

---

# Expresiones lambda

---

## Contenido

Antes de empezar.....	1
Sintaxis de las expresiones lambda.....	2
Los parámetros .....	3
El cuerpo.....	3
Interfaces funcionales y expresiones lambda .....	3
Interfaz Comparator y expresiones lambda .....	5
Algunas interfaces funcionales de la API de Java.....	8
Predicate<T> .....	8
Function<T, R> .....	9
Consumer<T> .....	10
Supplier<T> .....	11
Referencias a métodos .....	11
Referencias.....	12

## Antes de empezar

---

Vamos a partir de los siguientes ejemplos: tenemos unos métodos que hacen lo siguiente:

1. Comprueba si el parámetro recibido es un número impar.

```
public boolean esImpar(int n) {  
    return n % 2 != 0;  
}
```

2. Comprueba si el parámetro es la letra 'y' (minúscula).

```
public boolean esy(Character c) {  
    return c == 'y';  
}
```

3. Devuelve la suma de dos números.

```
public double sumar(double x, double y) {  
    return x + y;  
}
```

4. Devuelve la suma de los cuadrados de dos números.

```
public int sumarCuadrados(int a, int b) {  
    return a * a + b * b;  
}
```

5. Devuelve el número 42.

```
public int devolver42() {  
    return 42;  
}
```

6. Devuelve el número pi.

```
public double devolverPI() {  
    return Math.PI;  
}
```

7. Imprime una cadena de texto.

```
public void imprimirCadena(String s) {  
    System.out.println(s);  
}
```

8. Imprime "Hola, mundo".

```
public void imprimirHolaMundo {  
    System.out.println("Hola, mundo.");  
}
```

## Sintaxis de las expresiones lambda.

---

Pues bien, a partir de Java 8, todos estos métodos pueden expresarse de manera más compacta a través de expresiones lambda.

Una expresión lambda representa una función y se compone de:

- un conjunto (que puede estar vacío) de parámetros que recibe,
- un operador lambda (->)
- y un cuerpo de la función.

Los anteriores métodos, refactorizados a expresiones lambda, pueden expresarse así:

```
n -> n % 2 != 0; // 1
(Character c) -> c == 'y'; // 2
(x, y) -> x + y; // 3
(int a, int b) -> a * a + b * b; // 4
() -> 42 // 5
() -> {return 3.14}; // 6
(String s) -> { System.out.println(s); }; // 7
() -> { System.out.println(";Hola mundo!"); }; // 8
```

## Los parámetros

---

- Una expresión lambda puede recibir cero (ejemplos 5, 6, 8), uno (ejemplos 1, 2, 7) o más parámetros (ejemplos 3, 4).
- El tipo de los parámetros se puede declarar explícitamente (ejemplos 2, 4, 7) o se puede inferir del contexto (ejemplos 1, 3).
- Los parámetros van entre paréntesis y separados por comas.
- Los paréntesis vacíos se utilizan para indicar que la expresión no recibe parámetros (ejemplos 5, 6, 8).
- Cuando hay un solo parámetro, si se puede inferir su tipo del contexto, no es obligatorio usar paréntesis (ejemplo 1).

## El cuerpo

---

- El cuerpo de la expresión lambda puede contener cero, una (todos los ejemplos anteriores) o más instrucciones.
- Cuando hay una sola instrucción, los corchetes no son obligatorios (cualquiera de los ejemplos anteriores).

Un ejemplo de expresión lambda en la que el cuerpo tiene más de una instrucción puede ser este:

```
(a, b) -> {
    int suma = a + b;
    int resta = a - b;
    return new int[] {suma, resta};
};
```

Como ejercicio, trata de “traducir” la expresión anterior a un método (estático o de instancia, aquí es irrelevante):

## Interfaces funcionales y expresiones lambda

---

Una interfaz funcional es aquella que contiene un método abstracto.

Recordemos también que una interfaz no se puede instanciar, y que cuando lo intentamos, nos crea un código muy raro, que resulta ser una clase anónima.

```

1 public interface HolaMundo {
2     void imprimeHolaMundo();
3 }
4
1 public class Clase {
2     public static void main(String[] args) {
3         HolaMundo f = new HolaMundo() {
4             @Override
5             public void imprimeHolaMundo() {
6                 System.out.println("Hola, mundo");
7             }
8         };
9
10        f.imprimeHolaMundo();
11    }
12 }

```

A partir de Java 8, estas interfaces funcionales se pueden implementar mediante expresiones lambda y referencias a métodos. La expresión lambda viene a sustituir a la clase anónima que se genera al tratar de instanciar una interfaz.

Así, si sustituimos la clase anónima en el código anterior por la expresión lambda correspondiente (el ejemplo 8), nos queda un código mucho más reducido.

```

1 public interface HolaMundo {
2     void imprimeHolaMundo();
3 }
4
1 public class Clase {
2     public static void main(String[] args) {
3         HolaMundo f = () -> System.out.println("¡Hola, mundo!");
4
5         f.imprimeHolaMundo();
6     }
7 }

```

En resumen: **una expresión lambda puede ser utilizada en cualquier parte en donde el tipo declarado sea una interfaz funcional. La expresión lambda proporciona la implementación del método abstracto.**

Como ejercicio, trata de hacer lo mismo con los otros 7 ejemplos.

```

1 public class EjemplosLambda {
2     public static void main(String[] args) {
3         ValidarImpar ej1 = n -> n % 2 != 0;
4         boolean resultado1 = ej1.esImpar(21); // resultado1 = true
5         Validar ej2 = c -> c == 'y';
6         boolean resultado2 = ej2.esY('y'); // resultado2 = true
7         Suma ej3 = (x, y) -> x + y;
8         double resultado3 = ej3.sumar(6, 4); // resultado3 = 10.0
9         SumaCuadrados ej4 = (a, b) -> a * a + b * b;
10        double resultado4 = ej4.sumarCuadrados(5, 2); // resultado4 = 29.0
11        Devuelve42 ej5 = () -> 42;
12        int resultado5 = ej5.devolver42(); // resultado5 = 42
13        DevolverPI ej6 = () -> Math.PI;
14        double resultado6 = ej6.devuelvePI(); // resultado6 = 3.141592...
15        ImprimirCadena ej7 = (s) -> System.out.println(s);
16        ej7.imprimir("Lambda"); // imprime "Lambda"
17        ImprimirHolaMundo ej8 = () -> System.out.println("Hola, mundo");
18        ej8.imprimir(); // imprime "Hola, mundo"
19    }
20 }

```

## Interfaz Comparator y expresiones lambda

---

Vamos a tomar como ejemplo la interfaz Comparator, que ya conocemos, y vamos a ver cómo las expresiones lambda (una vez que se saben utilizar) facilitan y reducen enormemente la escritura de código Java.

Recordemos que la interfaz Comparator es una interfaz funcional porque solamente tiene el método abstracto `compare(Object o1, Object o2)` para ordenar objetos de una clase según diversos criterios.

Para ello vamos a recordar lo que tenemos que hacer si queremos ordenar una colección por un atributo concreto:

- Tenemos la clase Socio con tres atributos (idSocio, nombre y fNacimiento):

```
import java.time.LocalDate;

public class Socio {
    int idSocio;
    String nombre;
    LocalDate fNacimiento;

    public Socio(int idSocio, String nombre, LocalDate fNacimiento) {
        this.idSocio = idSocio;
        this.nombre = nombre;
        this.fNacimiento = fNacimiento;
    }

    @Override
    public String toString() {
        return "Socio nº " + idSocio + ": " + nombre + " (" +
fNacimiento + ")";
    }
}
```

- Si queremos ordenar una colección de socios por fecha de nacimiento, tenemos que crear una clase que implemente la interfaz Comparator:

```
import java.util.Comparator;

public class ComparaFNacimiento implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        // lo primero es convertir los objetos a Socio
        Socio socio1 = (Socio) o1;
        Socio socio2 = (Socio) o2;
        // si socio1 nace antes que socio2 -> -1
        // si es al revés -> 1
        // si son iguales devuelve 0
        if (socio1.fNacimiento.isBefore(socio2.fNacimiento)) {
            return -1;
        } else if (socio1.fNacimiento.isAfter(socio2.fNacimiento)) {
            return 1;
        }
    }
}
```

```

    } else {
        return 0;
    }
}
}

```

- En el main tenemos un ArrayList de socios, que inicialmente se ordenan por orden de entrada (cada vez que añadimos un socio se queda al final del ArrayList).
  - Si queremos ordenar por fecha de nacimiento, tenemos que instanciar la clase que implementa Comparator y pasar este objeto al método sort() del ArrayList.

```

ComparaFNacimiento comparaFNacimiento = new ComparaFNacimiento();
socios.sort(comparaFNacimiento);

```

- Si queremos ordenar por fecha de nacimiento, pero en orden inverso, debemos utilizar el método reversed() de Comparator.

```

Comparator comparaFNacimientoDesc = comparaFNacimiento.reversed();
socios.sort(comparaFNacimientoDesc);

```

Todo esto quedaría así:

```

public class Main {
    public static void main(String[] args) {
        Socio socio1 = new Socio(3, "Valentin", LocalDate.of(2001, 2, 8));
        Socio socio2 = new Socio(2, "Claudia", LocalDate.of(2002, 10, 16));
        Socio socio3 = new Socio(1, "Javier", LocalDate.of(1996, 10, 14));
        Socio socio4 = new Socio(4, "Victor", LocalDate.of(1991, 5, 10));

        ArrayList<Socio> socios = new ArrayList<>();
        socios.add(socio1);
        socios.add(socio2);
        socios.add(socio3);
        socios.add(socio4);

        System.out.println("Según entran en el arraylist:");
        imprimirSocios(socios);

        // ordenamos por fecha de nacimiento
        // instanciamos la clase comparadora
        ComparaFNacimiento comparaFNacimiento = new ComparaFNacimiento();
        // y ahora ya podemos ordenar pasando ese objeto al método sort
        socios.sort(comparaFNacimiento);
        System.out.println("\nOrdenados por fecha de nacimiento");
        imprimirSocios(socios);
        // ordenamos por fecha de nacimiento, al revés
        Comparator comparaFNacimientoDesc = comparaFNacimiento.reversed();
        socios.sort(comparaFNacimientoDesc);
        System.out.println("\nOrdenados por fecha de nacimiento al revés");
        imprimirSocios(socios);
    }

    static void imprimirSocios(ArrayList<Socio> socios) {
        for (Socio elemento: socios) {
            System.out.println(elemento);
        }
    }
}

```

```

    }
}

```

Hasta aquí, nada nuevo. Vemos que para hacer algo tan sencillo como ordenar tenemos que crear una clase adicional para cada criterio de ordenación, y eso es mucho código. Las expresiones lambda simplifican todo esto.

Vamos a aplicar la misma lógica que con los ejemplos del principio al uso de la interfaz `Comparator`.

Para empezar, podríamos, en vez de crear una clase nueva para cada comparador, utilizar clases anónimas instanciando la interfaz `Comparator` directamente en el main.

Vamos a crear una clase `MainClaseAnonima` para ver cómo se haría esto (se supone que no tenemos ninguna de las clases que hemos creado anteriormente, sólo la clase `Socio`, la interfaz `Comparator` disponible en Java SE y el main con el `ArrayList` de personas):

```

Comparator<Socio> comparaFNac = new Comparator<Socio>() {
    @Override
    public int compare(Socio o1, Socio o2) {
        // lo primero es convertir los objetos a Socio
        Socio socio1 = (Socio) o1;
        Socio socio2 = (Socio) o2;
        // implementamos la lógica
        if (socio1.fNacimiento.isBefore(socio2.fNacimiento)) {
            return -1;
        } else if (socio1.fNacimiento.isAfter(socio2.fNacimiento)) {
            return 1;
        } else {
            return 0;
        }
    }
};
socios.sort(comparaFNac);

```

Con esto ya nos hemos ahorrado la clase que implementa la interfaz `Comparator`.

*Ejercicio: haz lo mismo para los otros dos atributos de Socio.*

¿Podemos reducirlo más? Pues sí, sustituyendo la clase anónima por su correspondiente expresión lambda. Hacemos otra clase `Main` (por ejemplo, `MainLambda`) con el `ArrayList` de socios y en vez de lo anterior escribimos:

```

Comparator comparaFNac = ((Object o1, Object o2) -> {
    Socio s1 = (Socio) o1;
    Socio s2 = (Socio) o2;
    if (socio1.fNacimiento.isBefore(socio2.fNacimiento)) {
        return -1;
    } else if (socio1.fNacimiento.isAfter(socio2.fNacimiento)) {
        return 1;
    } else {
        return 0;
    }
});

```

```
socios.sort(comparaFNac);
```

Más reducido: metemos todo lo que está detrás del = dentro del método sort():

```
socios.sort(((Object o1, Object o2) -> {  
    Socio s1 = (Socio) o1;  
    Socio s2 = (Socio) o2;  
    if (socio1.fNacimiento.isBefore(socio2.fNacimiento)) {  
        return -1;  
    } else if (socio1.fNacimiento.isAfter(socio2.fNacimiento)) {  
        return 1;  
    } else {  
        return 0;  
    }  
}));
```

Más reducido: probamos si se puede aplicar inferencia de tipos y resulta que funciona:

```
socios.sort((s1, s2) -> {  
    if (socio1.fNacimiento.isBefore(socio2.fNacimiento)) {  
        return -1;  
    } else if (socio1.fNacimiento.isAfter(socio2.fNacimiento)) {  
        return 1;  
    } else {  
        return 0;  
    }  
});
```

Como el cuerpo tiene más de una línea no podemos quitar las llaves, y como hay más de un return no podemos quitar esta palabra.

*Ejercicio: prueba a hacer lo mismo con los otros dos atributos de Socio.*

## Algunas interfaces funcionales de la API de Java

---

Hay una serie de interfaces funcionales que, como Comparator, corresponden a operaciones frecuentes en las tareas del programador. Además, nos vendrán al pelo cuando más adelante trabajemos con Stream.

### Predicate<T>

---

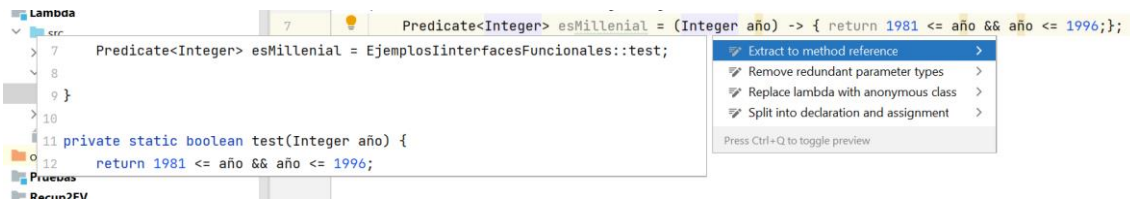
Su método abstracto es:

boolean test(T t)    Evalúa este predicado en el argumento dado.

Es decir, pasamos un valor de cualquier tipo, y nos devuelve true o false según la condición que especifiquemos nosotros.

Por ejemplo: voy a hacer un predicado que, pasando el año de nacimiento, me diga si es millennial o no (puedes ayudarte de IntelliJ IDEA para ver cómo se implementaría el método correspondiente):





```
Predicate<Integer> esMillennial = año -> 1981 <= año && año <= 1996;
// Utilizamos ese predicado
boolean resultado = esMillennial.test(1995);
```

Hay algunos métodos que necesitan un objeto de tipo Predicate para ser utilizados.

- **removeIf(Predicate<T> predicate):** se utiliza para filtrar elementos de una colección que cumplan cierta condición especificada por el Predicate.

Por ejemplo, imaginemos que en el ArrayList de socios queremos quitar los socios cuyo nombre comience por V.

```
System.out.println("Quitamos los que empiezan por V");
socios.removeIf(socio -> socio.nombre.startsWith("V"));
```

## Function<T, R>

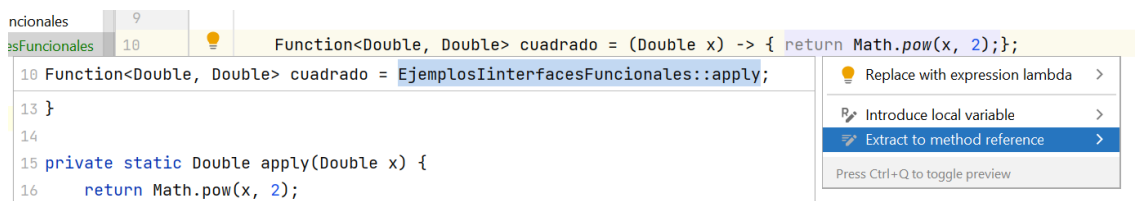
Representa la funcionalidad de las funciones matemáticas (f(x)).

Su único método abstracto es

**R apply(T t):** Aplica una función al argumento dado.

Es decir, recibe un parámetro de cualquier tipo (T), hace una serie de operaciones con él y devuelve el resultado, que puede ser del mismo tipo o no (R).

Por ejemplo: utilizamos Function para calcular el cuadrado de un número:



Abreviando la expresión:

```
Function<Double, Double> cuadrado = x -> Math.pow(x, 2);
// Utilizamos esta función
double cuad = cuadrado.apply(5.0); // cuad = 25
```

Hay algunos métodos que necesitan un objeto de tipo Function para ser utilizados.

- **replaceAll(Function<? super K, ? extends V> function):** Este método se utiliza en la clase Map para reemplazar cada valor del mapa por el resultado de aplicar la función especificada por el Function.

Por ejemplo, supongamos que tenemos un mapa que asocia nombres (clave) con edades (valor), y que queremos aumentar a todas las entradas la edad en un año:

```
Map<String, Integer> mapaEdades = new HashMap<>();
mapaEdades.put("Juan", 25);
mapaEdades.put("María", 30);
Function<Integer, Integer> aumentarEdad = edad -> edad + 1;
mapaEdades.replaceAll((nombre, edad) -> aumentarEdad.apply(edad));
System.out.println(mapaEdades); // Imprime {Juan=26, María=31}
```

## Consumer<T>

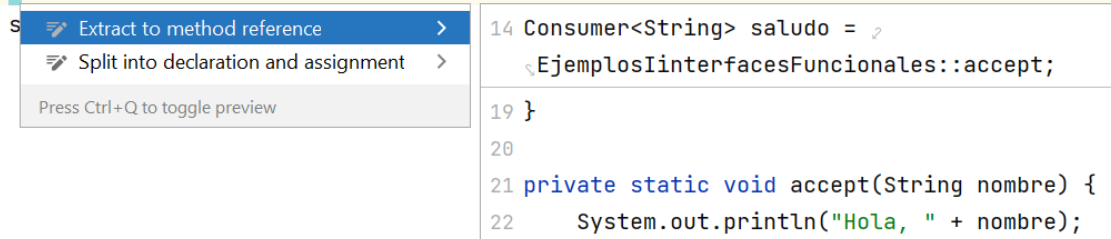
Su único método abstracto es:

`void accept(T t)`: Realiza una operación en el argumento dado.

Es decir, recibe un argumento de cualquier tipo y hace lo que sea con él, pero no devuelve nada (sólo consume el argumento).

Ejemplo: queremos saludar al nombre recibido como parámetro:

```
Consumer<String> saludo = (String nombre) -> {
    System.out.println("Hola, " + nombre);
};
```



Abreviando la expresión:

```
Consumer<String> saludo = nombre -> System.out.println("Hola, " +
nombre);
saludo.accept("Caracola");
```

Una aplicación inmediata de esto es imprimir listas:

```
Consumer<List<Socio>> imprimeSocios = (lista) -> {
    for (Socio socio: lista) {
        System.out.println(socio);
    }
};
```

Un método que utiliza un objeto de tipo Consumer es el método `forEach` de las colecciones.

```
socios.forEach(socio -> System.out.println(socio));
```

## Supplier<T>

Representa un proveedor de resultados. Hace lo contrario que Consumer, es decir, no recibe nada y devuelve algo.

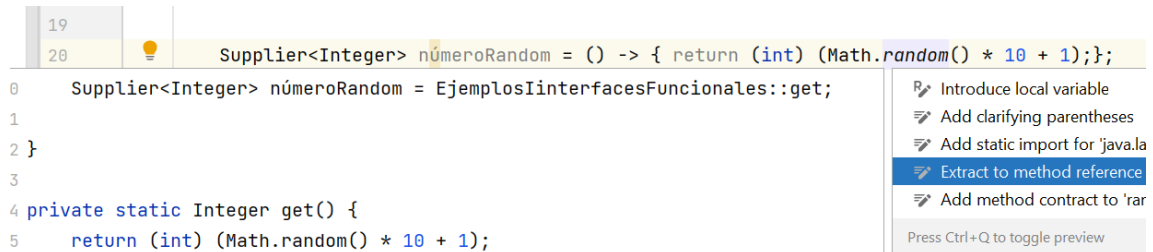
Su único método abstracto es:

**T get()** Obtiene un resultado.

Ejemplo: generamos un número aleatorio.

```
19
20
Supplier<Integer> númeroRandom = () -> { return (int) (Math.random() * 10 + 1);};

0 Supplier<Integer> númeroRandom = EjemplosInterfacesFuncionales::get;
1
2 }
3
4 private static Integer get() {
5     return (int) (Math.random() * 10 + 1);
}
```



```
Supplier<Integer> númeroRandom = () -> (int) (Math.random() * 100);
System.out.println(númeroRandom.get());
```

## Referencias a métodos

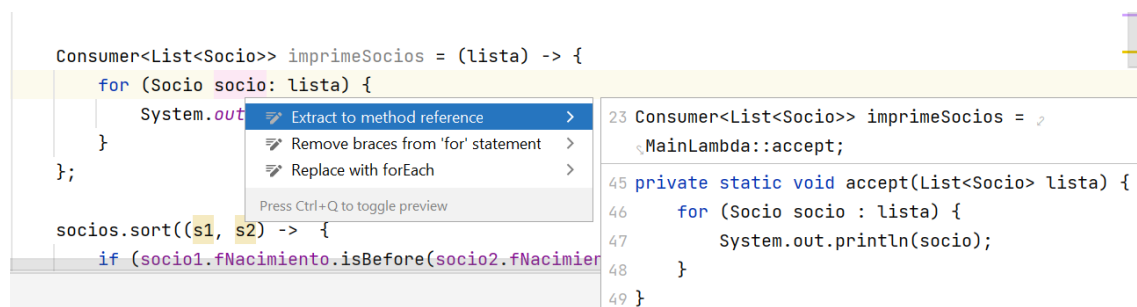
A partir de la versión 8 de Java, junto con las expresiones lambda podemos trabajar con referencias a métodos.

Las referencias a métodos en Java son una característica que permite simplificar la creación de expresiones lambda que llaman a un método ya existente. En lugar de escribir una lambda que implementa la lógica del método, se puede hacer referencia directamente al método mediante su nombre.

Ejemplo:

```
Function<String, Integer> miFuncion = Integer::parseInt;
miFuncion.apply("5"); // devuelve el entero 5
```

Hay varias formas de utilizar esta sintaxis en función del contexto. Una manera de irse familiarizando con ella es aprovechar las sugerencias del IDE.



## Referencias

---

<https://dzone.com/articles/java-lambda-expressions-basics>

<https://cursos.arquitecturajava.com/p/java-lambda>

Jiménez Martín, Alfonso y Pérez Montes, Francisco Manuel (2021). *Programación*. Editorial Paraninfo.

<https://lemus.webs.upv.es/wordpress/index.php/java/>

[Y ChatGPT](#)