java 8

Funciones Lambda

Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase.

Su sintaxis básica se detalla a continuación:

(parámetros) -> {cuerpo-lambda}

El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.

Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

Cuerpo de lambda:

Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula return en el caso de que deban devolver valores.

Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula return en el caso de que la función deba devolver un valor.

Algunos ejemplos de expresiones lambda pueden ser:

z -> z + 2

() -> System.out.println(” Mensaje 1 “)

(int longitud, int altura) -> { return altura \* longitud; }

(String x) -> { String retorno = x;

retorno = retorno.concat(” \*\*\*”);

return retorno; }

Podemos implementar una interfaz funcional (solo tiene un metódo abtracto) con una expresión lambda:

package com.antonio.test;

@FunctionalInterface // es opcional

public interface ICalculadoraLambda {

public abstract int operacion (int x, int y);

}

package com.antonio.test;

public class TestCalculadoraLambda {

ICalculadoraLambda iSuma = (x, y) -> x + y;

ICalculadoraLambda iMultiplica = (x, y) -> x \* y;

public void test1 () {

System.out.println("suma=" + iSuma.operacion (2, 2));

System.out.println("multiplicacion=" + iMultiplica.operacion (2, 3));

}

public static void main(String[] args) {

TestCalculadoraLambda c = new TestCalculadoraLambda();

c.test1();

}

}

Tipos de funciones Lambda

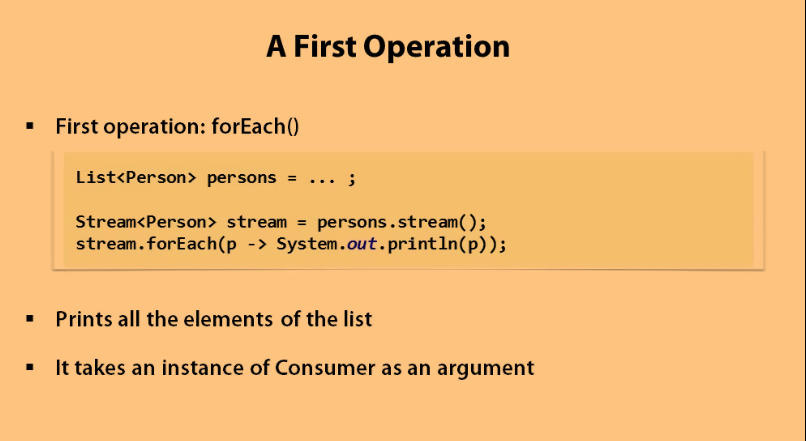
Consumidores

Se trata de aquellas expresiones lambda que aceptan un solo valor y no devuelven valor alguno.

String message -> System.out.println(message);

Las expresiones BiConsumidoras, un caso especial de las expresiones consumidoras, son aquellas que toman dos valores como parámetro y no devuelven resultado.

(String key, String value) -> System.out.println("Key: %s, value: %s%n", key, value);



Proveedores

No tienen parámetros pero de vuelven un resultado

() -> return CreateRamdonInteger ();

Funciones

Aquellas expresiones que aceptan un argumento y devuelven un valor como resultado y cuyos tipos no tienen por qué ser iguales.

Order persistedOrder -> persistedOrder.getIdientifier();

Order persistedOrder -> persistedOrder.getIdientifier();

Las BiFunciones son aquellas expresiones de tipo función que aceptan dos argumentos y devuelven un resultado.

(Address address, String name) -> new Person(name, address);

(Address address, String name) -> new Person(name, address);

**Operadores Unarios**

Caso especial de funciones en las que tanto el parámetro como el valor devuelto son del mismo tipo.

String message -> message.toLowerCase()

String message -> message.toLowerCase()

**Operadores Binarios**

Igual que en el caso de los Operadores Unarios, se trata de un caso especial de funciones en las que los dos argumentos y el resultado son del mismo tipo.

(String message, String anotherMesssage) -> message.concat(anotherMessage);

(String message, String anotherMesssage) -> message.concat(anotherMessage);

Predicados

Se trata de expresiones que aceptan un parámetro y devuelven un valor lógico.

String message -> message.length > 50

String message -> message.length > 50

Como en los casos anteriores, se pueden tener BiPredicados, predicados que en lugar de tener un parámetro, tienen dos.

(path, attr) -> String.valueOf(path).endsWith(".js") && attr.size() > 1024

(path, attr) -> String.valueOf(path).endsWith(".js") && attr.size() > 1024

Referencias a métodos

Las referencias a los métodos nos permiten reutilizar un método como expresión lambda. Para hacer uso de las referencias a métodos basta con utilizar la siguiente sintáxis: referenciaObjetivo::nombreDelMetodo.

File::canRead // en lugar de File f -> f.canRead();

File::canRead // en lugar de File f -> f.canRead();

Con las referencias a los métodos se ofrece una anotación más rápida para expresiones lambda simples y existen 3 tipos diferentes:

Métodos estáticos.

Métodos de instancia de un tipo.

Métodos de instancia de un objeto existente.

Ejemplo de uso con método estático:

(String info) -> System.out.println(info) // Expresión lambda sin referencias.

System.out::println // Expresión lambda con referencia a método estático.

(String info) -> System.out.println(info) // Expresión lambda sin referencias.

System.out::println // Expresión lambda con referencia a método estático.

Ejemplo de uso con método de un tipo:

(Student student, int registryIndex) -> student.getRegistry(registryIndex) // Expresión lambda sin referencias.

Student::getRegistry // Expresión lambda con referencia a método de un tipo.

(Student student, int registryIndex) -> student.getRegistry(registryIndex) // Expresión lambda sin referencias.

Student::getRegistry // Expresión lambda con referencia a método de un tipo.

Ejemplo de uso con método de un objeto existente:

Student student -> getMarks(student) // Expresión lambda sin referencias.

this::getMarks // Expresión lambda con referencia a método de un objeto existente.

Student student -> getMarks(student) // Expresión lambda sin referencias.

this::getMarks // Expresión lambda con referencia a método de un objeto existente.

Optional, la alternativa a NULL.

Optional aparece en java 8 para evitar los nullPointerExcption que aparecen en la ejeución. Con la introducción de optional, se obliga al desarrollador a pensar que un objeto o una función puede tener o no algún objeto.

Para controlar que un ordenador tiene tarjeta de sonido y que esta tiene puerto USB, el código quedaría:

String version = "UNKNOWN";

if(computer != null){

Soundcard soundcard = computer.getSoundcard();

if(soundcard != null){

USB usb = soundcard.getUSB();

if(usb != null){

version = usb.getVersion();

}

}

}

La alternativa con optional queda:

class Computer {

private Optional<Soundcard> soundcard;

public Optional<Soundcard> getSoundcard() {

return soundcard;

};

}

class Soundcard {

private Optional<USB> usb;

public Optional<USB> getUSB() {

return usb;

}

}

class USB{

public String getVersion(){

return "v1.0";

}

}

String name = computer.flatMap(Computer::getSoundcard)

.flatMap(Soundcard::getUSB)

.map(USB::getVersion)

.orElse("UNKNOWN");

Creación de objetos Optional

En primer lugar, ¿cómo se crean objetos Optional? Existen diversos modos:

Este es un Optional vacío:

Optional<Soundcard> sc = Optional.empty();

Y este es un Optional con un valor no nulo:

SoundCard soundcard = new Soundcard();

Optional<Soundcard> sc = Optional.of(soundcard);

Si soundcard fuera nulo, se arrojaría de inmediato una excepción NullPointerException (en lugar de obtener un error latente cuando se intente acceder a las propiedades de soundcard).

Asimismo, utilizando ofNullable, es posible crear un objeto Optional que puede contener un valor nulo:

Optional<Soundcard> sc = Optional.ofNullable(soundcard);

Si soundcard fuera nulo, el objeto Optional resultante estaría vacío.

Valores predeterminados y acciones:

Soundcard soundcard =

maybeSoundcard != null ? maybeSoundcard

: new Soundcard("basic\_sound\_card");

Si se usa un objeto Optional, es posible reescribir el código anterior empleando el método orElse(), que proporciona un valor predeterminado si Optional está vacío:

Soundcard soundcard = maybeSoundcard.orElse(new Soundcard("defaut"));

//soundcar con objeto null

Soundcard soundcard2 = null;

Optional<Soundcard> scNull = Optional.ofNullable(soundcard2);

scNull.ifPresent(System.out::println);

Soundcard soundcard3 = scNull.orElse(new Soundcard("defaut"));

System.out.println(soundcard3.name);