

Programación Funcional en Java 8

Por: Ing. Sergio Rubén Irigoyen Guerra

Historia

Historia

- Clases Internas Anónimas

```
JButton button = new JButton("Haga click aquí.");  
  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Click.");  
    }  
});
```

Historia

- Clases Internas Anónimas

```
JButton button = new JButton("Haga click aquí.");  
  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Click.");  
    }  
});
```

Clase interna
anónima

Historia

- Clases Internas Anónimas

```
JButton button = new JButton("Haga click aquí.");  
  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Click.");  
    }  
});
```

Sintaxis difícil
de manejar

Clase interna
anónima

Historia

- Clases Internas Anónimas

```
JButton button = new JButton("Haga click aquí.");  
  
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Click.");  
    }  
});
```

Sintaxis difícil
de manejar

Clase interna
anónima

Problema de
"verticalidad"

Interfaces Funcionales

Interfaces Funcionales

- Una interfaz que tiene exactamente un método abstracto

Interfaces Funcionales

- Una interfaz que tiene exactamente un método abstracto
 - Runnable
 - Comparator
 - Comparable
 - ActionListener
 - EventHandler

Interfaces Funcionales

```
public interface Runnable {  
    public abstract void run();  
}
```

Interfaces Funcionales

Nueva anotación
en Java 8

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

Interfaces Funcionales

Nueva anotación
en Java 8

No es necesaria,
pero es útil

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

Expresiones Lambda

Expresiones Lambda

`() -> {} ;`

Parámetros
de entrada

Expresiones Lambda

`() -> {} ;`

Parámetros
de entrada

Bloque de código

Expresiones Lambda

Un sólo
parámetro

```
(String s) -> {};  
(String s, Algo c) -> {};
```


Expresiones Lambda

Un sólo
parámetro

```
(String s) -> {};  
(String s, Algo c) -> {};
```

Múltiples parámetros,
separados por comas.

Expresiones Lambda

Bloque de código

```
(String s) -> {System.out.println(s);};
```

Expresiones Lambda

Bloque de código

```
(String s) -> {System.out.println(s);};  
(String s) -> System.out.println(s);
```

Se pueden omitir
las llaves en
ciertos casos

Expresiones Lambda

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

Bloque de código
con múltiples
instrucciones

Expresiones Lambda

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

Interface
Comparator

Expresiones Lambda

```
Comparator<String> comp = (String s1, String s2) ->  
    s1.compareTo(s2);
```

```
int result = comp.compare("Argentina", "Aruba");
```

Implementación
usando una
expresión lambda

Expresiones Lambda

```
Comparator<String> comp = (String s1, String s2) ->  
    s1.compareTo(s2);
```

```
int result = comp.compare("Argentina", "Aruba");
```

Usando la expresión
lambda para obtener
un resultado

Expresiones Lambda

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Click.");  
    }  
});
```

```
button.addActionListener((ActionEvent e) ->  
    System.out.println("Click."));
```


Expresiones Lambda

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Click.");  
    }  
});
```

Clase interna
anónima

```
button.addActionListener((ActionEvent e) ->  
    System.out.println("Click."));
```

Expresión
lambda

Inferencia de Tipos

Inferencia de Tipos

```
button.addActionListener((ActionEvent e) ->  
    System.out.println("Click."));
```

Inferencia de Tipos

```
button.addActionListener((ActionEvent e) ->  
    System.out.println("Click."));
```



```
button.addActionListener(e ->  
    System.out.println("Click."));
```

Inferencia de Tipos

```
button.addActionListener((ActionEvent e) ->  
    System.out.println("Click."));
```



Tipo
Inferido

```
button.addActionListener(e ->  
    System.out.println("Click."));
```

Inferencia de Tipos

```
button.addActionListener((ActionEvent e) ->  
    System.out.println("Click."));
```



Tipo
Inferido

```
button.addActionListener(e ->  
    System.out.println("Click."));
```

Se pueden
omitir los
paréntesis

Inferencia de Tipos

```
Comparator<String> comp = (String s1, String s2) ->  
    s1.compareTo(s2);
```



```
Comparator<String> comp = (s1, s2) -> s1.compareTo(s2);
```

Inferencia de Tipos

```
Comparator<String> comp = (String s1, String s2) ->  
    s1.compareTo(s2);
```



Tipos
Inferidos

```
Comparator<String> comp = (s1, s2) -> s1.compareTo(s2);
```


Alcance léxico

Alcance léxico

```
public class ScopeTest {

    public static void main(String[] args) {
        ScopeTest j = new ScopeTest();
        j.hacer();
    }

    public void hacer() {
        Runnable r = () -> System.out.println(toString());
        r.run();
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                System.out.println(toString());
            }
        };
        r2.run();
    }

    @Override
    public String toString(){
        return "Test";
    }
}
```

Alcance léxico

```
public class ScopeTest {  
  
    public static void main(String[] args) {  
        ScopeTest j = new ScopeTest();  
        j.hacer();  
    }  
  
    public void hacer() {  
        Runnable r = () -> System.out.println(toString());  
        r.run();  
        Runnable r2 = new Runnable() {  
            @Override  
            public void run() {  
                System.out.println(toString());  
            }  
        };  
        r2.run();  
    }  
  
    @Override  
    public String toString(){  
        return "Test";  
    }  
}
```

Imprime
"ScopeTest\$1@5acf9800"

Alcance léxico

```
public class ScopeTest {  
  
    public static void main(String[] args) {  
        ScopeTest j = new ScopeTest();  
        j.hacer();  
    }  
  
    public void hacer() {  
        Runnable r = () -> System.out.println(toString());  
        r.run();  
        Runnable r2 = new Runnable() {  
            @Override  
            public void run() {  
                System.out.println(toString());  
            }  
        };  
        r2.run();  
    }  
  
    @Override  
    public String toString(){  
        return "Test";  
    }  
}
```

Imprime
"Test"

Imprime
"ScopeTest\$1@5acf9800"

Referencias a métodos

Referencias a métodos

```
public abstract void run();
```

```
public static void imprime() {  
    System.out.println("Funciona");  
}
```

Referencias a métodos

```
public abstract void run();
```



Mismo tipo
de retorno

```
public static void imprime() {  
    System.out.println("Funciona");  
}
```

Referencias a métodos

```
public abstract void run();
```

Mismo tipo
de retorno

Mismos
parámetros

```
public static void imprime() {  
    System.out.println("Funciona");  
}
```


Referencias a métodos

```
public abstract void run();
```

Mismo tipo
de retorno

Mismos
parámetros

```
public static void imprime() {  
    System.out.println("Funciona");  
}
```

Firma compatible

Referencias a métodos

```
public class MethodReferenceTest {
    public static void main(String[] args) {
        Runnable r2 = MethodReferenceTest::imprime;
        r2.run();
    }

    public static void imprime(){
        System.out.println("Funciona");
    }
}
```

Referencias a métodos

```
public class MethodReferenceTest {  
    public static void main(String[] args) {  
        Runnable r2 = MethodReferenceTest::imprime;  
        r2.run();  
    }  
  
    public static void imprime(){  
        System.out.println("Funciona");  
    }  
}
```



Referencia a
método

Interfaces Funcionales Predefinidas en Java 8

Interfaces funcionales predefinidas en Java 8

- `java.util.function`
 - `Consumer<T>` argumento tipo T y no retorna ningún resultado
 - `Function<T, R>` argumento tipo T y retorna un resultado tipo R
 - `Supplier<T>` sin argumentos y retorna un resultado tipo T
 - `Predicate<T>` argumento tipo T y retorna un resultado tipo `boolean`

Métodos de extensión virtuales

Métodos de extensión virtuales

- **Método** `forEach` de la interfaz `Iterable`

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

Métodos de extensión virtuales

- Método `forEach` de la interfaz `Iterable`

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

Método
default

Métodos de extensión virtuales

- Método `forEach` de la interfaz `Iterable`

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

Consumer es una
Interfaz funcional

Métodos de extensión virtuales

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
}
```

Métodos de extensión virtuales

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
}
```

Un sólo
Método
abstracto

Métodos de extensión virtuales

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

}
```



Métodos
default

Streams

Streams

- Operaciones en Streams
 - Terminales
 - `count()`
 - `forEach()`
 - Intermedias
 - `filter()`
 - `distinct()`

Streams

- Operaciones en Streams

- Terminales

- `count()`
 - `forEach()`

- Intermedias

- `filter()`
 - `distinct()`



Lazy
(flojas)

Streams

- Operaciones en Streams

- Terminales

- `count()`
 - `forEach()`



Eager
(ansiosas)

- Intermedias

- `filter()`
 - `distinct()`



Lazy
(flojas)

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}
```

```
Predicate<Integer> sm5 = s -> s >= 50;  
Predicate<Integer> sm7 = s -> s <= 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado: " + b);
```

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}
```

Lista con cien
números entre
0 y 100

```
Predicate<Integer> sm5 = s -> s >= 50;  
Predicate<Integer> sm7 = s -> s <= 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado: " + b);
```

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();
Random rnd = new Random();
for (int i = 0; i < 100; i++) {
    numeros.add(rnd.nextInt(100));
}
```

```
Predicate<Integer> sm5 = s -> s >= 50;
Predicate<Integer> sm7 = s -> s <= 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado: " + b);
```

Obtenemos el
Stream

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}
```

```
Predicate<Integer> sm5 = s -> s >= 50;  
Predicate<Integer> sm7 = s -> s <= 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado " + b);
```

filter() recibe un
Predicate como
parámetro

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}
```

```
Predicate<Integer> sm5 = i -> i >= 50;  
Predicate<Integer> sm7 = i -> i < 70;
```

Creamos dos Predicates
utilizando expresiones lambda

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado " + b);
```

filter() recibe un
Predicate como
parámetro

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}
```

```
Predicate<Integer> sm5 = i -> i >= 50;  
Predicate<Integer> sm7 = i -> i < 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado: " + b);
```

Utilizamos el método default
and() de Predicate

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}
```

```
Predicate<Integer> sm5 = i -> i >= 50;  
Predicate<Integer> sm7 = i -> i < 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado: " + b);
```

Contamos el número
de elementos

Streams

- Si tenemos una lista con cien números entre 0 y 100, y queremos obtener la cantidad de números que sean mayores o iguales a 50 y menores a 70, podemos hacerlo de la siguiente forma usando Streams y expresiones lambda:

```
List<Integer> numeros = new ArrayList<>();
Random rnd = new Random();
for (int i = 0; i < 100; i++) {
    numeros.add(rnd.nextInt(100));
}
```

```
Predicate<Integer> sm5 = i -> i >= 50;
Predicate<Integer> sm7 = i -> i < 70;
```

```
long b = numeros.stream().filter(sm5.and(sm7)).count();
```

```
System.out.println("Resultado: " + b);
```

Mostrar el resultado

Streams

- Optimización:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();  
rnd.ints(100, 0, 100).forEach(numeros::add);
```

```
Predicate<Integer> sm5 = i -> i >= 50;  
Predicate<Integer> sm7 = i -> i < 70;
```

```
long b = numeros.stream().filter((i) -> (i>=50 && i<70)).count();
```

```
System.out.println("Resultado: " + b);
```

Convertir a Stream
y expresiones lambda

Streams

- **Java 7 vs Java 8:**

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();
```

```
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}  
int count = 0;  
for (Integer i : numeros) {  
    if(i>=50 && i<70){  
        count++;  
    }  
}
```

Java 7

```
rnd.ints(100, 0, 100).forEach(numeros::add);  
long b = numeros.stream().filter(i -> i>=50 && i<70).count();
```

```
System.out.println("Resultado Java7: " + count);  
System.out.println("Resultado Java8: " + b);
```

Streams

- Java 7 vs Java 8:

```
List<Integer> numeros = new ArrayList<>();  
Random rnd = new Random();
```

```
for (int i = 0; i < 100; i++) {  
    numeros.add(rnd.nextInt(100));  
}  
int count = 0;  
for (Integer i : numeros) {  
    if(i>=50 && i<70){  
        count++;  
    }  
}
```

Java 7

Java 8

```
rnd.ints(100, 0, 100).forEach(numeros::add);  
long b = numeros.stream().filter(i -> i>=50 && i<70).count();
```

```
System.out.println("Resultado Java7: " + count);  
System.out.println("Resultado Java8: " + b);
```

Preguntas

¡Gracias!