



JDK 8
EXPRESIONES LAMB
DA



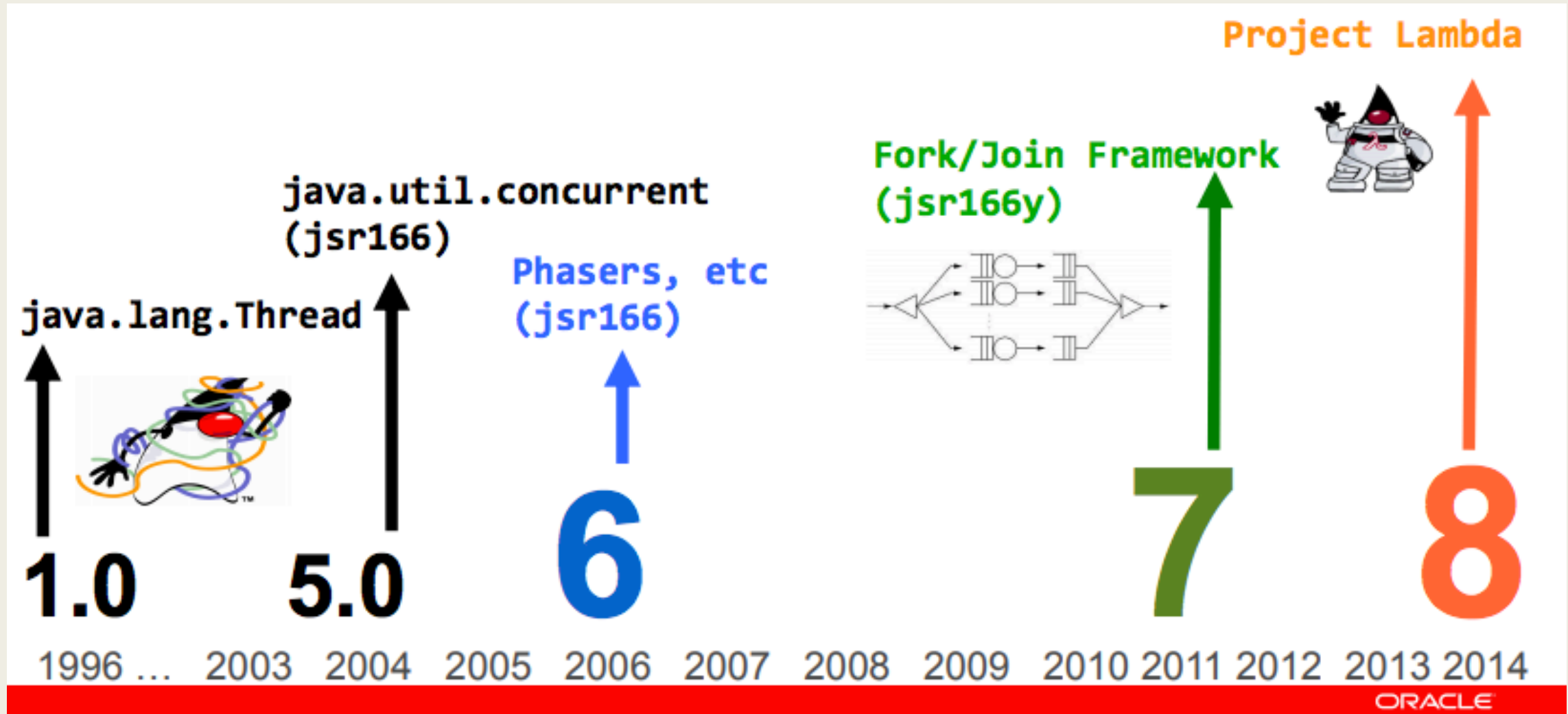
Objetivo:
Aprender a utilizar
las expresiones
lambda y los flujos
de Java 8



Expresiones lambda

- ¿Porqué Java requiere las expresiones lambda?
- Sintaxis de las expresiones lambda
- Interfaces funcionales y su definición
- Package `java.util.function`
- Referencia a métodos y constructores
- Referencia a variables externas
- Métodos útiles en el JDK 8 que pueden utilizar lambdas

Concurrencia en JAVA



Problema: Iteración externa

```
List<Student> students = ...  
double highestScore = 0.0;  
  
for (Student s : students) {  
    if (s.getGradYear() == 2011) {  
        if (s.getScore() > highestScore)  
            highestScore = s.getScore();  
    }  
}
```

- Nuestro código controla la iteración
- Inherentemente es serial: itera del inicio al fin
- No es thread-safe

Iteración interna con INNER CLASSES

```
double highestScore = students
    .filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.getGradYear() == 2011;
        }
    })
    .map(new Mapper<Student, Double>() {
        public Double extract(Student s) {
            return s.getScore();
        }
    })
    .max();
```

- La iteración es controlada por la biblioteca
- No es inherentemente serial – transversal se podría realizar en paralelo
- Se podría hacer transversal de forma perezosa (lazily)
- Thread safe
- Barrera: Sintacticamente horrible

Iteración interna con expresiones lambda

```
List<Student> students = ...  
double highestScore = students  
    .filter(Student s -> s.getGradYear() == 2011)  
    .map(Student s -> s.getScore())  
    .max();
```

- más legible
- más abstracta
- menos propensa a errores

Nota: se requiere más trabajo para que este código compile!

Conclusión

- Necesidad de hacer cambios a Java para simplificar la codificación que pueda ser paralelizada
- Las expresiones lambda simplifican el paso de comportamiento utilizando parámetros



SINTAXIS

Expresiones lambda SON Funciones Anónimas

- Función anónima: son semejantes a los métodos pero sin clase (parámetros) -> { cuerpo-lambda }
- El cuerpo de la expresión lambda puede generar excepciones
- Las lambdas de una única línea
 - *No necesitan llaves*
 - *no requieren una instrucción de retorno explícita*
- las lambdas con un único parámetro no requieren paréntesis
- lambdas sin parámetros deben tener paréntesis

Ejemplos: sintaxis expresiones lambda

- `() -> System.out.println("Hello Lambda")`
- `x -> x + 10`
- `(int x, int y) -> { return x + y; }`
- `(String x, String y) -> x.length() - y.length()`
- `(String x) -> {
 listA.add(x);
 listB.remove(x);
 return listB.size();
}`

Expresiones lambda: Intferencia de tipo

■ Definición de método

- *static<T> T process (List <T> L , Comparator<T> C)*

■ Uso del método

- *List <String> list = getList();*
- *process (list, (String x, String y) -> x.length() - y.length());*

■ El compilador es astuto

- *String r = process(list, (x, y) -> x.length() - y.length())*

Conclusiones

- La sintaxis de las expresiones lambda es sencillo
 - *Corchetes y llaves son opcionales en ciertas situaciones*
- A menudo no es necesario indicar el tipo de la interfaz
 - *Java sigue siendo un lenguaje fuertemente tipado*



INTERFACES FUNCIONALES Y SU DEFINICIÓN



Tipos de las expresiones lambda

- Una expresión lambda es una función anónima
 - *No está asociada con una clase*
- Java es un lenguaje fuertemente tipado
 - *¿Cuál es el tipo de una expresión lambda?*
- Una expresión lambda puede ser utilizada en cualquier parte en donde el tipo es una interfaz funcional
 - *La expresión lambda proporciona la implementación del método abstracto*

Definición de interfaz funcional


- Una interfaz
- Tiene un único método abstracto
- Antes de JDK 8 esto era obvio
 - *Sólo un método*
- JDK introdujo métodos por defecto
- JDK 8 permite métodos estáticos en las interfaces
- anotación `@FunctionalInterface`

Interfaces funcionales

```
interface FileFilter      { boolean accept(File x); }  
interface ActionListener { void actionPerformed(...); }  
interface Callable<T>    { T call(); }
```

¿La siguiente función es una interfaz funcional?

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




Sí. Sólo tiene un
método abstracto

¿La siguiente función es una interfaz funcional?

@FunctionalInterface

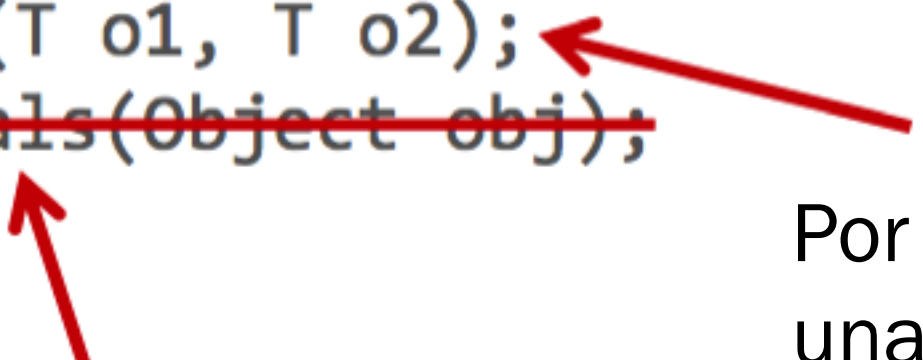
```
public interface Predicate<T> {  
    default Predicate<T> and(Predicate<? super T> p) {...};  
    default Predicate<T> negate() {...};  
    default Predicate<T> or(Predicate<? super T> p) {...};  
    static <T> Predicate<T> isEqual(Object target) {...};  
    boolean test(T t);  
}
```



Sí. Sólo tiene un método abstracto

¿La siguiente función es una interfaz funcional?

```
@FunctionalInterface
public interface Comparator {
    // Static and default methods elided
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```



El método equals(Object)
es implícito de la clase Object

Por lo que sí, es
una interfaz
funcional.
Sólo tiene un
método abstracto

ejemplos de uso de las expresiones lambda

- Variable assignment

```
Callable c = () -> process();
```

- Method parameter

```
new Thread(() -> process()).start();
```

Conclusiones

- Las expresiones lambda pueden ser utilizadas en cualquier parte en donde el tipo es una interfaz funcional
 - *Una interfaz funcional tiene un único método abstracto*
- Las expresiones lambda proporcionan la implementación de un único método abstracto de la interfaz funcional

EL

PACKAGE

JAVA.UTIL.FUNCTIO

N

- The interfaces in this package are general purpose functional interfaces used by the JDK, and are available to be used by user code as well.
- While they do not identify a complete set of function shapes to which lambda expressions might be adapted, they provide enough to cover common requirements.
- Other functional interfaces provided for specific purposes, such as [FileFilter](#), are defined in the packages where they are used.
- The interfaces in this package are annotated with [FunctionalInterface](#). This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.
- Functional interfaces often represent abstract concepts like functions, actions, or predicates. In documenting functional interfaces, or referring to variables typed as functional interfaces, it is common to refer directly to those abstract concepts, for example using "this function" instead of "the function represented by this object". When an API method is said to accept or return a functional interface in this manner, such as "applies the provided function to...", this is understood to mean a *non-null* reference to an object implementing the appropriate functional interface, unless potential nullity is explicitly specified.
-

- The functional interfaces in this package follow an extensible naming convention, as follows:
- There are several basic function shapes, including [Function](#) (unary function from T to R), [Consumer](#) (unary function from T to void), [Predicate](#) (unary function from T to boolean), and [Supplier](#) (nilary function to R).
- Function shapes have a natural arity based on how they are most commonly used. The basic shapes can be modified by an arity prefix to indicate a different arity, such as [BiFunction](#) (binary function from T and U to R).
- There are additional derived function shapes which extend the basic function shapes, including [UnaryOperator](#) (extends Function) and [BinaryOperator](#) (extends BiFunction).
- Type parameters of functional interfaces can be specialized to primitives with additional type prefixes. To specialize the return type for a type that has both generic return type and generic arguments, we prefix ToXxx, as in [ToIntFunction](#). Otherwise, type arguments are specialized left-to-right, as in [DoubleConsumer](#) or [ObjIntConsumer](#). (The type prefix Obj is used to indicate that we don't want to specialize this parameter, but want to move on to the next parameter, as in [ObjIntConsumer](#).) These schemes can be combined, as in [IntToDoubleFunction](#).
- If there are specialization prefixes for all arguments, the arity prefix may be left out (as in [ObjIntConsumer](#)).

Package java.util.pakage

- Bien definido conjunto de interfaces funcionales de propósito general
 - *Todas tienen un único método abstract*
 - *las expresiones lambda pueden ser utilizadas en todos los lugares en donde estos tipos son referenciados*
 - *ampliamente utilizados en la biblioteca de clases de Java*
 - *Especialmente en la API STREAM*
 - *A continuación se describen las interfaces genéricas*
 - Hay numerosas versiones para diferentes tipos
 - Double, Int, Long y Obj

Consumer(T)

- Operación que toma un único valor y no devuelve resultado alguno
- También BiConsumer<T, U> acepta 2 valores y no regresa valor alguno
- Contienen un método para la composición de funciones
 - *andThen(Consumer after)*

```
String s -> System.out.println(s)
```

```
(k, v) -> System.out.println("key:" + k + ", value:" + v)
```

Supplier

Un proveedor de resultados

- Lo opuesto a un consumidor

```
() -> createLogMessage()
```

function<T, R>

Una función que acepta un argumento y proporciona un resultado

- El tipo del argumento y el del resultado pueden ser diferentes
- También la función BiFunction<T, U, R> que acepta 2 parámetros y regresa un resultado
- métodos útiles para realizar la composición
 - *compose, and Then*

```
Student s -> s.getName()
```

```
(String name, Student s) -> new Teacher(name, s)
```

UnaryOperator<T>

- Forma Especializada de una función
- Un único argumento y el resultado es del mismo tipo que el valor devuelto
 - *T apply(T a)*

```
String s -> s.toLowerCase()
```

BinaryOperator<T>

- Forma especializada de BiFuncion
- Dos argumentos y un resultado, todos del mismo tipo
 - *T apply(T a, T b)*

```
(String x, String y) -> {  
    if (x.length() > y.length())  
        return x;  
    return y;  
}
```

Predicado

Función bivaluada booleana

- Toma dos argumentos
- Proporciona métodos estáticos y por defecto útiles para la combinación
 - *and()*, *or()*, *negate()*, *isEqual()*

```
Student s -> s.graduationYear() == 2011
```

```
Files.find(start, maxDepth,  
    (path, attr) -> String.valueOf(path).endsWith(".js") &&  
        attr.size() > 1024,  
    FileVisitOption.FOLLOW_LINKS);
```


Conclusiones

- El paquete functions proporciona una amplia gama e interfaces funcionales
- Ampliamente utilizada en Streams
- Poco probable que necesite definir sus propias extensiones al paquete functions



MÉTODOS Y CONSTRUCTOR



Referencias DE método

- Las referencias de método nos permite reutilizar un método como una expresión lambda

```
FileFilter x = (File f) -> f.canRead();
```



```
FileFilter x = File::canRead;
```

Referencias de método

Más detalles

- Formato: `referencia_objetivo::nombre_metodo`
- tres tipos de referencias de método
 - *Static method*
 - *Instance method of an arbitrary type*
 - *Instance method of an existing object*

Referencias de método

Reglas de construcción

Lambda	<code>(args) -> ClassName.staticMethod(args)</code>
Method Ref	<div><div>↓</div><div>↓</div><code>ClassName::staticMethod</code></div>
Lambda	<code>(arg0, rest) -> arg0.instanceMethod(rest)</code>
Method Ref	<div><div>instanceOf ↓</div><div>↓</div><code>ClassName::instanceMethod</code></div>
Lambda	<code>(args) -> expr.instanceMethod(args)</code>
Method Ref	<div><div>↓</div><div>↓</div><code>expr::instanceMethod</code></div>

Referencias de método

Ejemplos:

Lambda

Method Ref

```
(String s) -> Integer.parseInt(s);
```


Integer::parseInt

Lambda

Method Ref


```
(String s, int i) -> s.substring(i)
```


String::substring

Lambda

Method Ref

```
Axis a -> getLength(a)
```


this::getLength

Referencias de constructores

- Mismo concepto de referencias de método
 - *Para el constructor*

```
Factory<List<String>> f = () -> new ArrayList<String>();
```



```
Factory<List<String>> f = ArrayList<String>::new;
```

Conclusiones

- Las referencias de método proporcionan una notación compacta para las lambdas simples
- Existen tres tipos dependiendo en como son utilizadas
- Pueden también ser utilizadas para constructores

Referenciación a variables externas en las expresiones lambda

Captura de variable local

- Expresiones lambda pueden referenciar a variables locales finales del ámbito
 - *final*: una variable que cumple los requerimientos de las variables
 - Closures sobre los valores no sobre las variables

```
void expire(File root, long before) {  
    root.listFiles(File p -> p.lastModified() <= before);  
}
```

¿Qué significa this en una lambda

- 'this' se refiere al objeto envolvente, no a la misma lambda
- Piense en this como un “final predefined local”
- Recuerde que la lambda es una función anonima
 - *No esta asociada con una clase*
 - *`pr lo tanto no existe un “this” para la lambda*

Referenciación a variables de instancia


Cuales no son finales o efectivamente finales

```
class DataProcessor {  
    private int currentValue;  
  
    public void process() {  
        DataSet myData = myFactory.getDataSet();  
        dataSet.forEach(d -> d.use(currentValue++));  
    }  
}
```

Referenciación a variables de instancia

Las cuales no son finales o efectivamente finales


```
class DataProcessor {  
    private int currentValue;  
  
    public void process() {  
        DataSet myData = myFactory.getDataSet();  
        dataSet.forEach(d -> d.use(this.currentValue++));  
    }  
}
```




'this' (efectivamente final) es insertado por el compilador

Conclusiones

- Las variables en el ámbito circundante pueden ser utilizadas en las expresiones lambda
 - *Pero deben ser final o efectivamente final*
- 'this' en una lambda se refiere al objeto del ámbito circundante
 - *el compilador insertará una referencia a 'this' donde sea necesario*



NUEVOS MÉTODOS ÚTILES EN JDK 8 QUE PUEDEN UTILIZAR LAMBDAS



interfaz iterable

- `Iterable.forEach(Consumer c)`

```
List<String> myList = ...  
myList.forEach(s -> System.out.println(s));
```



```
myList.forEach(System.out::println);
```


Interface Collection

- `Collection.removeIf(Predicate p)`

```
List<String> myList = ...
```

```
myList.removeIf(s -> s.length() == 0);
```

Interface List

- `List.replaceAll(UnaryOperator o)`

```
List<String> myList = ...
```

```
myList.replaceAll(s -> s.toUpperCase());
```



```
myList.replaceAll(String::toUpperCase);
```

Interface List

- `List.sort(Comparator c)`
- Replaces `Collections.sort(List l, Comparator c)`

```
List<String> myList = ...
```

```
myList.sort((x, y) -> x.length() - y.length());
```

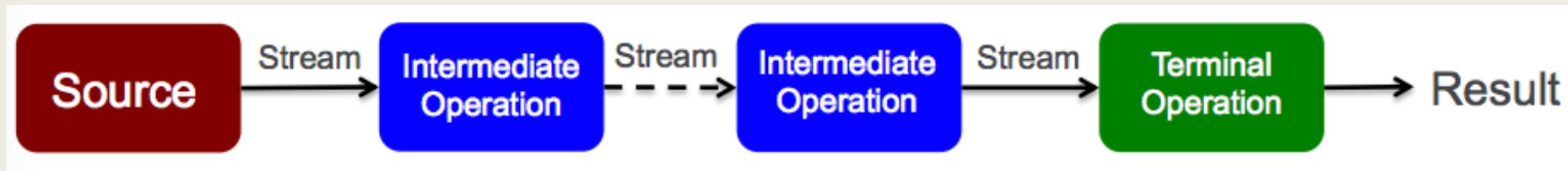
Class Logger

- This is a common problem
 - `logger.finest(createComplexMessage());`
- `createComplexMessage()` is always called, even when not required
 - Heisenberg's Uncertainty Principle in software
- New methods in Logger class
 - Takes a Supplier as an argument (which is a functional interface)
- Simple change to code has big impact on performance
 - `logger.finest(() -> createComplexMessage());`
- We now pass *how* to create the message, not the actual message

Conclusiones

- Utilice los nuevos métodos del JDK 8 para eliminar la necesidad de bucles
- Recuerde que una lambda proporciona comportamiento, no un valor
 - *Útiles para el uso condicional de datos*

STREAM API



Indice

- Introducción a los conceptos de programación funcional
- Elementos de un Stream
- Tipos primitivos de objetos Stream
- Fuentes de Stream en JDK 8
- Interface Stream: Operaciones intermedias
- Interface Stream: Operaciones Terminales
- Clase Optional

Introducción a los conceptos de programación funcional

Programación imperativa

Nombres y valores

- Uso de variables como asociación entre nombres y valores
- Uso de secuencias de comandos
 - *Cada comando consiste de una asignación*
 - *Puede modificar el valor de variables*
 - *Form <nombre_variable> = <expresion>*
 - *Las expresiones pueden referirse a otras variables*
 - Cuyos valores pueden haber sido cambiados por comandos precedentes
 - *Los valores pueden por lo tanto, ser pasados de un comando a otro*
 - *Los comandos pueden ser repetidos en los bucles*

Programación funcional

Nombres y valores

- Basado en llamados a funciones estructuradas
- El llamado a una función llama otras funciones (composición)
- `<function1>(<function2>(<function3> ...> ...)`
- Cada una de las funciones recibe valores de la función que la invoca y a su vez proporciona valores de regreso a dicha función
- Los nombres son utilizados como parametros formales
 - *Una nvez que un valor es asignado este no puede ser modificado*
- no hay concepto de comando, como en el código imperativo
 - *Por lo tanto no existe el concepto de repetición*

Nombres y Valores

- Imperativo
 - *El mismo nombre puede ser asociado con diferentes valores*
- Funcional
 - *Un nombre es asociado con un valor*

Orden de ejecución

■ Imperativo

- *Los valores asociados con nombres pueden ser modificados*
- *El orden de ejecución de comandos establece un contrato*
 - Si es modificado, el comportamiento de la aplicación podría cambiar

■ Funcional

- *Los valores asociados con nombres no pueden ser cambiados*
- *El orden de ejecución no tiene impacto en el resultado*
- *No existe un orden de ejecución preestablecido*

Repetición

■ Imperativo

- *los valores asociados con nombres pueden ser modificados por comandos*
- *los comandos pueden ser repetidos conduciéndonos a cambios repetidos*
- *nuevos valores pueden ser asociados con el mismo nombre a través de repetición (bucles)*

■ Funcional

- *los valores asociados con nombres no deben ser modificados*
- *Cambios repetitivos son logrados anidando llamados a funciones*
- *los nuevos valores pueden ser asociados al mismo nombre a través de la recursión*

Funciones como valores

- La programación funcional, permite que las funciones sean tratadas como valores
 - *Esta es la razón por la que se necesitan las expresiones lambda*
 - *Para lograr que la programación sea mas sencilla que las clases internas anónimas*

Conclusiones

- La programación imperativa y funcional son diferentes
- Imperativa
 - *Los valores asociados con nombres pueden ser modificados*
 - *El orden de ejecución es definido en un contrato*
 - *la repetición es explícita y externa*
- Funcional
 - *Los valores asociados con nombres son establecidos una vez y no pueden ser modificados*
 - *El orden de ejecución no está definido*
 - *La repetición se realiza mediante el uso de la recursión*



ELEMENTOS DE UN STREAM

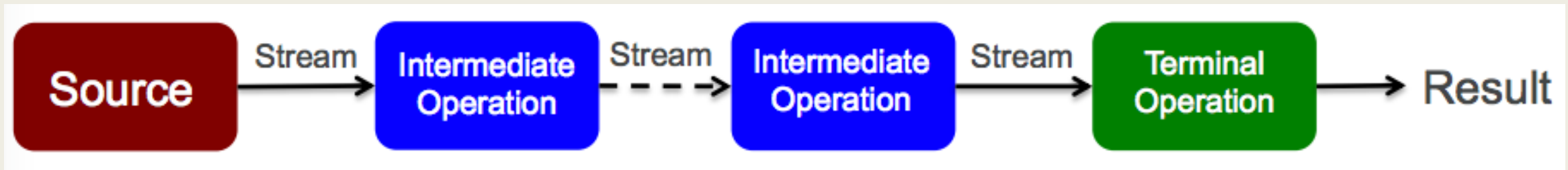


Stream Overview

- Abstracción para especificar cálculos agregados
 - *No es una estructura de datos*
 - *Puede ser infinita*
- Simplifica la descripción de cálculos agregados
 - *Expone oportunidades de optimización*
 - *Fusing, laziness y paralelismo*

Descripción de Stream

- Una tubería de 'stream' consiste de tres tipos de cosas:
 - *Una fuente*
 - *Cero o más operaciones intermedias*
 - *Una operación terminal*
 - Genera un resultado o un efecto secundario



Descripción de Stream

Ejemplo:

Operaciones
intermedias

Fuente

```
int total = transactions.stream()  
    .filter(t -> t.getBuyer().getCity().equals("London"))  
    .mapToInt(Transaction::getPrice)  
    .sum();
```

Operación terminal

Operaciones terminales

- La tubería es evaluado cuando se invoca la operación terminal
 - *Todas las operaciones pueden ser ejecutadas secuencialmente o en paralelo*
 - *las operaciones intermedias pueden ser unidas*
 - Evitando pases redundantes sobre los datos
 - Operaciones de corto-circuito (ejemplo findFirst)
 - Evaluación perezosa Lazy

Conclusiones

- Considere al Stream como una tubería
- Procesamiento de datos proporcionados por la fuente
 - *No hay uso explícito de bucles*
 - *lo cual implica que un Stream puede ser creado en paralelo*

- Abstracción para especificar cálculos agregados
 - *no es una estructura de datos*
 - *puede ser infinita*
- Simplifica la descripción de áculos agregados
 - *Expone oportunidades para su optimización*
 - *Fusing, Laziness y paralelismo*



STREAMS DE OBJETOS Y TIPOS PRIMITIVOS



Objetos y primitivas

Resumen

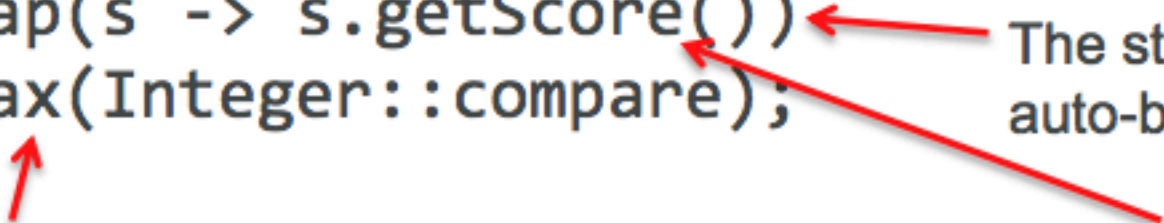
- El lenguaje Java no es realmente orientado a objetos
- Se incluyen los tipos primitivos
 - *byte, short, int, long double, float char*
- En algunas situaciones se encapsulan como objetos
 - *Por ejemplo en colecciones*
 - *Byte, Short, Integer, ...*
- La conversión entre tipos primitivos y representación de objetos es muy a menudo hecho por el auto-boxing y unboxing

Objetos Stream

- By default, a stream produces elements that are objects
- Sometimes, this is not the best solution

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .map(s -> s.getScore())  
    .max(Integer::compare);
```

The stream from map has to auto-box ints to objects




max() debe desempacar
(unbox) cada objeto Integer
para obtener el valor

getScore() devuelve un tipo primitivo (int)

Streams Primitivos

- Para evitar la creación innecesaria de objetos, se cuenta con tres tipos primitivos de stream:
 - *IntStream*, *DoubleStream* y *LongStream*
- Estos pueden ser utilizados con ciertas operaciones

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .mapToInt(s -> s.getScore())  
    .max();
```



El stream que devuelve mapToInt es un stream de valores tipo int,. No se requiere boxing and unboxing

Conclusiones

- El lenguaje Java tiene valores de tipos primitivos así como tipos de objetos
- Para mejorar la eficiencia de stream tenemos tres tipos de primitivas stream
 - *IntStream, DoubleStream y LongStream*
- Utilice métodos tales como `mapToInt()`, `mapToDouble()` y `mapToLong()`

Fuentes de Stream en JDK 8

Bibliotecas JDK 8

- Existen 95 métodos en 23 clases que regresan un Stream
- 71 métodos en 15 clases pueden ser utilizadas como fuentes Stream prácticas

Interface Collection

- `stream()`
 - *Proporciona un stream secuencial de elementos en la colección*
- `parallelStream()`
 - *Proporciona un Stream paralelo de elementos en la colección*
 - *Utiliza el marco de trabajo fork-join*

Clases de Array

- `stream()`
 - *Un array es una colección de datos. así que es capaz de crear un stream*
 - *Proporciona un stream secuencial*
 - *métodos sobrecargados para diferentes tipos*
 - `double, int, long, Object`

Clases File

- `find(Path, BiPredicate, FileVisitOption)`
 - *Un Stream de tipo File referencia que coinciden un BiPredicado dado*
- `list(Path)`
 - *Un Stream de entradas de un directorio dado*
- `lines(Path)`
 - *Un Stream de strings que son las líneas leídas de un fichero dado*
- `walk(Path, FileVisitOption)`
 - *Un stream de referencias de tipo File*

Números aleatorios

- Tres clases relacionadas
 - *Random, ThreadLocalRandom, SplittableRandom*
- Métodos para producir streams finitos o infinitos de números aleatorios
 - *ints(), doubles(), longs()*
 - *cuatro versiones de cada una*
 - Finito o infonito
 - Con y sin semilla

Clases misceláneas y métodos

- `JarFile/ZipFile: stream ()`
 - *Regresa un stream de tipo `File` con el contenido de los ficheros comprimidos*
- `BufferedReader: Lines()`
 - *Regresa unStream de strings que son las. líneas leídas de la entrada*
- `Pattern; splitAsStream()`
 - *Regresa un stream de strings de coincidencias de un patrón*
 - *Al igual que `split()`, pero regresa un stream en vez de un array*

Clases misceláneas y métodos

- CharSequence

- *chars(): Char* regresa una secuencia de valores de tipo *int*
- *codePoints(): Code* apunta a valores de esta secuencia

- BitSet

- *stream():* indices de bits que son establecidos

Métodos estáticos

IntStreams, DoubleStream, LongStream

- Estas interfaces son primitivas especializadas de la interfaz Stream
- `concat(Stream, Stream)`, `empty()`
 - *Concatena dos streams especializadas, regresa un stream vacío*
- `of(T... values)`
 - *Un stream que consiste en los valores especificados*
- `range(int, int)`, `rangeClosed(int, int)`
 - *Un stream desde inicio hasta fin (exclusivo e inclusivo)*
- `generate(IntSupplier)`, `iterate(int, IntUnaryOperator)`
 - *Un stream infinito creado por un proveedor dado*
 - *iterate() utiliza una semilla para inicializar el stream*

Conclusiones

- Muchos lugares para obtener stream que son fuentes
 - *Métodos útiles para recuperar líneas procedentes de ficheros, ficheros de archivos, etc.*
- Solo la Collection puede proporcionar un stream paralelo de forma directa



STREAM INTERFACE: INTERMEDIATE OPERATIONS



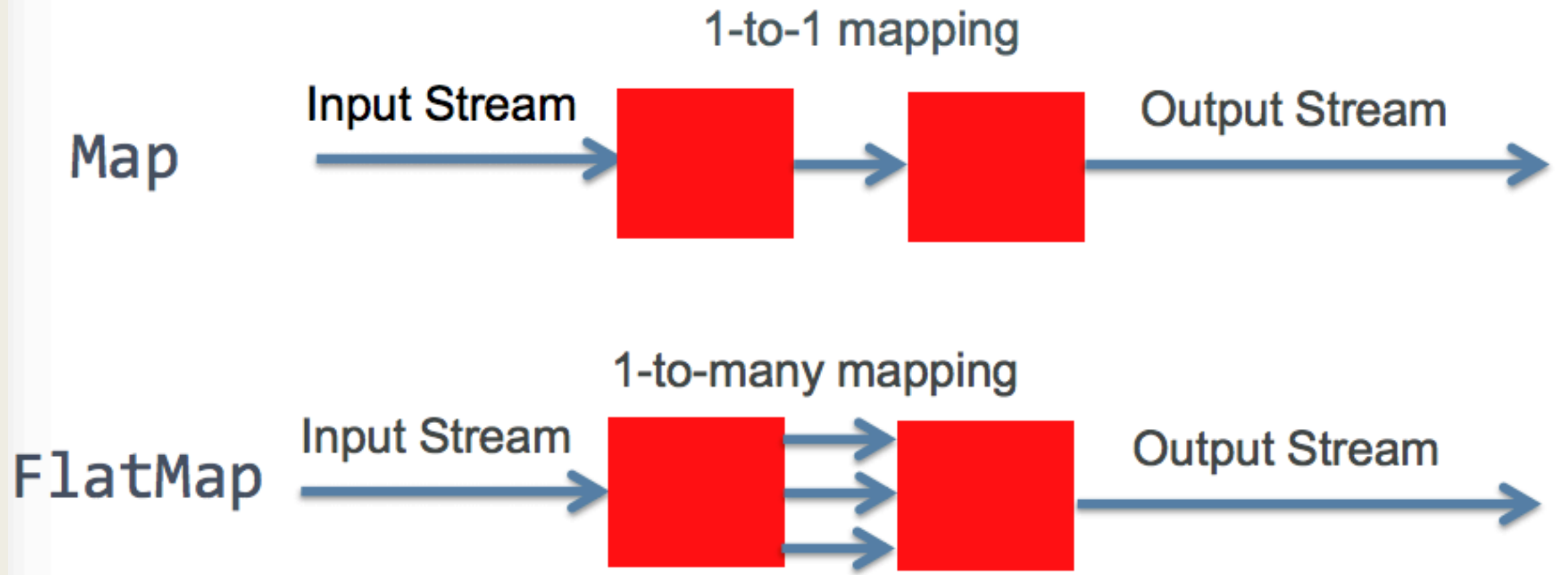
Stream Interface

- Un Stream proporciona una secuencia de elementos
 - *Soportan operaciones de agregado secuenciales o en paralelo*
- La mayoría de la operaciones requieren un parámetro que describe su comportamiento
 - *En general utilizando expresiones lambda*
 - *La mayoría no modifican el stream (Non-Interfering)*
 - *En gneral son sin estado*
- Los streams pueden ser modificados de secuencial a paralelo (y viceversa)
 - *Todo el procesamiento es realizado ya sea secuencialmente o en paralelo*

Filtrado y Mapeo

- `distinct()`
 - *Regresa un flujo sin elementos duplicados*
- `filter(Predicado p)`
 - *Regresa un flujo con aquellos elementos que cumplen el predicado*
- `map(Function p)`
 - *Regresa un stream en donde la función proporcionada es aplicada a cada uno de los elementos del stream*
- `mapToInt(), mapToDouble(), mapToLong()`
 - *Funcionan como `map()` pero producen streams de primitivas es vez de objetos*

Maps and FlatMaps



Ejemplo de FlatMap

Palabras en un fichero

```
List<String> output = reader
    .lines()
    .flatMap(line -> Stream.of(line.split(REGEXP)))
    .filter(word -> word.length() > 0)
    .collect(Collectors.toList());
```

Restringiendo el tamaño de un Stream

- skip(long n)
 - *Regresa un stream que excluye los primeros n elementos del stream de entrada*
- limit(long n)
 - *Regresa un stream que unicamente contiene los primeros n elementos del stream de entrada*

```
String output = bufferedReader  
    .lines()  
    .skip(2)  
    .limit(2)  
    .collect(Collectors.joining());
```

Ordenación y desordenación

A medida que van pasando

- `sorted(Comparator c)`
 - *Regresa un stream que es ordenado, el comparador determina la ordenación*
 - *`sorted()` sin argumentos ordena en base a una ordenación natural*
- `unordered()`
 - *Heredado de `BaseStream`*
 - *Regresa un stream que está desordenado*
 - *Puede mejorar la eficiencia de operaciones tales como `distinct()` y `groupingBy()`*

Orservación de elementos de Stream

- `peek(Consumer c)`
 - *Regresa un stream de salida que es idéntico al input de entrada*
 - *Cada uno de los elemento es pasado por el método `accept()` de Consumer*
 - *El Consumer no debe modificar los elementos del stream*
 - *Es útil para depuración*

Conclusiones

- La interfaz Stream representa operaciones de agregado sobre elementos
- La mayoría de los métodos pueden utilizar expresiones Lambda para definir comportamiento
- Poderosas operaciones intermedias que permite manipular los streams
 - *Se pueden construir procesamientos complejos a partir de bloque de construcción simples*

A thick black L-shaped frame is positioned on the left and bottom edges of the slide, framing the central text.

INTERFAZ STREAM: OPERACIONES TERMINALES

Operaciones Terminales

- Finaliza la tubería de operaciones sobre el stream
- Únicamente hasta este punto se realizan los procesamientos
 - *Esto permite la optimización de la tubería*
 - *Evaluación perezosa (Lazy)*
 - *Operaciones de fusionado o de fusionado*
 - *Eliminación de operaciones redundantes*
 - *Ejecución en paralelo*
- Genera un resultado explícito o un efecto secundario

Elementos coincidentes

- `findFirst()`
 - *La primer coincidencia*
- `findAny()`
 - *Trabaja igual que `findFirst()`, pero actúa sobre un stream paralelo*
 - *`boolean allMatch(Predicate p)`*
 - Si todos los elementos del stream coinciden utilizando el Predicado `p`
 - *`boolean anyMatch(Predicate p)`*
 - Si al menos uno de los elementos coincide con la condición del predicado `p`
 - *`boolean noneMatch(Predicate p)`*
 - ninguno de los elementos coincide de acuerdo al predicado

Recolección de resultados

A medida que van pasando

- `collect(Collector c)`
 - *Realiza una reducción mutable sobre el stream*
- `toArray()`
 - *regresa un array que contiene los elementos del stream*

Resultados Numéricos

A medida que van pasando

- `count()`
 - *Regresa el numero de elementos que hay en el stream*
- `max (Comparator c)`
 - *El elemento con el máximo valor que está dentro del stream en base al comparador*
- `min(Comparador c)`
 - *El elemento con el valor más pequeño que está dentro del stream en base al comparador*
 - *Regresa un Optional, si el stream esta vacío*

Resultados Numéricos

Streams de tipo primitivo (IntStream, DoubleStream(), LongStream)

- `average()`
 - *Regresa la media aritmética del stream*
 - *Si el stream está vacío. regresa un Optional*
- `sum()`
 - *Regresa la suma de los elementos del stream*

Iteración

- `forEach(Consumer c)`
 - *Realiza una acción por cada elemento del stream*
- `forEachOrdered(Consumer c)`
 - *Funciona igual que `forEach`, pero asegura que el orden de los elementos (Si hay elementos) es respetado cuando se utiliza por un stream paralelo*
- Utilízelo con precaución
 - *Alienta el estilo de programación no-funcional (imperativo)*

Plegando una secuencia

Creción de un resultado a partir de varios elementos de entrada

- `reduce(BinaryOperator accumulator)`
 - *realiza una reducción utilizando el Operador Binario*
 - *El acumulador toma un resultado parcial y el elemento next, y regresa un nuevo resultado parcial*
 - *Regresa un Optional*
 - *Dos versiones*
 - Una que toma un valor inicial (no regresa Optional)
 - Una que toma un valor inicial y una BiFuncion (equivalente a un fused map and reduce)

Conclusiones

- Las operaciones de tipo Terminal proporcionan resultados o efectos secundarios
- Se dispone de muchos tipos de operación
- Los del tipo reduce and collect necesitan ser observados con más detalle



LA CLASE OPTIONAL



Problemas de null

- Ciertas situaciones en Java regresan como resultado un valor null
 - *El cual referencia a un objeto que no ha sido inicializado*

Evitar NullPointerException

```
String direction = gpsData.getPosition().getLatitude().getDirection();
```

```
String direction = "UNKNOWN";
```

```
if (gpsData != null) {  
    Position p = gpsData.getPosition();  
  
    if (p != null) {  
        Latitude latitude = p.getLatitude();  
  
        if (latitude != null)  
            direction = latitude.getDirection();  
    }  
}
```

Clase Optional

Ayuda a eliminar la excepción NullPointerException

- Las operaciones terminales como `min()` , `max()` podrían no poder regresar un resultado directo
 - *Suponga que el stream de entrada está vacío*
- `Optional <T>`
 - *Contenedor para una referencia de objeto (null o objeto real)*
 - *Considere como un stream con 0 o 1 elemento*
 - *Garantizado que la referencia Optional regresada no es null*

Optional ifPresent()

Haz algo cuando ha sido establecido

```
if (x != null) {  
    print(x);  
}
```

```
opt.ifPresent(x -> print(x));  
opt.ifPresent(this::print);
```

Optional filter()

rechaza ciertos valores del Optional

```
if (x != null && x.contains("a")) {  
    print(x);  
}
```

```
opt.filter(x -> x.contains("a"))  
    .ifPresent(this::print);
```

Optional map()

Transforma un valor si lo hay

```
if (x != null) {  
    String t = x.trim();  
    if (t.length() > 0)  
        print(t);  
}
```

```
opt.map(String::trim)  
    .filter(t -> t.length() > 0)  
    .ifPresent(this::print);
```

Optional flatMap()

En profundidad

```
public String findSimilar(String s)

Optional<String> tryFindSimilar(String s)

Optional<Optional<String>> bad = opt.map(this::tryFindSimilar);
Optional<String> similar = opt.flatMap(this::tryFindSimilar);
```

Actualizar código del GPS

```
class GPSData {  
    public Optional<Position> getPosition() { ... }  
}  
  
class Position {  
    public Optional<Latitude> getLatitude() { ... }  
}  
  
class Latitude {  
    public String getDirection() { ... }  
}
```


Actualizar el código GPS

getPosition y
getLatitude regresa
un Optional

getDirection
regresa un String

```
String direction = Optional  
    .ofNullable(gpsData)  
    .flatMap(GPSData::getPosition)  
    .flatMap(Position::getLatitude)  
    .map(Latitude::getDirection)  
    .orElse("None");
```

Crea un objeto
Optional con una
referencia que
podría ser null

si getDirection regresa un null regresa
"None", de otra forma regresa la dirección

Conclusiones

- La clase Optional elimina los problemas de NullPointerException
- Puede ser utilizado de diferentes formas para proporcionar el manejo condicional complejo

Conclusiones de la Introducción a Stream

- Los Streams proporcionan una forma directa para el estilo de programación funcional en Java
- Los streams pueden ser objetos o objetos de tipo primitivo
- Un stream consiste de una fuente, posibles operaciones intermedias y una operación terminal
 - *Ciertas operaciones de tipo Terminal regresan un Objeto Optional para evitar posibles problemas generados por NullPointerExceptions*

The image features two large, thick, black L-shaped brackets. One is positioned on the left side, with its vertical bar extending downwards and its horizontal bar extending to the right. The other is on the right side, with its vertical bar extending upwards and its horizontal bar extending to the left. These brackets frame the central text.

CONCEPTOS AVANZADOS DE LAMBDA Y STREAM

Índice

- Comprender el uso de reducciones
- Streams finitos e infinitos
- Evitar el uso del método forEach
- Uso de collectors
- Streams paralelo (y cuando no utilizarlos)
- Depuración de streams y lambdas
- Conclusiones del curso

Problema sencillo

- Encontrar la longitud de la línea de mayor longitud de un fichero

```
Path input = Paths.get("lines.txt");

int longestLineLength = Files.lines(input)
    .mapToInt(String::length)
    .max()
    .getAsInt();
```

Otro problema sencillo

- Encontrar ~~la longitud~~ de la línea de mayor longitud de un fichero

Solución ingenua

```
String longest = Files.lines(input).  
    sort((x, y) -> y.length() - x.length()).  
    findFirst().  
    get();
```

- Esto resuelve el problema
- No exactamente. Ficheros de gran tamaño necesitan muchos recursos y gran tiempo de ejecución
- Debe existir una mejor alternativa

Solucion con iteración externa

```
String longest = "";  
String s;  
while ((s = reader.readLine()) != null)  
    if (s.length() > longest.length())  
        longest = s;
```

- Simple, pero inherentemente serial
- No es thread-safe
- No utiliza el paradigma de programación funcional

Solución recursiva: El método

```
String findLongestString(String s, int index, List<String> l) {  
    if (index >= l.size())  
        return s;  
  
    if (index == l.size() - 1) {  
        if (s.length() > l.get(index).length())  
            return s;  
        return l.get(index);  
    }  
  
    String s2 = findLongestString(l.get(index), index + 1, l);  
  
    if (s.length() > s2.length())  
        return s;  
    return s2;  
}
```

Resolviendo el problema

```
List<String> lines = new ArrayList<>();  
String s;  
while ((s = reader.readLine()) != null)  
    lines.add(s);  
  
String longest = findLongestString("", 0, lines);
```

- No hay bucle explícito, no hay estado mutable, ahora tenemos una solución funcional
- Desafortunadamente no es una solución útil
 - *Conjuntos de datos grandes generara una excepción OOM*

Una solución utilizando Stream

- La API stream utiliza el patrón bien definido **filter-map-reduce**
- En este caso no necesitamos filtrar datos o realizar transformaciones, únicamente reducción
- Recordemos la definición del método reduce
- `Optional<T> reduce (BinaryOperator<T> accumulator)`
- La clave está en utilizar el acumulador correcto
 - *De nuevo, recuerde que el acumulador toma un resultado parcial y el elemento next, y regresa un nuevo resultado parcial*
 - *En esencia esto hace lo mismo que nuestra solución recursiva*
 - *Sin el uso de stack frames*

Una solución utilizando Stream


- Utiliza el enfoque recursivo como un acumulador para una reducción

```
String longestLine = Files.lines(input)
    .reduce((x, y) -> {
        if (x.length() > y.length())
            return x;
        return y;
    })
    .get();
```

Una solución utilizando Stream

- Utiliza el enfoque recursivo como un acumulador para una reducción

```
String longestLine = Files.lines(input)
    .reduce((x, y) -> {
        if (x.length() > y.length())
            return x;
        return y;
    })
    .get();
```



en efecto **x**, mantiene el estado por nosotros, manteniendo siempre la cadena más grande encontrada

La solución más sencilla con Stream

- Uso de la forma especializada de max()
- El que tiene un Comparador como parámetro

```
Files.lines(input)
    .max(comparingInt(String::length))
    .get();
```

- ComparingInt() es un método estático

Comparator<T> comparingInt(ToIntFunction <? extends T> keyExtractor

Conclusiones

- La reducción toma un stream y lo reduce a un valor
- La forma en que la reducción funciona es definida por el acumulador
 - *El cual es un BinaryOperator*
 - *El acumulador es aplicado sucesivamente a los elementos del stream*
 - *El método reduce() mantiene un resultado parcial*
 - *Al igual que un objeto recursivo, pero sin la sobrecarga del recurso*
- Necesita que usted cambie de forma de pensar: no utilice el enfoque basado en bucles del enfoque imperativo



STREAMS FINITOS E INFINITOS



Lidiando con indeterminados

Java Imperativo

- ¿Cómo continuar procesando cuando no podemos predecir cuanto falta por procesar?

```
while (true) {  
    doSomeProcessing();  
  
    if (someCriteriaIsTrue())  
        break;  
  
    // Loop repeats indefinitely  
}
```

Uso de Streams infinitos

Haciendo el Stream finito

- Finaliza el Stream cuando un elemento es leído del stream de entrada
 - *findFirst()*
 - *findAny()*

```
OptionalInt r = Random.ints()  
    .filter(i -> i > 256)  
    .findFirst();
```

Infinite stream of random integers

stream terminates when a number greater than 256 is encountered

Uso de Streams infinitos

Manteniendolo infinito

- Algunas veces necesitamos continuar utilizando un stream indefinidamente
- ¿Qué operación terminal deberíamos utilizar en este caso?
 - *Use forEach()*
 - *Este consume el elemento del stream*
 - *Pero no lo termina*

Uso de Streams infinitos

Ejemplo infinito

- Lectura de temperatura procedente de un sensor serie
 - *Convertir de grados Farenheit a grados Celsius, quitar el simbolo F*
 - *Notificar a un listener de los cambios si este ha sido registrado*

```
thermalReader.lines()
    .mapToDouble(s ->
        Double.parseDouble(s.substring(0, s.length() - 1)))
    .map(t -> ((t - 32) * 5 / 9))
    .filter(t -> !currentTemperature.equals(t))
    .peek(t -> listener.ifPresent(l -> l.temperatureChanged(t)))
    .forEach(t -> currentTemperature.set(t));
```

Conclusiones

- Los Streams pueden ser finitos o infinitos
- No hay concepto de 'breaking' out de un stream
- Utilizar la operación terminal apropiada para detener el procesamiento
- Utilización infinita de un stream infinito



EVITAR EL USO DE
FOREACH

Uso efectivo de los Streams


Pare de pensar de forma imperativa

- La programación imperativa utiliza bucles para comportamiento repetitivo
- También utiliza variables para mantener el estado
- Podemos continuar haciendolo de alguna forma con streams
- ESTO ES ERRONEO

Ejemplo de Stream

Todavía pensando de forma imperativa

```
List<Transactions> transactions = ...  
  
LongAdder transactionTotal = new LongAdder();  
  
transactions.stream()  
    .forEach(t -> transactionTotal.add(t.getValue()));  
  
long total = transactionTotal.sum();
```

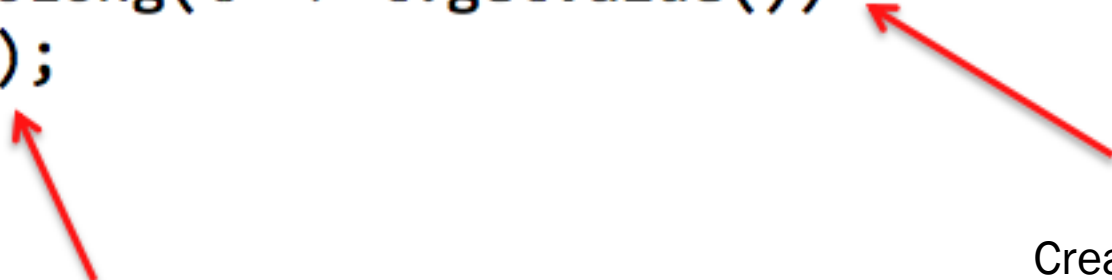


Estamos modificando el estado lo cual es erróneo para un enfoque funcional

Ejemplo de Stream

Ahora utilizaremos el enfoque funcional correcto

```
List<Transactions> transactions = ...  
  
long total = transactions.stream()  
    .mapToLong(t -> t.getValue())  
    .sum();
```



Utiliza una reducción para crear un único resultado

Crea un Stream de valores tipo long que son pasados a la siguiente función

Uso legitimado de forEach

Ningun estado has modificado

- Iteración simplificada
- Podría realizarse de forma paralela si el orden no es importante

```
List<Transactions> transactions = ...  
  
transactions.stream()  
    .forEach(t -> t.printClientName());
```

Conclusiones

- Si esta pensando en utilizar `forEach()`, detengase
- ¿Puede ser reemplazado con una combinación de transformación y reducción?
 - *Si es así, es poco probable que sea el enfoque correcto para ser funcional*
- Ciertas situaciones son validas para utilizar `forEach()`
 - *Ejemplo imprimir valores del Stream*



USO DE COLLECTORS



Fundamento de los collectors

- Un collector realiza una reducción mutable sobre un stream
 - *Acumula los elementos del stream entrada en un contenedor resultado mutable*
 - *El contenedor de resultados puede ser: List, Map, String, ...*
- Utiliza el método Collect() para terminar el flujo
- La clase Collectors utility tiene muchos métodos que pueden crear un collector

Composición de Collectors

- Varios de los métodos Collectors tienen versiones con un collector descendente
- Permiten el uso de un segundo collector
 - *collectingAndThen()*
 - *groupingBy()/groupingByConcurrent()*
 - *mapping()*
 - *partitioningBy()*

Collecting into a Collection

- `toCollection(Supplier factory)`
 - *Agrega los elementos del stream a una Collection (creada utilizando una factoria)*
- `toList()`
 - *Agrega los elementos del stream a una lista*
- `toSet()`
 - *Agrega los elementos del stream a un conjunto(Set)*
 - *Elimina duplicados*

Collecting a un Map

- `toMap(Function keyMapper, Function valueMapper)`
 - *Crea un Map a partir de los elementos del stream*
 - *la llave y el valor son generados utilizando las funciones proporcionadas*
 - *Utiliza `Functions.identity()` para obtener el elemento stream*

```
Map<Student, Double> studentToScore = students.stream()  
    .collect(toMap(Functions.identity(),  
        student -> getScore(student)));
```

Collecting a un Map

Gestionando las llaves duplicadaa

```
toMap(Function keyMapper, Function valueMapper,  
      BinaryOperator merge)
```

- El mismo proceso que el primer método toMap()
 - Pero utiliza el BinaryOperator para fusionar valores para una clave duplicada

```
Map<String, String> occupants = people.stream()  
    .collect(toMap(Person::getAddress,  
                  Person::getName,  
                  (x, y) -> x + "," + y));
```

Las personas con la misma dirección
son fusionadas en un string separado
por coma (CSV)



Agrupación de resultados

- `groupBy(Function)`
 - *Agrupar elementos de un stream utilizando la function en un Map*
 - *Resultado es un `Map<K, List<V>>`*
 - *`Map m = words.stream()`*
 - *`.collect`*
 - *`groupBy(Function, Collector)`*
 - Agrupar los elementos del stream utilizando la Function
 - Se realiza una reducción sobre cada grupo utilizando el Collector descendente

```
Map m = words.stream()  
    .collect(Collectors.groupBy(String::length, counting()));
```

Concatenación de resultados

- `joining()`
 - *Collector que concatena strings de entrada*
- `joining(delimiter)`
 - *Collector concatena los fluos de cadenas utilizando el delimitador CharSequence*

```
collect(Collectors.joining(", ")); // Create CSV
```

- `joining(delimiter, prefix, suffix)`
 - *Collector concatena el prefijo, las cadenas del stream seperados por el delimitador y el sufijo*

Collectores numéricos

También disponible en Formas Double y Long

- `averagingInt(ToIntFunction)`
 - Promedia los resultados generados por la función proporcionada
- `summarizingInt(ToIntFunction)`
 - Resume (count, sum, min, max, average) los resultados generados por la función proporcionada
- `summingInt(ToIntFunction)`
 - equivalente a `map()` then `sum()`
- `maxBy(Comparator)`, `minBy(Comparator)`
 - Valor máximo o mínimo en base al comparador


Otros Collectores

- `reducing(BinaryOperator)`
 - *Collector Equivalente a la moperacion terminal `reduce()`*
 - *Sólo utilizado para reducciones multi-nivel o colectores descendentes*
- `partitioningBy(Predicate)`
 - *Crea un `Map<Boolean, List>` que contiene 2 grupos basados en un Predicado*
- `mapping(Function, Collector)`
 - *Adapta un Collector para aceptar diferentes tipos de elementos mapeados por la Function*


```
Map<City, Set<String>> lastNamesByCity = people.stream()
    .collect(groupingBy(Person::getCity,
        mapping(Person::getLastName, toSet())));
```

Conclusiones

- Los Collectores proporcionan una forma potente de obtener elementos de un stream de entrada
 - *En collections*
 - *En formas numericas como totales y promedios*
- Los collectors pueden ser creados para construir collectors mas complejos
- Usted puede crear sus propios Collectors



STREAMS PARALELO
(CUANDO NO
UTILIZARLOS)



Stream seriales y en paralelo

- fuentes de los stream
 - *stream()*
 - *parallelStream()*
- Stream puede ser hecho paralelo o secuencial en cualquier punto
 - *-parallel()*
 - *-sequential()*
- Vence la última llamada
 - *el stream completo es secuencial o paralelo*
- Llamar a *concat()* con un stream secuencial y uno paralelo producirá un stream paralelo

Streams Paralelos

- Implementado internamente utilizando el marco de trabajo fork-join
- tendrá como valor por defecto tantos hilos de ejecución para el pool como procesadores informa el procesador
 - *el cual podría no ser lo que usted quiere*

```
System.setProperty(  
"java.util.concurrent.ForkJoinPool.common.parallelism",  
"32767");
```

- Recuerde, los streams paralelos siempre necesitan mas trabajo para procesar
 - *pero pueden terminar mas rápido*

Consideraciones para utilizar Stream paralelos

- `findFirst()` y `findAny()`
 - *`findAny()` es no-determinista, así que es más apropiado para mejorar rendimiento de stream paralelo*
 - *utilice `findFirst()` si se necesita un resultado determinista*
- `forEach()` y `forEachOrdered()`
 - *`forEach()` es no-determinista para un stream paralelo y datos ordenados*
 - *Uso `forEachOrdered()` si un resultado determinista es requerido*

¿Cuándo utilizar Streams paralelo?

No existe una respuesta sencilla

- El tamaño del conjunto de datos es importante, así como el tipo de la estructura de datos
 - *ArrayList: GOOD*
 - *HashSet, TreeSet: OK*
 - *LinkedList: BAD*
- Las operaciones son también importantes
 - *Ciertas operaciones se realizan en paralelo mejor que otras*
 - *filter() y map() son excelentes*
 - *sorted() y distinct() no se descomponen bien*

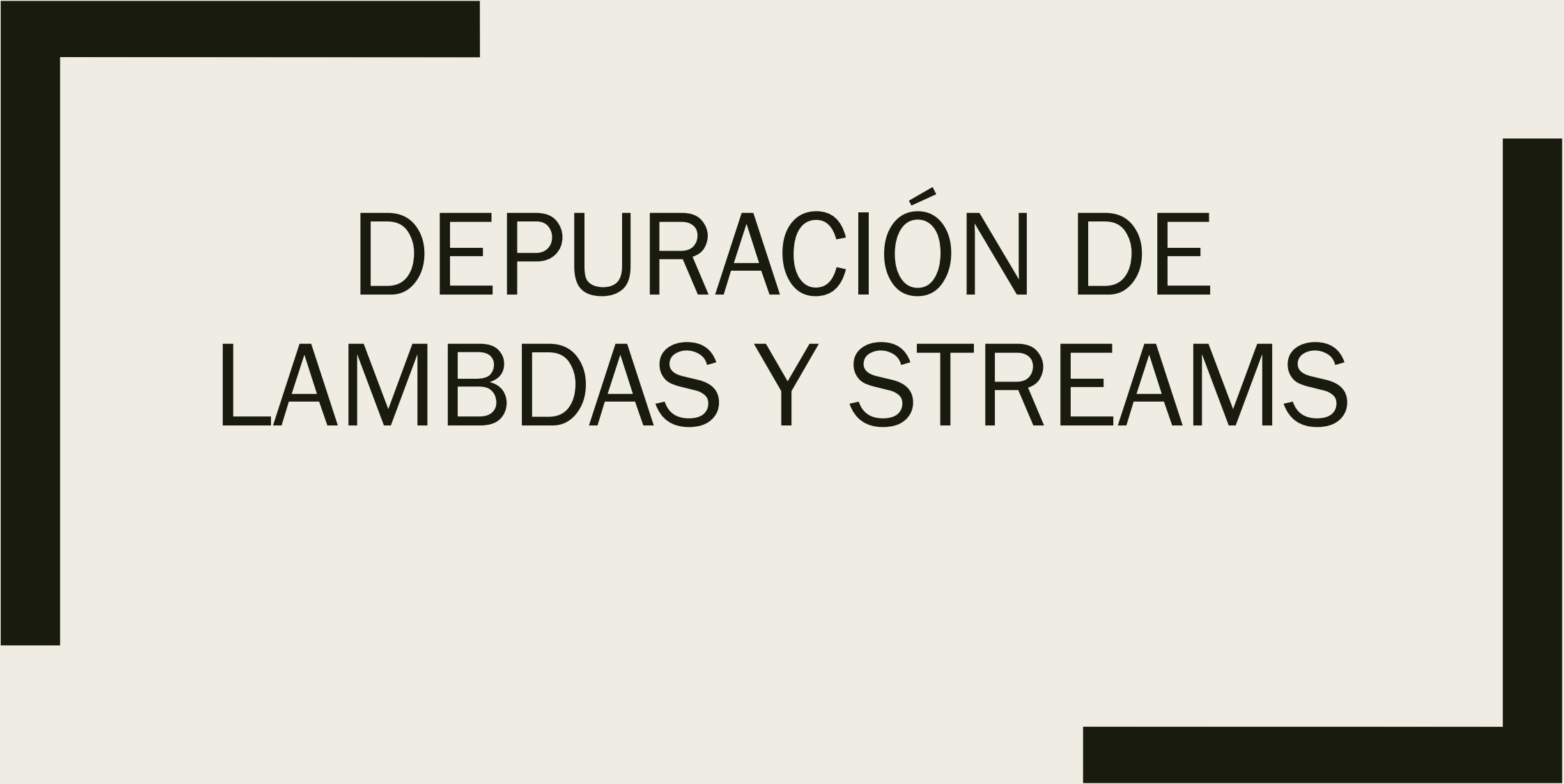
¿Cuándo utilizar Streams paralelo?

Consideraciones cuantitativas

- N = tamaño del conjunto de datos
- Q = Costo por elemento a través de la tubería del stream
- $N \times Q$ = Coste total de las operaciones en la tubería
- entre más grande sea el valor $N \times Q$, el rendimiento del stream paralelo será mejor
- Es más fácil conocer N que Q . pero Q puede ser estimado
- Si duda, profile

Conclusiones

- Los streams pueden ser procesados secuencialmente o en paralelo
 - *Todo el stream es procesado de forma secuencial o en paralelo*
 - *En la mayoría de los casos la forma en la que está definida el stream no afecta al resultado*
 - *findFirst(), findAny(), forEach(), forEachOrdered()*
- No asuma que un stream paralelo proporcionará un resultado más rápido
 - *Muchos factores afectan el rendimiento*

A thick black L-shaped frame is positioned on the left and bottom edges of the slide, framing the central text.

DEPURACIÓN DE LAMBDA Y STREAMS

Problemas con la depuración de Streams

- Los Streams proporcionan un alto nivel de abstracción
 - *Esto es bueno para hacer código claro y fácil de entender*
 - *Esto es malo para depurar*
 - Pasan muchas cosas internamente en el código de la biblioteca
 - Colocar puntos de ruptura no es sencillo
 - Las operaciones son unidas para mejorar eficiencia

Depuración sencilla

Encontrar que está pasando entre métodos

- Utilice peek()
 - *Similar al uso de instrucciones print*

```
List<String> sortedWords = reader.lines()           // Lines from file
    .flatMap(line -> Stream.of(line.split(REGEXP))) // Words from file
    .map(String::toLowerCase)                       // In lower case
    .distinct()                                     // Remove duplicates
    .sort((x, y) -> x.length() - y.length())        // Sort by length
    .collect(Collectors.toList());                  // Collect to list
```

Depuración sencilla

Encontrar que está pasando entre métodos

- Utilice peek()
 - *Similar al uso de instrucciones print*

```
List<String> sortedWords = reader.lines()           // Lines from file
    .peek(System.out::println)                      // Print lines from file
    .flatMap(line -> Stream.of(line.split(REGEXP))) // Words from file
    .map(String::toLowerCase)                       // In lower case
    .distinct()                                     // Remove duplicates
    .sort((x, y) -> x.length() - y.length())        // Sort by length
    .collect(Collectors.toList());                  // Collect to list
```

Depuración sencilla

Encontrar que está pasando entre métodos

- Utilice peek()
 - *Similar al uso de instrucciones print*

```
List<String> sortedWords = reader.lines()           // Lines from file
    .flatMap(line -> Stream.of(line.split(REGEXP))) // Words from file
    .peek(System.out::println)                      // Print words
    .map(String::toLowerCase)                       // In lower case
    .distinct()                                     // Remove duplicates
    .sort((x, y) -> x.length() - y.length())        // Sort by length
    .collect(Collectors.toList());                  // Collect to list
```

Estableciendo un Breakpoint

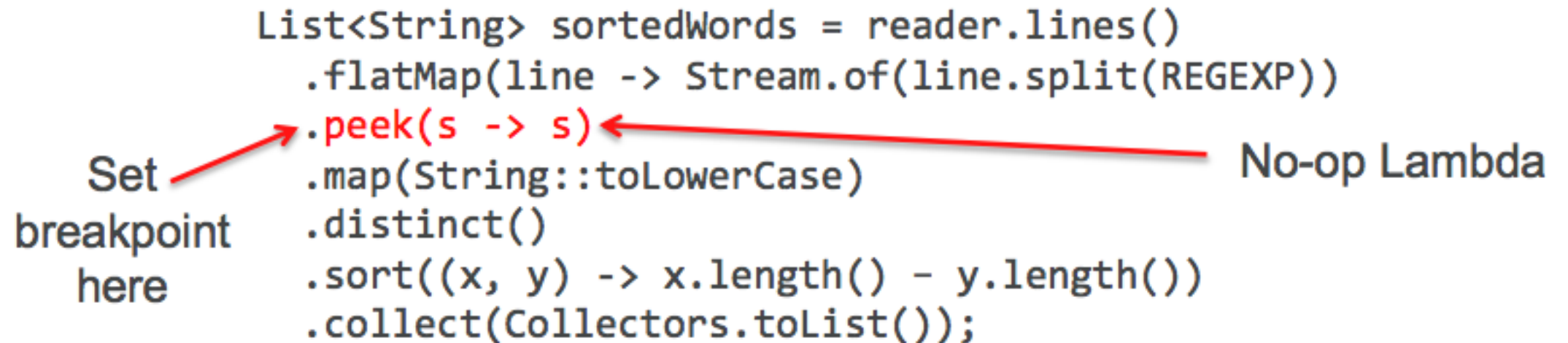
Utilización de peek()

- Agregar una llamada al método peek() entre las operaciones del stream
- Utilizar un Consumer que haga nada en caso necesario
 - *Algunos depuradores necesitan que los cuerpos no estén vacíos*

```
List<String> sortedWords = reader.lines()
    .flatMap(line -> Stream.of(line.split(REGEXP)))
    .peek(s -> s)
    .map(String::toLowerCase)
    .distinct()
    .sort((x, y) -> x.length() - y.length())
    .collect(Collectors.toList());
```

Set breakpoint here

No-op Lambda



Estableciendo un Breakpoint

Utilización de una referencia de método

- Las expresiones Lambda no compila a clases internas equivalentes
 - *Compilada a llamada invocada dinámicamente*
 - *La implementación es decidida en tiempo de ejecución*
 - *Mejores oportunidades de optimización. hace la depuración más difícil*
- Solución:
 - *Extraer el código de la expresión Lamba en un método separado*
 - *Reemplazar la lambda con la referencia a un método para el nuevo método*
 - *Establecer puntos de ruptura en las instrucciones del método nuevo*
 - *Examinar el estado del programa utilizando el depurador*

Conclusiones

- La depuración es difícil con Lambdas y streams
 - *Los métodos Stream son mezclados*
 - *Las lambdas son convertidas a bytecodes invocados de forma dinámica y su implementación se decide en tiempo de ejecución*
 - *Difícil colocar puntos de ruptura*
- Las referencias a métodos y la función `peek()` puede simplificar las cosas

Conclusiones del curso

Expresiones lambda

- Las expresiones lambda proporcionan una forma sencilla de definir comportamiento
 - *Pueden ser asignadas a una variable o ser pasada como un parametro*
- Pueden ser utilizadas en cualquier parte en donde el tipo sea una interfaz funcional
 - *Una que solo tiene un método abstracto*
 - *La expresión lambda proporciona una implementación del método abstracto*

API Stream

- Tubería de operaciones para procesar colecciones de datos
 - *Múltiples fuentes, no solo de la API Collections*
 - *Pueden ser procesados secuencialmente o en paralelo*
- Fuentes, intermediarios y operaciones terminales
- El comportamiento de operaciones intermedias y terminales a menudo son definidas utilizando expresiones lambda
- Las operaciones terminales a menudo regresan un Optional
- Podemos utilizar un estilo de programación funcional con Java

Lambdas y Streams: Piensa diferente

- Es necesario pensar de forma funcional en vez de pensar de forma imperativa
 - *Intenta parar de pensar en bucles y en utilizar estados mutables*
- Piense como enfocar problemas utilizando recursión
 - *En vez de un bucle explicito*
 - *Evita el forEach(excepto en casos especiales: imprimir)*
- Los streams infinitos no necesariamente necesitan ser infinitos
- Recuerde, los stream paralelos siempre involucran mas trabajo
 - *Algunas veces ellos completan el trabajo más rápido*