2;
Integer result = f.apply(2, 3);
System.out.println(result);        // imprime 5
```

- BinaryOperator es útil para definir métodos que apliquen distintas funciones a sus operandos.

Ejemplo: método op

```
// Aplica el operador f a dos listas
public static <T> List<T> op(List<T> list1, List<T> list2, BinaryOperator<T> f)
{
    List<T> result = new ArrayList<>();
    if (list1.size() != list2.size()) throw new IllegalArgumentException("Las dos listas deben tener el mismo tamaño");
    for (int i = 0; i < list1.size(); i++) {
        result.add(f.apply(list1.get(i), list2.get(i)));
    }
    return result;
}
```

Ejemplo: uso de op

```
List<Integer> list1 = Arrays.asList(10, 8, 2, 6, 7, 5);
List<Integer> list2 = Arrays.asList(-4, 4, 7, 2, 8, -5);
List<Integer> list3 = op(list1, list2, (x1, x2) -> x1 + x2);
list3.forEach(p -> System.out.println(p));  // 6, 12, 9, 8, 15, 0
List<Integer> list4 = op(list1, list2, (x1, x2) -> x1 * x2);
list4.forEach(p -> System.out.println(p));  // -40, 32, 14, 12, 56, 25
```

Ejemplo: método reduce

```
// Reduce una lista a un número
public static <T> T reduce(List<T> list, T init, BinaryOperator<T>
accumulator)
{
    T result = init;
    for (T t : list) {
        result = accumulator.apply(result, t);
    }
    return result;
}
```

Ejemplo: uso de reduce

```
List<Integer> list = Arrays.asList(4, 4, 7, 2, 8, 5);
// suma de los elementos
Integer s1 = reduce(list, 0, (x1, x2) -> x1 + x2);
System.out.println("s1: " + s1);          // Imprime 30
// suma de los elementos al cuadrado
Integer s2 = reduce(list, 0, (x1, x2) -> x1 + x2 * x2);
System.out.println("s2: " + s2);          // Imprime 174
```

Interface Consumer<T>

- Define un método abstracto accept.
- accept recibe un objeto genérico de tipo T y no regresa nada (void).
- Se usa cuando se necesita accesar un objeto de tipo T y hacer una operación sobre el.
- Por ejemplo, el método forEach de ArrayList:

`void`	`forEach(Consumer<? super E> action)`	Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.

```
people.forEach(p -> System.out.println(p));
```

Ejemplo

- Consumer que multiplica cada elemento de una lista por dos.

```
Consumer <List<Integer>> porDos = lista -> {
    for (int i = 0; i < lista.size(); i++)
        lista.set(i, 2 * lista.get(i));
};
```

- Uso:

```
List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
porDos.accept(lista);
lista.forEach(p -> System.out.println(p));   // 2, 4, 6, …, 18, 20
```

Interface Function<T, R>

- Define un método abstracto apply.
- apply recibe un objeto genérico de tipo T y regresa un objeto de tipo genérico R.
- Se usa cuando se necesita mapear un objeto de entrada a otro de salida.

Ejemplo: método map

- Método que mapea una lista de objetos tipo T a una lista de objetos de tipo R.

```
public static <T, R> List<R> map(List<T> list, Function<T, R> f)
{
    List<R> result = new ArrayList<>();
    for(T t: list) {
        result.add(f.apply(t));
    }
    return result;
}
```

Ejemplo: uso de map

- Mapea una lista de strings a una lista de enteros con el tamaño:

```
List<Integer> lista = map(Arrays.asList("lambdas", "in", "action"),
                                        s -> s.length());
lista.forEach(p -> System.out.println(p));    // 7, 2, 6
```

- Mapea una lista de ángulos enteros a una lista de dobles con los cosenos de los ángulos.

```
List<Integer> angles = Arrays.asList(0, 15, 30, 45, 60, 75, 90);
List<Double> cos = map(angles, x -> Math.cos(Math.toRadians(x)));
cos.forEach(p -> System.out.println(p));  // cos(0), cos(15), …, cos(90)
```

Interface Predicate<T>

- Define un método abstracto llamado test.
- test recibe un objeto genérico de tipo T y regresa un booleano.
- Sirve para representar expresiones booleanas que usan un objeto de tipo T.

Ejemplo

- Hacer un método `filter` que reciba una lista de objetos y una condición y devuelva en otra lista todos los objetos que cumplan con la condición.

Ejemplo: método filter

```
public static <T> List<T> filter(List<T> list, Predicate<T> p)
{
    List<T> results = new ArrayList<>();
    for (T t : list) {
        if (p.test(t)) {
            results.add(t);
        }
    }
    return results;
}
```

Ejemplo: uso de filter

- Con una lista de enteros:

```
List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> pares = filter(lista, x -> (x % 2) == 0);
pares.forEach(p -> System.out.println(p));    // 2, 4, 6, 8, 10
```

- Con una lista de personas:

```
List<Person> adultos = filter(people, p -> p.getEdad() >= 18);
adultos.forEach(p -> System.out.println(p));   // imprime los adultos
```

Interface Supplier<T>

- Define un método abstracto get.
- get no recibe argumentos y devuelve un valor de tipo T.
- Supplier se puede usar para crear objetos.
- Por ejemplo, imprimir la hora con cierto formato:

```
private static final DateTimeFormatter timeFormat =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
Supplier<String> time = () -> timeFormat.format(LocalDateTime.now());
System.out.println(time.get());          // imprime p.e. 2021-08-21 17:27:11
```

Ejemplo

- Hacer un método generate(n, Supplier<T>) que regrese una lista de n números aleatorios de tipo T.
- El tipo T puede ser cualquier tipo numérico (int, long, float, double, etc.)

Ejemplo: método generate

```
public static <T> List <T> generate(int limit, Supplier<T> supplier)
{
    List<T> result = new ArrayList<>();
    for (int i = 0; i < limit; i++) {
        result.add(supplier.get());
    }
    return result;
}
```

Ejemplo: uso del método generate

```
Random random = new Random();
// 5 números aleatorios en el rango [0, 100)
List<Integer> random1 = generate(5, () -> random.nextInt(100));
random1.forEach(p -> System.out.println(p));
// 7 números aleatorios en el rango [0, 1)
List<Double> random2 = generate(7, () -> random.nextDouble());
random2.forEach(p -> System.out.println(p));
```

Interface UnaryOperator<T>

- Define un método abstracto apply.
- apply recibe un argumento T y devuelve un valor de tipo T.
- Es una especialización de Function<T, R> para cuando T y R son el mismo tipo.

```
Function<Integer, Integer> f = x -> x * 2;
Integer result = f.apply(2);
System.out.println(result);        // 4
UnaryOperator<Integer> g = x -> x * 2;
Integer result2 = g.apply(2);
System.out.println(result2);       // 4
```

Ejemplo

- Hacer un método aplica que reciba una lista y una función y aplique la función a cada elemento de la lista regresando una lista nueva con los resultados.

Ejemplo: método aplica

```
public static <T> List<T> aplica(List<T> list, UnaryOperator<T> f)
{
    List<T> result = new ArrayList<>();
    for (T item : list) {
        result.add(f.apply(item));
    }
    return result;
}
```

Ejemplo: uso de aplica

```
// Lista de dobles
List<Double> listNum = Arrays.asList(7.2, 5.1, 16.0, 15.4, 10.0);
List<Double> listSqrt = aplica(listNum, x -> Math.sqrt(x));
listSqrt.forEach(p -> System.out.println(p));  // sqrt(7.2) … sqrt(10)
// Lista de strings
List<String> listStr1 = Arrays.asList("ESTO", "es", "una", "PRUEBA");
List<String> listStr2 = aplica(listStr1, s -> s.toLowerCase());
listStr2.forEach(s -> System.out.println(s)); // "esto" "es" "una" "prueba"
```

Especializaciones primitivas

- Las interfaces funcionales Predicate<T>, Consumer<T> y Function<T, R> son genéricas.
- Hay interfaces funcionales especializadas para tipos primitivos.
- Son más eficientes que las interfaces funcionales genéricas.

Especializaciones primitivas

Interface funcional	Genérica	Especializadas
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>

Especializaciones primitivas

Interface funcional	Genérica	Especializadas
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<T, U>	(T, U) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer, ObjLongConsumer, ObjDoubleConsumer
BiFunction<T, U, R>	(T, U) –> R	ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction

Resumen

Use case	Example of lambda	Matching functional interface
A boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Apple(10)`	`Supplier<Apple>`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`	`Consumer<Apple>`
Select/extract from an object	`(String s) -> s.length()`	`Function<String, Integer>` or `ToIntFunction<String>`
Combine two values	`(int a, int b) -> a * b`	`IntBinaryOperator`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`	`Comparator<Apple>` or `BiFunction<Apple, Apple, Integer>` or `ToIntBiFunction<Apple, Apple>`