Introducción a Expresiones Lambda y API Stream en Java SE 8 – Parte 1

Por Alexis Lopez

Publicado en Agosto 2015

Las características más importantes de Java SE 8 son la adición de **Expresiones Lambda** y la **API Stream**. Con la adición de expresiones lambda podemos crear código más conciso y significativo, además de abrir la puerta hacia la programación funcional en Java, en donde las funciones juegan un papel fundamental. Por otro lado, la API Stream nos permite realizar operaciones de tipo filtro/mapeo/reducción sobre colecciones de datos de forma secuencial o paralela y que su implementación sea transparente para el desarrollador. Lambdas y Stream son una combinación muy poderosa que requiere un cambio de paradigma en la forma en la que hemos escrito código Java hasta el momento.

En esta primera parte, describiremos por qué la necesidad de expresiones lambda, su sintaxis y funcionamiento, así como las adiciones y cambios al lenguaje que soportan esta nueva característica.

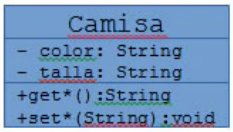
En la segunda parte, mostraremos el uso de la API Stream y la combinación ganadora Lambdas + Stream.

Funciones como entidades de primer nivel

Uno de los conceptos de la programación funcional habla de que las funciones (métodos) sean definidas como entidades de primer nivel, es decir, que puedan aparecer en partes del código donde otras entidades de primer nivel, como valores primitivos u objetos, aparecen. Esto significa poder pasar funciones, en tiempo de ejecución, como valores de variables, valores de retorno o parámetros de otras funciones. Este es un concepto muy poderoso que se puede entender como la posibilidad de pasar comportamiento como valor y es precisamente lo que podemos lograr con la adición de expresiones lambda al lenguaje Java.

Paso de comportamiento como valor/parámetro

Para entender mejor el concepto de funciones como entidades de primer nivel, analicemos el siguiente caso con ayuda de la clase *Camisa* definida a continuación:



Se nos solicita obtener un subconjunto de las camisas de color ROJO, para lo cuál podríamos pensar en crear un método como sigue:

```
public static List<Camisa> filtrarRojas(List<Camisa> inv)  { List<Camisa> sub = new ArrayList<>();
   for(Camisa camisa:  inv){
   if( "ROJO".equals(camisa.getColor()  ) {
sub.add(camisa);
}
   }
   return sub;
   }
```

Pronto sentimos la necesidad de hacer el método más genérico, por lo que adicionamos un segundo parámetro que nos indica el color a filtrar:

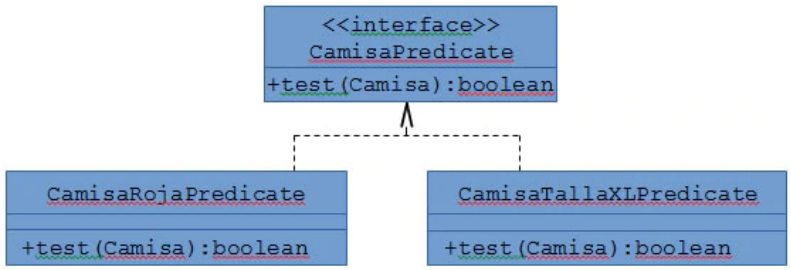
```
public static List<Camisa> filtrar(List<Camisa>  inv, String color) { List<Camisa>  sub = new ArrayList<>();
   for(Camisa camisa:  inv){
   if( camisa.getColor().equals(color) ) {
sub.add(camisa);
}
   }
   return sub;
   }
```

Pero a medida que los requerimientos crecen, y se nos solicita filtrar por otras características de nuestra clase, podríamos caer en el error de crear métodos como el que sigue, que intenta filtrar el listado basado en alguna de las dos características definidas en nuestra clase *Camisa*:

```
public static List<Camisa> filtrar(List<Camisa>  inv, String color, String talla, boolean  bool) {
 List<Camisa> sub  = new ArrayList<>();
 for(Camisa camisa:  inv){
 if( (bool && camisa.getColor().equals(color))  || (!bool && camisa.getTalla().equals(talla))  )
{
sub.add(camisa);
}
 }
 return sub;
 }
```

Es claro que ninguno de los lectores ha escrito código como el anterior *;o)*

El problema se hace más evidente cuando tenemos más de dos características por las cuales filtrar, el código se vuelve inmanejable y muy difícil de mantener. Pero gracias a la programación orientada a objetos y los patrones de diseño, podríamos crear una solución más genérica basada en una jerarquía y haciendo uso del patrón **Estrategia**:



Con el anterior diseño, ahora podemos encapsular nuestro filtro en un objeto de tipo **CamisaPredicate**, el cual tiene un método que permitirá establecer si se cumplen o no nuestras condiciones. Veamos algunos ejemplos de las clases concretas que se pueden definir gracias al diseño anterior:

```
public class CamisaRojaPredicate  implements CamisaPredicate{
     public boolean test(Camisa camisa){
     return “ROJO”.equals(camisa.getColor());
     }
     }
```

```
public class  CamisaTallaXLPredicate implements CamisaPredicate{
     public boolean test(Camisa camisa){
     return “XL”.equals(camisa.getTalla());
     }
     }
```

```
public class CamisaRojaXLPredicate implements  CamisaPredicate{
     public boolean test(Camisa camisa){
     return “ROJO”.equals(camisa.getColor())  && “XL”.equals(camisa.getTalla());
     }
     }
```

Y por lo tanto, en nuestro código podemos definir un método que filtre de una manera muy genérica. Nótese que el siguiente método recibe un comportamiento como parámetro (predicado) que le indica cuando adicionar la camisa al subconjunto:

```
public static List<Camisa> filtrar(List<Camisa>  inv,
     CamisaPredicate predicado) {  List<Camisa> sub = new ArrayList<>();
     for(Camisa camisa:  inv){
     if( predicado.test(camisa) )  {
sub.add(camisa);
}
     }
     return sub;
     }
```

¿Cuál es el inconveniente de esta solución? Dado que está basada en el patrón de diseño Estrategia, presenta el inconveniente de que tendríamos que crear tantas clases como filtros necesitemos, lo cual nos llevaría a repetir mucho código.

Algunos podrán argumentar que no es necesario crear tantas clases como filtros sean necesarios, sino que podríamos usar clases anónimas desde nuestros programas cliente y así reducir la cantidad de clases creadas. Esto es cierto y funciona como se puede ver a continuación:

```
List<Camisa> camisas  = …

CamisaPredicate rojas  = new CamisaPredicate() {
   public boolean test(Camisa camisa){
   return "ROJO".equals(camisa.getColor());
   }
   };

CamisaPredicate rojasXL = new CamisaPredicate() {
   public boolean test(Camisa camisa){
   return “ROJO”.equals(camisa.getColor())  && “XL”.equals(camisa.getTalla());
   }
   };

List<Camisa> camisasRojas = filtrar(camisas,  rojas); List<Camisa> camisasRojasXL  = filtrar(camisas, rojasXL);
```

Aunque esto reduce la cantidad de clases creadas y el paso de comportamiento como parámetro se mantiene, aún se puede detectar mucho código repetido. La evolución del ejemplo anterior, desde nuestros métodos con parámetros para filtrar un listado hasta el paso de comportamiento como parámetro, pero sin código repetido, se muestra a continuación haciendo uso de **expresiones lambda**:

```
List<Camisa> camisas  = …

List<Camisa> camisasRojas = filtrar(camisas,
   (Camisa c) > “ROJO”.equals(c.getColor()));
```

Lo anterior puede parecer bastante confuso en un principio, pero ya revisaremos los conceptos y cambios en el lenguaje que nos permitirán escribir código Java de esa manera.

Expresiones Lambda
Por medio de expresiones lambda podemos referenciar métodos anónimos o métodos sin nombre, lo que nos permite escribir código más claro y conciso que cuando usamos clases anónimas. Una expresión lambda se compone de:

- Listado de parámetros separados por comas y encerrados en paréntesis, por ejemplo: (a,b).
- El símbolo de flecha hacia la derecha: ->
- Un cuerpo que puede ser un bloque de código encerrado entre llaves o una sola expresión.

A continuación algunos ejemplos de expresiones lambda:

```
(int a, int b) >  a + b
(int a) > a +  1
(int  a, int b) > { System.out.println(a + b);  return a + b; } () > new ArrayList()
```

Del bloque de código anterior podemos destacar lo siguiente:

- El primer ejemplo es una expresión lambda que tiene dos (2) parámetros tipo *int* y una sola sentencia que realiza la suma de dichos parámetros
- La segunda expresión tiene un solo parámetro y una sola sentencia que suma una unidad a dicho parámetro
- La tercera expresión cuenta con dos (2) parámetros tipo *int* y tiene un cuerpo con dos (2) sentencias que primero escribe a la consola la suma de ambos parámetros y luego retorna dicho valor. Nótese que cuando tenemos varias sentencias dentro del cuerpo de una expresión lambda, además de encerrar su cuerpo entre llaves ({}) también debemos separar las sentencias por medio de puntos y comas (;)
- La última expresión no tiene parámetros, pero retorna una nueva instancia de la clase *java.util.ArrayList*

Interfaces recargadas
Java SE 8 hace un cambio grande a las interfaces con el fin de que las librerías puedan evolucionar sin perder compatibilidad. A partir de esta versión, las interfaces pueden proveer métodos con una implementación por defecto. Las clases que implementen dichas interfaces heredarán automáticamente la implementación por defecto si éstas no proveen una explícitamente:

- Llamados métodos por defecto, métodos virtuales o métodos defensores , son especificados e implementados en la interface. Usan la nueva palabra reservada *default* antes del tipo de retorno.
- La implementación por defecto es usada solo cuando la clase implementadora no provee su propia implementación .
- Desde el punto de vista de quién invoca al método, es un método más de la interface.
- Conflicto cuando se implementen interfaces con métodos por defecto con el mismo nombre.

En el siguiente ejemplo vemos como se ha adicionado el método por defecto *+sort(Comparator):void* a la interface j*ava.util.List* sin que esto afecte sus implementaciones:

```
Interface List<T> {
    …
    default void sort(Comparator<? super T>  cmp)
    {
    Collections.sort(this, cmp);
    }
    …
}
```

Con la adición de métodos por defecto, las clases ahora pueden heredar diferentes comportamientos de múltiples interfaces. En caso de conflictos, este es el orden en el que se selecciona el método por defecto:

1. Implementaciones en clases concretas
2. Implementaciones en subinterfaces
3. Explícitamente seleccionando el método deseado usando: X.**super**.m(...) , donde X es la interface y m es el método.

A continuación definiremos dos (2) interfaces, ambas definen un método por defecto con el mismo nombre:

```
public interface  SaludoMañanaInterface {
default void saludo(){ System.out.println("Buenos días");
}
}



public interface  SaludoTardeInterface {
default void saludo(){ System.out.println("Buenas tardes");
}
}
```

Dado que la clase concreta implementa ambas interfaces, ésta deberá sobreescribir obligatoriamente el método y decidir cuál de los dos invocar:

```
class MultipleHerencia implements  SaludoMañanaInterface, SaludoTardeInterface {

@Override
public void saludo() {
SaludoMañanaInterface.super.saludo();
}
 }
```

También debemos mencionar que, a partir de Java SE 8, además de métodos por defecto, las interfaces también pueden definir e implementar métodos estáticos. A continuación se muestra uno de los métodos estáticos que ahora existen en la interface *java.util.Comparator*:

```
public interface Comparator<T>  {
   ...
   public static <T  extends Comparable<? super T>>  Comparator<T>  reverseOrder()  {
   return Collections.reverseOrder();
   }
   ...
}
```

Con estos cambios, las librerías podrán evolucionar sin afectar implementaciones actuales, por ejemplo, esta es la forma en la que ha evolucionado el framework de colecciones en esta nueva versión de Java.

Interfaces funcionales
Concepto nuevo en Java SE 8 y que es la base para que podamos escribir expresiones lambda. Una interface funcional se define como una interface que tiene uno y solo un método abstracto y que éste sea diferente a los métodos definidos en java.lang.Object (a saber: equals, hashcode, clone, etc.). La interface puede tener métodos por defecto y estáticos sin que esto afecte su condición de ser interface funcional.

Existe una nueva anotación denominada **@FunctionalInterface** que permite al compilador realizar la validación de que la interface tenga solamente un método abstracto. A continuación se muestra un código que no compilará, debido a que la interface define más de un método abstracto:

```
@FunctionalInterface
   public interface  MiInterface
   {
   int  getNum();
   String  getString(); String toString();
 }
```

Nótese que la interface anterior define tres métodos abstractos, sin embargo *+toString():String* es uno de los métodos definidos en la clase *java.lang.Object* y por lo tanto no cuenta para la regla de interfaces funcionales. Lo cual nos deja con dos métodos abstractos y es por ello que el compilador mostrará el error: "**MiInterface is not a functional interface**"

Java SE 8 define cuatro (4) grandes grupos de interfaces funcionales agrupadas en el paquete *java.util.function*. A continuación veremos las principales de cada grupo:

- **java.util.function.Predicate<T>**: Define el método *+test(T):boolean* y es usada para validación de criterios.
- **java.util.function.Supplier<T>**: Define el método *+get():T* y es usada para la creación de objetos.
- **java.util.function.Consumer<T>**: Define el método *+accept(T):void* y es usada para consumir métodos del parámetro T, causando posibles

efectos secundarios.
 java.util.function.Function<T, R>: Define el método *+apply(T):R* y es usada para convertir de un valor T a otro valor R.

Las interfaces mencionadas anteriormente tienen variantes que invitamos al lector a revisar con mayor detalle en el paquete *java.util.function*.

Inferencia de tipos
Una expresión lambda puede ser usada para crear una instancia de una interface funcional. El tipo de interface funcional es inferida de acuerdo al contexto y funciona tanto en contextos de asignación como en invocación de métodos (parámetros). Además, el compilador infiere el tipo de los parámetros de la expresión lambda basándose en la definición del método abstracto de la interface funcional y por lo tanto no hay necesidad de escribir su tipo:

```
List<Camisa> camisas = …

//Contexto de asignación
     Predicate<String>  p = s > “ROJO”.equals(s);

//Contexto de invocación de  métodos
     List<Camisa> lista  = filtrar(camisas, c > “ROJO”.equals(c.getColor()));
```

Del anterior bloque de código podemos notar que:

 El tipo de interface funcional en el contexto de asignación o de invocación de método es inferido por el compilador basándose en el tipo de variable/parámetro al que se asignará la expresión lambda.

 En el contexto de asignación no es necesario escribir el tipo de parámetro, dado que el compilador sabe que la interface funcional *java.util.function.Predicate<String>* define el método *+test(String):boolean* y por lo tanto infiere que el parámetro es de tipo *String*.

 En el contexto de invocación de métodos no es necesario escribir el tipo de parámetro, dado que el compilador sabe que la interface funcional *CamisaPredicate* (definida para nuestro ejemplo de camisas) define el método *+test(Camisa):boolean* y por lo tanto infiere que el parámetro es de tipo *Camisa*.

De forma general, a continuación se listan los pasos que usa el compilador para inferir el tipo de una expresión lambda:

1. Identificación de contexto: asignación o invocación de método.
2. Identificar el tipo destino: variable o parámetro.
3. Identificar la interface funcional del tipo destino.
4. Identificar el descriptor de la función (firma del método abstracto).
5. Verificar que el descriptor de la función sea coherente con la expresión lambda.

Existen casos especiales que se deben tener en cuenta a la hora de trabajar con expresiones lambda:

1. La misma expresión lambda puede satisfacer diferentes descriptores de funciones, esto es, diferentes interfaces funcionales.

```
//Misma expresión lambda, diferentes interfaces funcionales
 Callable<Integer>  c = () > 42; PrivilegedAction<Integer> p =  () > 42;
```

Es importante notar que si tenemos dos métodos con el mismo nombre pero diferentes interfaces funcionales como parámetro, a las cuáles la misma expresión lambda las satisface, entonces al momento de invocar uno de estos métodos y pasar una expresión lambda como parámetro, el compilador no sabrá a cuál de los dos métodos nos referimos y marcará un error:

```
Public static void  main(String... args){
     metodo(() > 42); //Error de  compilación
 }

public static void metodo(Callable<Integer> p)
     { ... }

public static void metodo(PrivilegedAction<Integer> c)
     { ... }
```

2. Regla de compatibilidad por no retorno. Si una expresión lambda se compone solo de una sentencia, entonces es compatible con descriptores de funciones que no tengan retorno, es decir void.

```
//Regla de compatibilidad  por no retorno
 List<String> list  = ... Predicate<String>  p = s > list.add(s); Consumer<String>  c = s > list.add(s);
```

En el anterior bloque de código sabemos que el método que define la interface funcional *java.util.function.Predicate* retorna un valor *boolean* y que el método que define la interface funcional *java.util.function.Consumer* retorna *void*, sin embargo, dado que la expresión lambda es de sólo una sentencia no hay error de compilación.

Alcance de una expresión lambda
Las expresiones lambda pueden usar variables/parámetros en su interior si éstos han sido definidos como constantes (*final*) o son efectivamente constantes:

Efectivamente Constante: Variable/Parámetro que solo es asignado una vez, incluso si no se ha definido usando la palabra *final*.

Un ejemplo de lo anterior se muestra a continuación, nótese que el parámetro usado dentro de la expresión lambda no está definido como constante, pero tampoco se actualiza en alguna otra parte del método:

```
public static void  alcance(int num)
 {
     List<String> palabras  = ...;
     Predicate<String>  odd = s > s.length()  > num;
     palabras.removeIf(odd);
 }
```

Si en alguna parte del código anterior modificamos el valor del parámetro, recibiríamos un error de compilación como el que sigue: “**local variables referenced from a lambda expression must be final or effectively final**”. Es como si la expresión lambda usara el valor más no la variable.

A diferencia de las clases anónimas, en expresiones lambda la palabra *this* hace referencia a la instancia de la clase sobre la cual se ha escrito la expresión lambda. Recordemos que en clases anónimas, la palabra *this* hace referencia a la clase anónima en sí.

El código a continuación muestra que podemos acceder al atributo de instancia nombrado **before** desde la expresión lambda, nótese también que el atributo de instancia no ha sido declarado como constante, pero es efectivamente constante ya que no se modifica en alguna otra parte del

código:

```
class SessionManager {
 long before = ...;

void expire(File  root) {
   root.listFiles(File p > checkExpiry(p.lastModified(),this.before));
   }

boolean checkExpiry(long time,  long expiry) { ... }
   }
```

Métodos de Referencia
Cuando la expresión lambda se compone de una sola sentencia e invoca algún método existente por medio de su nombre, existe la posibilidad de escribirla usando métodos de referencia, con lo cual se logra un código más compacto y fácil de leer. Existen tres (3) tipos de métodos de referencia y uno adicional (1) para constructores:

Métodos Estáticos
Cuando el método invocado es estático, la forma de escribir la expresión lambda usando métodos de referencia es la siguiente: **NombreClase::métodoEstático**, donde **NombreClase** es el nombre de la clase que contiene el método y **métodoEstático** es el nombre del método estático a invocar. En el siguiente ejemplo, definimos una operación de suma por medio del nuevo método estático *+Integer.sum(int,int):int* el cual suma los dos parámetros y retorna su resultado.

Primero veamos cómo se escribiría usando una expresión lambda:

```
BinaryOperator<Integer> sum = (a,b) > Integer.sum(a,b);
```

Y ahora usando métodos de referencia:

```
BinaryOperator<Integer> sum = Integer::sum;
```

Nótese el uso de la interface funcional *java.util.function.BinaryOperator*, la cual define una función que recibe dos parámetros del mismo tipo y retorna un resultado del mismo tipo de sus parámetros:
+apply(T,T):T.

Métodos de instancia de un objeto
Cuando contamos con una referencia a un objeto y deseamos invocar alguno de sus métodos de instancia dentro de la expresión lambda, la forma en la que la escribiríamos usando métodos de referencia es la siguiente: **RefObjeto::métodoInstancia**, donde **RefObjeto** es la referencia al objeto y **métodoInstancia** es el nombre del método a invocar. Por ejemplo, la clase *java.lang.System* tiene una referencia a un objeto de tipo *java.io.PrintStream* denominada *out*, usaremos esa referencia para nuestro siguiente ejemplo.

Primero veamos cómo se escribiría usando una expresión lambda:

```
Consumer<Integer>  print = (a) > System.out.println(a);
```

Y ahora usando métodos de referencia:

```
Consumer<Integer> print = System.out::println;
```

Nótese que la referencia al objeto la tenemos en *System.out* e invocamos su método de instancia *+println(int):void*

Métodos de instancia de algún tipo
Este caso es parecido al anterior, pero se diferencia en que no contamos con una referencia a un objeto, solo conocemos su tipo y podríamos escribir la expresión lambda de la siguiente forma: **Tipo::métodoInstancia**, donde **Tipo** es la clase y **métodoInstancia** es el nombre del método de instancia a invocar. El siguiente ejemplo define un *java.lang.Comparator* que nos permitirá comparar cadenas sin importar si son mayúsculas/minúsculas.

Primero veamos como se escribiría usando una expresión lambda:

```
Comparator<String>  upper = (a, b) > a.compareToIgnoreCase(b);
```

Y ahora usando métodos de referencia:

```
Comparator<String> upper = String::compareToIgnoreCase;
```

Nótese que en este caso no contamos con la referencia a un objeto como tal, pero sabemos que queremos comprar objetos de tipo String y con eso es suficiente para que podamos escribir nuestra expresión lambda usando métodos de instancia de algún tipo.

Constructores
Para el caso de constructores podemos escribir expresiones lambda como métodos de referencia de la siguiente forma: **Clase::new**, donde **Clase** es la clase que deseamos instanciar y **new** es la palabra reservada ya conocida. El uso de este método de referencia para constructores que no tienen parámetros es sencillo, pero cuando los constructores tienen parámetros, debemos cambiar un poco las cosas.

Primero veamos cómo se escribiría usando una expresión lambda y un constructor sin parámetros:

```
Supplier<List> listSupplier = () > new ArrayList(); List  lista = listSupplier.get();
```

Y ahora usando métodos de referencia:

```
Supplier<List> listSupplier = ArrayList::new; List lista = listSupplier.get();
```

Nótese el uso de la interface funcional *java.util.function.Supplier* y la invocación de su método *+get():T* el cual retorna la lista como tal.

Si el constructor recibe parámetros, tenemos que usar una interface funcional que defina un método que reciba los parámetros. En el siguiente ejemplo, vamos a crear nuevamente una lista, pero esta vez queremos que se invoque el constructor que recibe la capacidad inicial de la lista.

Primero veamos cómo se escribiría usando una expresión lambda:

```
Function<Integer,  List> listSupplier = (num) > new  ArrayList(num);
List lista = listSupplier.apply(5);
```

Y ahora usando métodos de referencia:

```
Function<Integer, List> listSupplier = ArrayList::new; List  lista = listSupplier.apply(5);
```

Nótese el uso de la interface funcional *java.util.function.Function* y la invocación de su método *+apply(T):R* el cual recibe el parámetro que luego será pasado al constructor. Si el constructor recibe dos parámetros, se podría usar la intrerface funcional *java.util.function.BiFunction<T,U,R>* la cual define el método *+apply(T,U):R* que recibe dos parametros.

Conclusión
En esta primera parte analizamos lo beneficioso que puede ser para nuestro código el poder pasar comportamiento como valores/parámetros y revisamos los cambios en el lenguaje que permiten el uso de expresiones lambda:

Funciones como entidades de primer nivel: Las funciones ahora tienen un rol protagónico cuando de expresiones lambda se trata.
Métodos por defecto/estáticos en interfaces: Evolución de librerías sin perder compatibilidad gracias a que ahora podemos definir e implementar métodos en las interfaces.
Interfaces funcionales: Concepto clave para poder escribir expresiones lambda. Interfaces con solo un método abstracto.
Inferencia de tipos: Revisamos los pasos que realiza el compilador para inferir los tipos de las expresiones lambda en contextos de asignación y de invocación de métodos (parámetros).
Alcance de las expresiones lambda: Aprendimos el significado de *Efectivamente Constante* y revisamos la diferencia entre clases anónimas y expresiones lambda en cuanto a la palabra reservada *this* se refiere.
Métodos de referencia: En el último tema vimos otra forma de escribir expresiones lambda de una sola sentencia, con lo cual se logra un código más compacto y fácil de leer.
Para la segunda parte revisaremos la *API Stream*, en dónde usaremos muchas expresiones **lambda** y todo lo aquí aprendido! Puedes continuar leyendo la **segunda parte aquí,**

Información Adicional
Los siguientes enlaces ofrecen mayor información respecto a este tema:

Tutorial de expresiones lambda creado por Oracle (en inglés):
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Documentación API Java SE 8
https://docs.oracle.com/javase/8/docs/api/

Libro recomendado: Java 8 in Action (en inglés)
http://www.manning.com/urma/

Alexis Lopez (@aa_lopez) es consultor independiente Java/ADF/BPM. Ha sido profesor universitario de cursos relacionados con Java y conferencista en congresos reconocidos como: Oracle Open World, JavaOne, Campus Party y OTN Tour. Cuenta con un título de ingeniero de sistemas y las siguientes certificaciones: SCJP, OCPJMAD, OCPWCD, especialista de implementación de Oracle ADF y Oracle BPM. Es líder del grupo de usuarios Java de Cali-Colombia (www.clojug.org), miembro del comité de dirección del grupo de usuarios virtual de Java (virtualjug.com) y blogger activo en www.java-n-me.com