(http://baeldung.com)

Expresiones Lambda e interfaces funcionales: consejos y mejores prácticas

Última modificación: 31 de agosto de 2017

por baeldung (http://www.baeldung.com/author/baeldung/)
(http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/) +

1. Información general

Ahora que Java 8 ha alcanzado un amplio uso, los patrones y las mejores prácticas han comenzado a surgir para algunas de sus características principales. En este tutorial, veremos más de cerca las interfaces funcionales y las expresiones lambda.

2. Prefiere las interfaces funcionales estándar

Las interfaces funcionales, que se recopilan en el paquete **java.util.function (https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html)** , satisfacen las necesidades de la mayoría de los desarrolladores al proporcionar tipos de destino para expresiones lambda y referencias de métodos. Cada una de estas interfaces es general y abstracta, lo que facilita su adaptación a casi cualquier expresión lambda. Los desarrolladores deben explorar este paquete antes de crear nuevas interfaces funcionales.

Considere una interfaz *Foo* :

```
1 @FunctionalInterface
2 public interface Foo {
3     String method(String string);
4 }
```

y un método *add ()* en alguna clase *UseFoo* , que toma esta interfaz como un parámetro:

```
1 public String add(String string, Foo foo) {
2     return foo.method(string);
3 }
```

Para ejecutarlo, deberías escribir:

```
1 Foo foo = parameter -> parameter + " from lambda";
2 String result = useFoo.add("Message ", foo);
```

Mire más de cerca y verá que *Foo* no es más que una función que acepta un argumento y produce un resultado. Java 8 ya proporciona dicha interfaz en la *Función <T, R> (https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)* del paquete java.util.function (https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) .

Ahora podemos eliminar la interfaz *Foo por* completo y cambiar nuestro código a:

```
public String add(String string, Function<String, String> fn) {
    return fn.apply(string);
}
```

Para ejecutar esto, podemos escribir:

```
Function<String, String> fn =
  parameter -> parameter + " from lambda";
String result = useFoo.add("Message ", fn);
```

3. Utilice la anotación *@FunctionalInterface*

Anota tus interfaces funcionales con *@FunctionalInterface (https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)* . Al principio, esta anotación parece ser inútil. Incluso sin él, su interfaz se tratará como funcional siempre que solo tenga un método abstracto.

Pero imagine un gran proyecto con varias interfaces: es difícil controlar todo manualmente. Una interfaz, que fue diseñada para ser funcional, podría ser cambiada accidentalmente agregando otros métodos / métodos abstractos, dejándola inutilizable como una interfaz funcional.

Pero al usar la anotación *@FunctionalInterface* , el compilador desencadenará un error en respuesta a cualquier intento de romper la estructura predefinida de una interfaz funcional. También es una herramienta muy útil para hacer que la arquitectura de su aplicación sea más fácil de entender para otros desarrolladores.

Entonces, usa esto:

```
@FunctionalInterface
public interface Foo {
    String method();
}
```

en lugar de simplemente

```
public interface Foo {
    String method();
}
```

4. No use en exceso los métodos predeterminados en las interfaces funcionales

Puede agregar fácilmente métodos predeterminados a la interfaz funcional. Esto es aceptable para el contrato de interfaz funcional siempre que haya una sola declaración de método abstracto:

```
@FunctionalInterface
public interface Foo {
    String method();
    default void defaultMethod() {}
}
```

Las interfaces funcionales pueden ser extendidas por otras interfaces funcionales si sus métodos abstractos tienen la misma firma. Por ejemplo:

```
@FunctionalInterface
public interface FooExtended extends Baz, Bar {}

@FunctionalInterface
public interface Baz {
    String method();
    default void defaultBaz() {}
}

@FunctionalInterface
public interface Bar {
    String method();
    default void defaultBar() {}
}
```

Al igual que con las interfaces regulares, extender diferentes interfaces funcionales con el mismo método predeterminado puede ser problemático. Por ejemplo, supongamos que las interfaces *Bar* y *Baz* tienen un método predeterminado *defaultCommon ().* En este caso, obtendrá un error en tiempo de compilación:

```
interface Foo inherits unrelated defaults for defaultCommon() from types Baz
```

Para solucionar esto, el método *defaultCommon ()* debe ser anulado en la interfaz de *Foo* . Por supuesto, puede proporcionar una implementación personalizada de este método. Pero si desea utilizar una de las implementaciones de las interfaces principales (por ejemplo, desde la interfaz *Baz*), agregue la siguiente línea de código al cuerpo del método *defaultCommon ()* :

```
Baz.super.defaultCommon();
```

Pero ten cuidado. **Agregar demasiados métodos predeterminados a la interfaz no es una muy buena decisión arquitectónica.** Se debe considerar como un compromiso, solo para ser utilizado cuando sea necesario, para actualizar las

interfaces existentes sin romper la compatibilidad con versiones anteriores.

5. Crear una instancia de interfaces funcionales con expresiones lambda

El compilador le permitirá usar una clase interna para instanciar una interfaz funcional. Sin embargo, esto puede conducir a un código muy detallado. Deberías preferir las expresiones lambda:

```
Foo foo = parameter -> parameter + " from Foo";
```

sobre una clase interna:

```
Foo fooByIC = new Foo() {
    @Override
    public String method(String string) {
        return string + " from Foo";
    }
};
```

El enfoque de expresión lambda se puede usar para cualquier interfaz adecuada de bibliotecas antiguas. Se puede usar para interfaces como *Runnable* , *Comparator* , etc. **Sin embargo, esto no significa que deba revisar toda su base de códigos anterior y cambiar todo.**

6. Evite métodos de sobrecarga con interfaces funcionales como parámetros

Use métodos con diferentes nombres para evitar colisiones; Veamos un ejemplo:

```
public interface Adder {
    String add(Function<String, String> f);
    void add(Consumer<Integer> f);
}

public class AdderImpl implements Adder {

    @Override
    public  String add(Function<String, String> f) {
        return f.apply("Something ");
    }

    @Override
    public void add(Consumer<Integer> f) {}
}
```

A primera vista, esto parece razonable. Pero cualquier intento de ejecutar cualquiera de *los* métodos de *AdderImpl* :

```
String r = adderImpl.add(a -> a + " from lambda");
```

finaliza con un error con el siguiente mensaje:

```
reference to add is ambiguous both method
add(java.util.function.Function<java.lang.String,java.lang.String>)
in fiandlambdas.AdderImpl and method
add(java.util.function.Consumer<java.lang.Integer>)
in fiandlambdas.AdderImpl match
```

Para resolver este problema, tienes dos opciones. El **primero** es usar métodos con diferentes nombres:

```
String addWithFunction(Function<String, String> f);

void addWithConsumer(Consumer<Integer> f);
```

El **segundo** es realizar el lanzamiento de forma manual. Esto no es preferido

```
String r = Adder.add((Function) a -> a + " from lambda");
```

7. No trate las expresiones Lambda como clases internas

A pesar de nuestro ejemplo anterior, donde esencialmente sustituimos la clase interna por una expresión lambda, los dos conceptos son diferentes de una manera importante: alcance.

Cuando usa una clase interna, crea un nuevo alcance. Puede sobreescribir las variables locales del alcance adjunto creando instancias de nuevas variables locales con los mismos nombres. También puede usar la palabra clave ***this*** dentro de su clase interna como referencia a su instancia.

Sin embargo, las expresiones lambda funcionan con el alcance adjunto. No puede sobrescribir las variables del ámbito adjunto dentro del cuerpo de lambda. En este caso, la palabra clave ***esta*** es una referencia a una instancia que encierra.

Por ejemplo, en la clase *UseFoo* tienes un *valor de* variable de instancia *:*

```
1 private String value = "Enclosing scope value";
```

Luego, en algún método de esta clase, coloque el siguiente código y ejecute este método.

```
1  public String scopeExperiment() {
2      Foo fooIC = new Foo() {
3          String value = "Inner class value";
4
5          @Override
6          public String method(String string) {
7              return this.value;
8          }
9      };
10     String resultIC = fooIC.method("");
11
12     Foo fooLambda = parameter -> {
13         String value = "Lambda value";
14         return this.value;
15     };
16     String resultLambda = fooLambda.method("");
17
18     return "Results: resultIC = " + resultIC +
19       ", resultLambda = " + resultLambda;
20 }
```

Si ejecuta el método *scopeExperiment ()* , obtendrá el siguiente resultado: *Resultados: resultIC = Valor interno de la clase, resultLambda = Valor del alcance adjunto*

Como puede ver, al invocar *this.value* en IC, puede acceder a una variable local desde su instancia. Pero en el caso de la lambda, *esta* llamada de valor le da acceso al *valor de* la variable que se define en la clase *UseFoo* , pero no al *valor de* la variable definida dentro del cuerpo de la lambda.

8. Mantenga las expresiones Lambda cortas y autoexplicativas

Si es posible, use construcciones de una línea en lugar de un bloque grande de código. Recuerda que **lambdas debería ser una expresión, no una narración.** A pesar de su sintaxis concisa, **lambdas debe expresar con precisión la funcionalidad que proporcionan.**

Esto es principalmente un consejo estilístico, ya que el rendimiento no cambiará drásticamente. En general, sin embargo, es mucho más fácil de entender y trabajar con dicho código.

Esto se puede lograr de muchas maneras, echemos un vistazo más de cerca.

8.1. Evitar bloques de código en el cuerpo de Lambda

En una situación ideal, lambdas debería escribirse en una línea de código. Con este enfoque, la lambda es una construcción autoexplicativa, que declara qué acción se debe ejecutar con qué datos (en el caso de lambdas con parámetros).

Si tiene un bloque grande de código, la funcionalidad de lambda no está clara inmediatamente.

Con esto en mente, haga lo siguiente:

```
Foo foo = parameter -> buildString(parameter);
```

```
private String buildString(String parameter) {
    String result = "Something " + parameter;
    //many lines of code
    return result;
}
```

en lugar de:

```
Foo foo = parameter -> { String result = "Something " + parameter;
    //many lines of code
    return result;
};
```

Sin embargo, no use esta regla de "lambda de una sola línea" como dogma . Si tiene dos o tres líneas en la definición de lambda, puede que no sea valioso extraer ese código en otro método.

8.2. Evite especificar tipos de parámetros

Un compilador en la mayoría de los casos puede resolver el tipo de parámetros lambda con la ayuda de la **inferencia (https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html)** de **tipo (https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html)**. Por lo tanto, agregar un tipo a los parámetros es opcional y se puede omitir.

Hacer esto:

```
1 (a, b) -> a.toLowerCase() + b.toLowerCase();
```

en lugar de esto:

```
1 (String a, String b) -> a.toLowerCase() + b.toLowerCase();
```

8.3. Evitar paréntesis alrededor de un parámetro único

La sintaxis Lambda requiere paréntesis solo alrededor de más de un parámetro o cuando no hay ningún parámetro. Es por eso que es seguro hacer que su código sea un poco más corto y excluir paréntesis cuando solo hay un parámetro.

Entonces, haz esto:

```
1 a -> a.toLowerCase();
```

en lugar de esto:

```
1 (a) -> a.toLowerCase();
```

8.4. Evitar declaración de devolución y llaves

Las llaves y las declaraciones de ***retorno*** son opcionales en cuerpos lambda de una sola línea. Esto significa que pueden omitirse por claridad y concisión.

Hacer esto:

```
1 a -> a.toLowerCase();
```

en lugar de esto:

```
1 a -> {return a.toLowerCase()};
```

8.5. Usar las referencias del método

Muy a menudo, incluso en nuestros ejemplos anteriores, las expresiones lambda simplemente llaman a métodos que ya están implementados en otros lugares. En esta situación, es muy útil usar otra característica de Java 8: **referencias de métodos (https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html)** .

Entonces, la expresión lambda:

```
1 a -> a.toLowerCase();
```

podría ser sustituido por:

```
1 String::toLowerCase;
```

Esto no siempre es más corto, pero hace que el código sea más legible.

9. Usar variables "efectivamente finales"

El acceso a una variable no final dentro de las expresiones lambda causará el error en tiempo de compilación. **Pero eso no significa que deba marcar todas las variables objetivo como *definitivas.***

De acuerdo con el concepto " **efectivamente final (https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html)** " , un compilador trata cada variable como *definitiva,* siempre que se asigne solo una vez.

Es seguro usar tales variables dentro de lambdas porque el compilador controlará su estado y desencadenará un error en tiempo de compilación inmediatamente después de cualquier intento de cambiarlas.

Por ejemplo, el siguiente código no se compilará:

```
1
2
3
4
5
6
7
```


El c ';

ameter;

```
1
```

dy defined in the scope.

Este eso de hacer que la ejecución de lambda sea

10. Proteja las variables de objeto de la mutación

Uno de los princ ambdas es el uso en informática paralela, lo que significa que cuando se trata de seguridad de hilos.

El paradigma "efectivamente final" ayuda mucho aquí, pero no en todos los casos. Lambdas no puede cambiar el valor de un objeto del alcance circundante. Pero en el caso de variables de objetos mutables, un estado podría cambiarse dentro de las expresiones lambda.

Considera el siguiente código:

```
1 int[] total = new int[1];
2 Runnable r = () -> total[0]++;
3 r.run();
```

Este código es legal, ya que *la* variable *total* sigue siendo "efectivamente final". ¿Pero el objeto al que hace referencia tiene el mismo estado después de la ejecución de la lambda? ¡No!

Mantenga este ejemplo como un recordatorio para evitar el código que puede causar mutaciones inesperadas.

11. Conclusión

En este tutorial, vimos algunas mejores prácticas y riesgos en las expresiones lambda e interfaces funcionales de Java 8. A pesar de la utilidad y el poder de estas nuevas funciones, son solo herramientas. Cada desarrollador debe prestar

atención mientras los usa.

El **código fuente** completo para el ejemplo está disponible en este proyecto GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-8) : este es un proyecto de Maven y Eclipse, por lo que puede importarse y utilizarse tal cual.

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-main-1.2.0.jpg)

(http://www.baeldung.com/wp-content/uploads/2016/05/baeldung-rest-post-footer-icn-1.0.0.png)

Learning to "Build your API **with Spring**"?

Enter your Email Address

>> Get the eBook

Subscribe ▾

▲ newest ▲ oldest ▲ most voted

Guest

Nicola Malizia (https://unnikked.ga/)

Very helpful

blog post! I read it with pleasure. I will definitively keep it in mind, thank you!

+ 2 −

2 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

Glad you enjoyed it Nicola. Cheers,
Eugen.

+ 1 −

2 years ago

Guest

Ola Kunysz

Good read, but it would have much more value if you use real life examples. If you replace Foo foo with some real code, it will be way easier to recall these practices in the right moment 🙂

+ 0 −

2 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

That's a good point, I'll add it to the content calendar of the site.
Cheers,
Eugen.

+ 0 −

2 years ago

Guest

Lukas Eder (http://www.jooq.org)

Regarding #10, this isn't true *in general*, but it is indeed a bad idea to implement lambdas with side-effects (stateful lambdas) in the context of some stream operations, e.g. `filter()` .

+ 0 −

2 years ago

Guest

Eugen Paraschiv (http://www.baeldung.com/)

Hey Lukas, I was actually thinking that I should run this article by you while I was reviewing it 🙂
Yeah, definitely agree – side effects are almost never a good thing as idempotence is such a usefull property. The more functional approach does help there, but since there's really no language level mechanism to ensure that, you need to be quite careful with that aspect.
Thanks for the feedback and keep in touch. Cheers,
Eugen.

+ 0 −

2 years ago

Guest

Alex Ve

En mi opinión, el # 10 se trata más de mostrar un verdadero denger de efectos secundarios inesperados, que de ofrecer una explicación de tales efectos secundarios. Tal vez, sería mejor llamar al # 10 - "Evitar mutaciones inesperadas de variables de objetos al usar lambdas" o algo así.

+ 0 −

Hace 2 años

Huésped

Brian Oxley

En tu ejemplo:

Foo foo = parámetro -> buildString (parámetro);

Puede preferir un ejemplo donde "buildString" tome un argumento adicional. De lo contrario, preferiría:

Foo foo = this :: buildString;

+ 0 −

Hace 2 años

Huésped

Eugen Paraschiv (http://www.baeldung.com/)

El operador de dos puntos es sin duda una adición muy útil al lenguaje; puedo explorarlo en un artículo dedicado. Saludos,
Eugen.

+ 0 −

Hace 2 años

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER A LO BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA DE

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)