1. Características Java Entorno Servidor

# 1.3 Lambdas y Streams (Java Moderno de Java 8 - en 2014)

En primer lugar, no deben confundirse los streams de Java 8 con los I/O streams (flujos de entrada/salida) de Java como FileInputStream.

En general, tienen muy poco que ver unos con otros.

## 1.3.1 I/O Streams o Flujos de Entrada/Salida

La E/S en Java sigue el mismo modelo que en Unix:

* Abrir, usar, cerrar flujo.
* Flujos estándar: System.in, System.out y System.err

|  |
| --- |
| **package** com.jmmartin.instituto;  **import** java.io.\*;  **class** **LecturaDeLinea** {   **public** **static** **void** **main**(String args[]) **throws** IOException {  **int** c;  **int** contador = 0; *// se lee hasta encontrar el fin de línea*  **while** ((c = System.in.read()) != '\n') {  contador++;  System.out.print((**char**) c);  }  System.out.println(); *// Se escribe el fin de línea*  System.err.println("Contados " + contador + " bytes en total.");  } } |

Dos tipos de clases de E/S:

* Readers y Writers para texto.
  + Basados en el tipo char. Heredan de las clases abstractas: java.io.Reader y java.io.Writer.
* Streams (InputStream y OutputStream) para datos binarios.
  + Basados en el tipo byte. Heredan de las clases abstractas: java.io.InputStream y java.io.OutputStream.

|  |
| --- |
| ¿Cuándo usar el flujo de caracteres sobre el flujo de bytes?  En Java, los caracteres se almacenan mediante Unicode.  El flujo de caracteres es útil cuando queremos procesar archivos de texto. Estos archivos de texto se podrán procesar carácter por carácter.  El tamaño de los caracteres suele ser de 16 bits.  Mientras que los flujos de bytes procesan los datos byte a byte (8 bits) |

Regla de IO Streams:

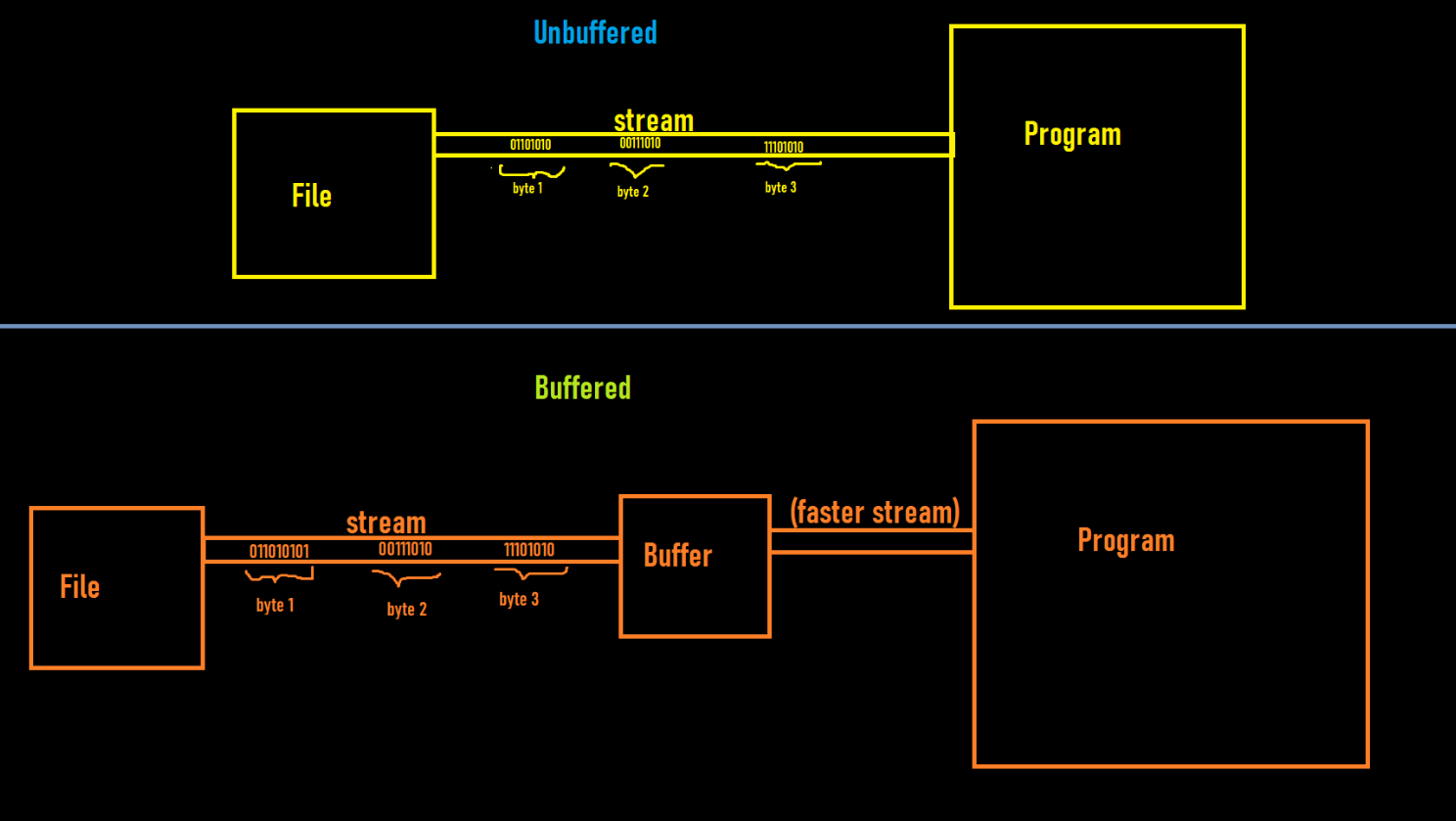
* Flujos de caracteres normalmente terminan con: Reader/Writer
* Flujos de bytes terminan con: InputStream/OutputStream.

Por lo general, se usarán lectores/escritores o flujos de E/S almacenados en búfer (Buffered) para mayor eficiencia:

* BufferedReader/BufferedWriter (para flujo de caracteres)
* BufferedInputStream/BufferedOutputStream (para flujo de bytes).

En el caso de flujos sin búfer cada petición de lectura o escritura se maneja directamente por el SO subyacente (API nativa). Esto puede hacer que un programa sea mucho menos eficiente, ya que cada solicitud de este tipo a menudo desencadena el acceso al disco, la actividad de la red o alguna otra operación que es relativamente costosa.

Para reducir este tipo de sobrecarga, Java implementa flujos de E/S almacenados en búfer. Los flujos de entrada almacenados en búfer leen datos de un área de memoria conocida como búfer; la API de entrada nativa se llama solo cuando el búfer está vacío. De manera similar, los flujos de salida almacenados en búfer escriben datos en un búfer; la API de salida nativa sólo se llama cuando el búfer está lleno.



Siempre se recomienda cerrar la transmisión del flujo si ya no está en uso. Esto asegura que las transmisiones no se verán afectadas si ocurre algún error.

La E/S suele ser propensa a errores

* Implica interacción con el entorno exterior
* Excepción IOException

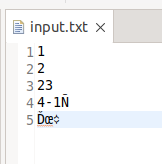
|  |
| --- |
| **package** com.jmmartin.instituto;  **import** java.io.BufferedInputStream; **import** java.io.FileInputStream;  **class** **Main** {  **public** **static** **void** **main**(String[] args) {  **try** {   *// Crea un FileInputStream*   FileInputStream file = **new** FileInputStream("/home/user/eclipse-workspace/Java Avanzado/src/com/jmmartin/instituto/input.txt");     *// Wrapeas (envuelves) el FileInputStream en un BufferedInputStream*  BufferedInputStream input = **new** BufferedInputStream(file);   *// Lees el primer byte*  **int** i = input .read();   **while** (i != -1) {  System.out.print((**char**) i);   *// Lee el siguiente byte*  i = input.read();  }  input.close();  } **catch** (Exception e) {  e.getStackTrace();  }  } } |

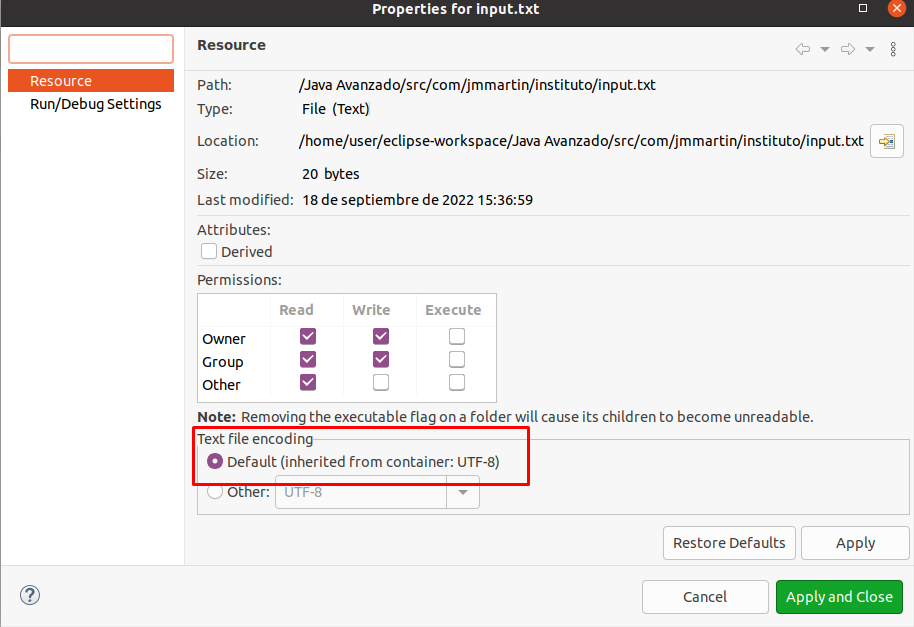
Pregunta: ¿Piensas que es correcto utilizar un InputStream para leer un fichero de texto?

Repuesta:

InputStream está diseñado para leer datos binarios, no caracteres, y en los datos binarios (esencialmente) siempre se lee un byte a la vez. Si se desea leer texto, debes utilizar un Reader (preferiblemente especificando explícitamente la codificación que se utilizará) para convertir los datos binarios en texto. Internamente, llamará a read() una o más veces para construir correctamente un carácter a partir de la secuencia de bytes basada en la codificación.

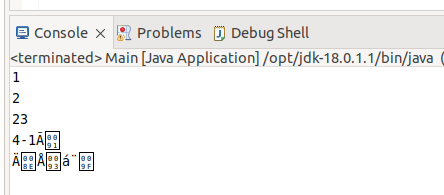
Supongamos que input.txt es como sigue, con codificación por defecto del entorno de utf-8







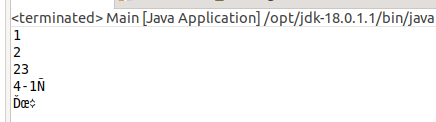
Si utilizo InputStream para leer el fichero y represento por salida de consola estándar tendremos:



En cambio si reescribo el programa como sigue:

|  |
| --- |
| **public** **static** **void** **main**(String[] args) {  **try** {   *// Crea un FileReader,*   FileReader file = **new** FileReader("/home/user/eclipse-workspace/Java Avanzado/src/com/jmmartin/instituto/input.txt");    *// Wrapeas (envuelves) el FileReader en un BufferedReader*  BufferedReader input = **new** BufferedReader(file);   *// Lees el primer carácter*   **int** i = input .read();   **while** (i != -1) {  System.out.print((**char**) i);   *// Lee el siguiente carácter*  i = input.read();  }  input.close();  }  **catch** (Exception e) {  e.getStackTrace();  }  } |
|  |

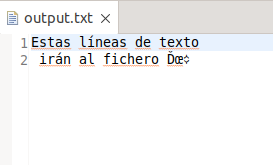
Ahora la salida será correcta. Ten en cuenta que FileReader utilizará el encoding por defecto del entorno en este caso utf-8, detectando los caracteres de 2 trasladándose a la salida int i.



Veamos ahora un ejemplo de escritura de flujo de salida a fichero:

|  |
| --- |
| **public** **static** **void** **main**(String[] args) {   String data = "Estas líneas de texto\n irán al fichero Ďœ᨟";   **try** {  *// Crea el FileOutputStream*  FileOutputStream file = **new** FileOutputStream("/home/user/eclipse-workspace/Java Avanzado/src/com/jmmartin/instituto/output.txt");   *// Se envuelve (wrapea) en un stream con bufer. Tamaño búfer de 8192 bytes, por defecto.*  BufferedOutputStream output = **new** BufferedOutputStream(file);   **byte**[] array = data.getBytes();   *// Se escriben los datos al flujo de salida.*  output.write(array);  *// Nunca olvidar que se debe cerrar el flujo.*  output.close();  }   **catch** (Exception e) {  e.getStackTrace();  }  } |

Obteniéndose la siguiente salida:



Vemos que en este caso no ha habido problema de codificación dado que hemos utilizado la función String.getBytes() que es segura. No obstante, al tratar de fichero de texto lo recomendable es emplear un Writer.

|  |
| --- |
| **public** **static** **void** **main**(String[] args) {   String data = "Estas líneas de texto\n irán al fichero Ďœ᨟\nĎœ᨟";   **try** {  *// Crea el FileWriter*  FileWriter file = **new** FileWriter("/home/user/eclipse-workspace/Java Avanzado/src/com/jmmartin/instituto/output.txt");   *// Se envuelve (wrap) en un stream con bufer, con tamaño 8196 bytes por defecto.*  BufferedWriter output = **new** BufferedWriter(file);   **byte**[] array = data.getBytes();   *// Se escribe los datos al flujo de salida.*  output.write(data);  *// Nunca olvidar que se debe cerrar el flujo.*  output.close();  }   **catch** (Exception e) {  e.getStackTrace();  }  } |

## 1.3.2 Interface Stream<T> de Java 8

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

Los flujos (streams) de Java 8 son una actualización de la API de Java que permiten manipular colecciones de datos de una forma declarativa:

* se expresa una consulta en lugar de codificar una implementación, ad hoc.

Se puede pensar en ellos como iteradores sofisticados sobre una colección de datos. Además, los flujos se pueden procesar en paralelo de forma transparente.

Veamos cómo cambia el código para devolver el nombre de platos bajos en calorías de una colección, ordenados por cantidad de calorías de la versión Java 7 a Java 8.

En Java 7 había que hacer:

|  |
| --- |
| List<Dish> lowCaloricDishes = **new** ArrayList<>();    **for** (Dish dish : menu) {  **if** (dish.getCalories() < 400) {  lowCaloricDishes.add(dish);  }  }  Collections.sort(lowCaloricDishes, **new** Comparator<Dish>() {  **public** **int** **compare**(Dish dish1, Dish dish2) {  **return** Integer.compare(dish1.getCalories(), dish2.getCalories());  }  });  List<String> lowCaloricDishesName = **new** ArrayList<>();  **for** (Dish dish : lowCaloricDishes) {  lowCaloricDishesName.add(dish.getName());  } |

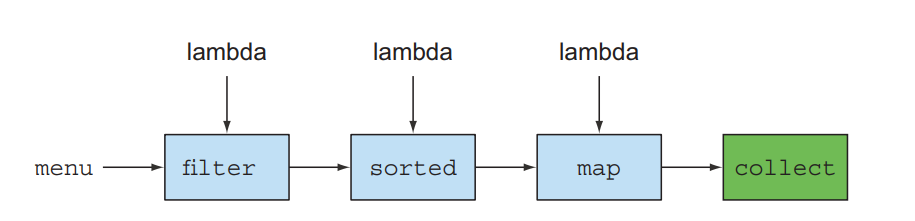
En Java 8 todo este código se simplifica a lo siguiente:

|  |
| --- |
| List<String> lowCaloricDishesName = menu  .stream()  .filter(d -> d.getCalories() < 400)  .sorted(comparing(Dish::getCalories))  .map(Dish::getName) .collect(toList()); |

Incluso se puede explotar el paralelismo del sistema mediante:

|  |
| --- |
| List<String> lowCaloricDishesName = menu  .parallelStream()  .filter(d -> d.getCalories() < 400)  .sorted(comparing(Dish::getCalories))  .map(Dish::getName)  .collect(toList()); |

En la siguiente figura vemos la cadena o pipeline de métodos aplicados al stream menu:

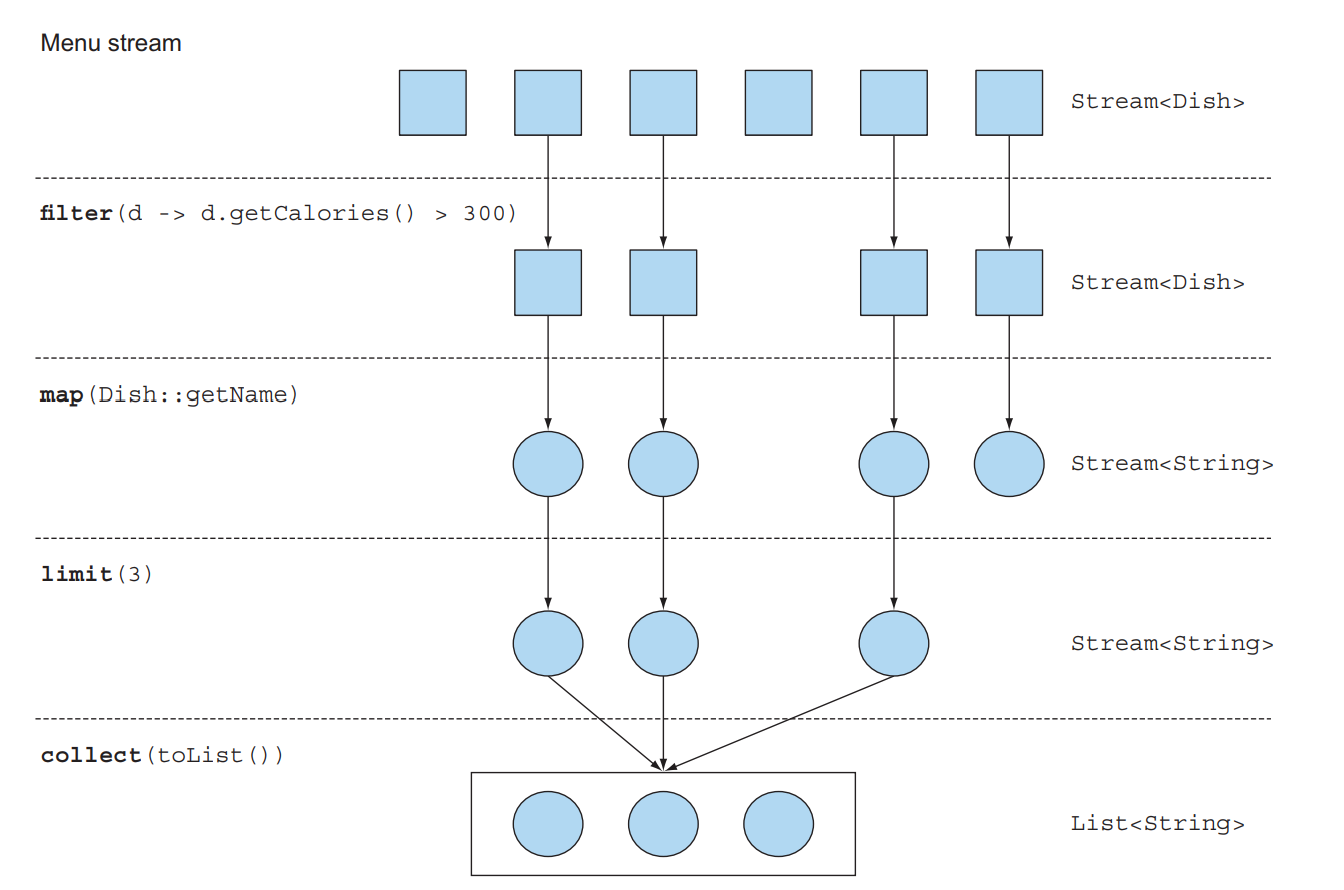


Donde lambda es una función o expresión funcional anónima.

En el siguiente ejemplo se quieren obtener una lista con 3 platos de más de 300 calorías:

|  |
| --- |
| List<String> threeHighCaloricDishNames = menu  .stream()   .filter(dish -> dish.getCalories() > 300)  .map(Dish::getName)   .limit(3)   .collect(toList()); |
|  |
|  |

Visualmente tendremos un flujo de la siguiente manera:

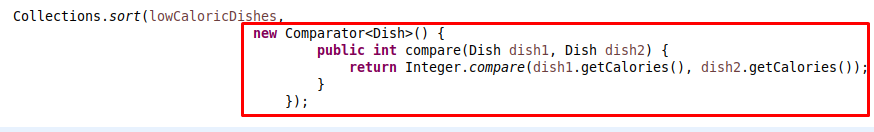


## 1.3.3 Lambdas



* Una expresión lambda puede entenderse como una especie de función anónima:
  + no tiene nombre,
  + pero tiene:
    - una lista de parámetros,
    - un cuerpo,
    - un tipo de devolución
    - y también posiblemente una lista de excepciones que se pueden lanzar.

Enlazando con el ejemplo de Comparator:



Nombre referenciador al objeto de la clase

|  |
| --- |
| *// Sin lambdas*  Comparator<Dish> cmpClaseAnonimaInterna = **new** Comparator<Dish>() {  **public** **int** **compare**(Dish dish1, Dish dish2) {  **return** Integer.compare(dish1.getCalories(), dish2.getCalories());  }  };   Collections.sort(lowCaloricDishes, cmpClaseAnonimaInterna); |

O sin la referencia a la clase anónima explícitamente:

Lista objeto con metodos

|  |
| --- |
| Collections.sort(lowCaloricDishes, **new** Comparator<Dish>() {  **public** **int** **compare**(Dish dish1, Dish dish2) {  **return** Integer.compare(dish1.getCalories(), dish2.getCalories());  }  }); |

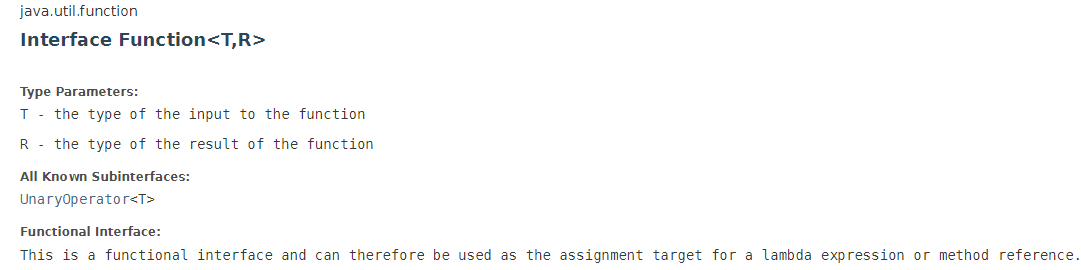
* Las expresiones lambda le permiten pasar el código de forma concisa:

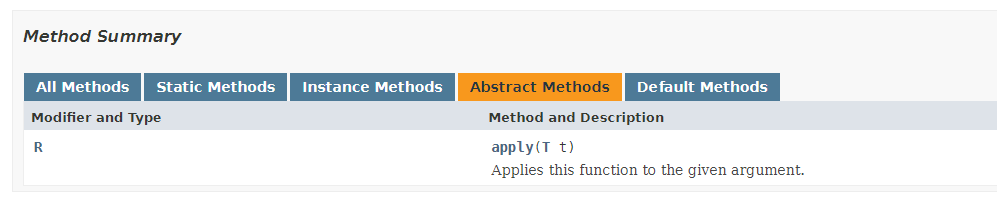
|  |
| --- |
| *// Con lambdas*  Comparator<Dish> cmpLambda = (dish1, dish2) -> Integer.compare(dish1.getCalories(), dish2.getCalories());   Collections.sort(lowCaloricDishes, cmpLambda); |

O sin la referencia a la función explícitamente:

|  |
| --- |
| Collections.sort(lowCaloricDishes, (dish1, dish2) -> Integer.compare(dish1.getCalories(), dish2.getCalories())); |

* Una interfaz funcional es una interfaz que declara exactamente un método abstracto. Por ejemplo, la interfaz *Function<T,R>* sólo declara un método abstracto: *R apply(T t).*





Excepción 1 solo método abstracto:

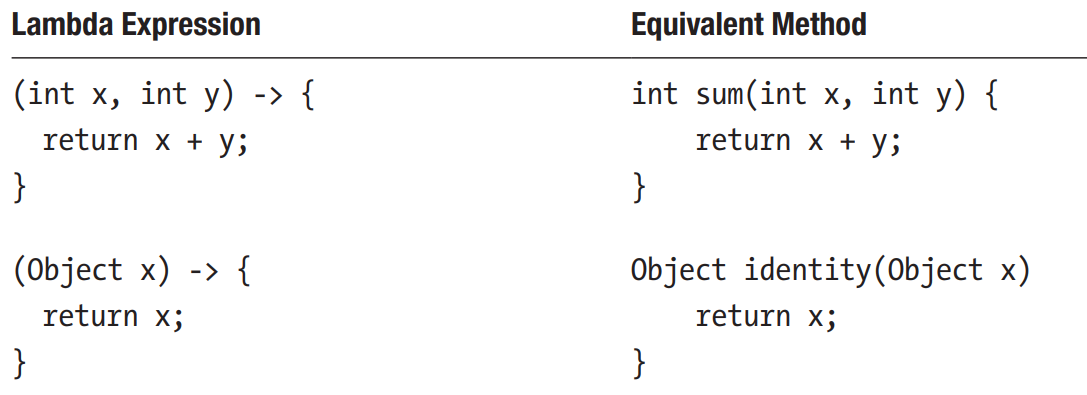
* Existe la excepción de que la interfaz funcional puede *redeclarar* métodos abstractos de la clase Object con fines de ampliar la documentación.

Regla:

|  |
| --- |
| Las expresiones lambda sólo se pueden usar cuando se espera una interfaz funcional. |

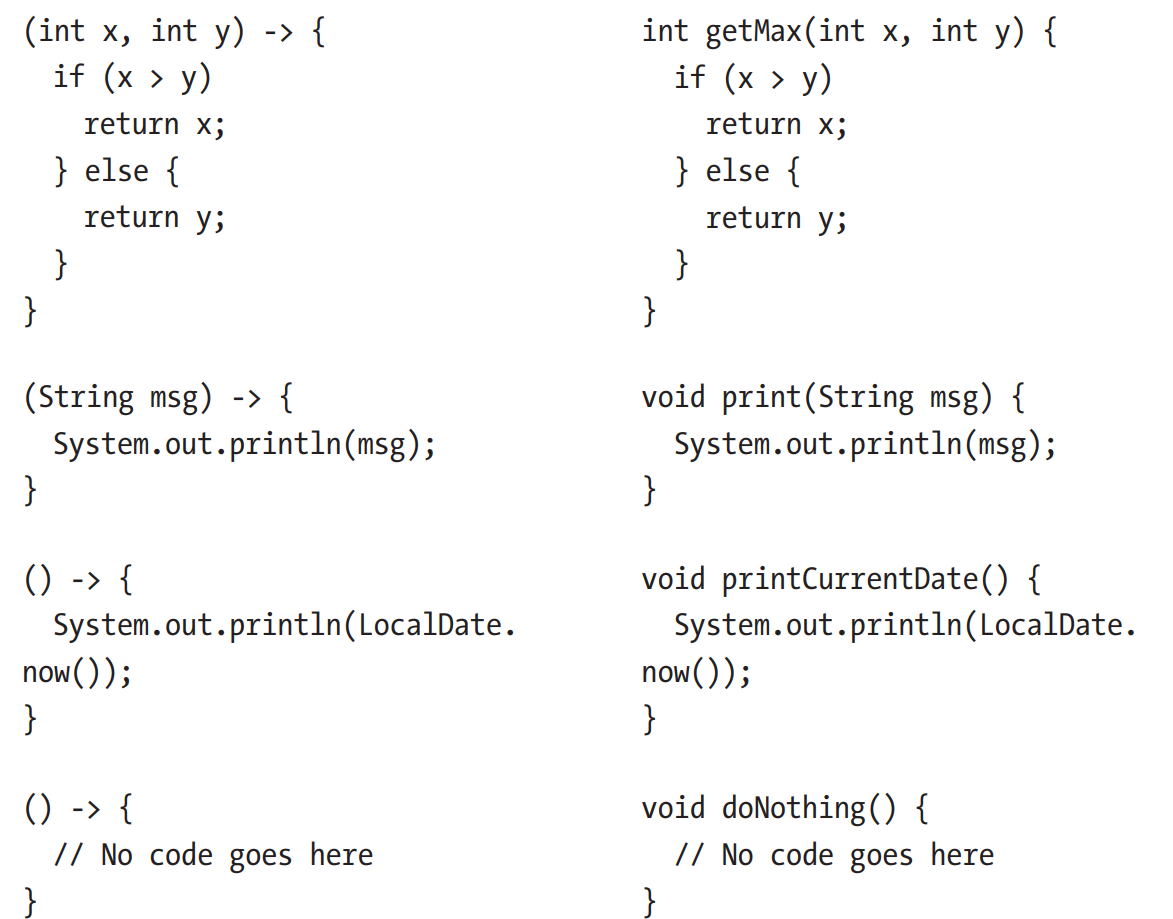
* Las expresiones lambda permiten:
  + la implementación del método abstracto de una interfaz funcional directamente en línea
  + y tratar la expresión completa como una instancia de una interfaz funcional.

Algunos ejemplos de lambdas y métodos equivalentes:



1.3.3.1 Clases anónimas internas JAVA 5 vs Lambdas JAVA 8

<https://github.com/toteabe/Functional_Java/blob/master/src/test/java/JavaFunctionalTest.java>

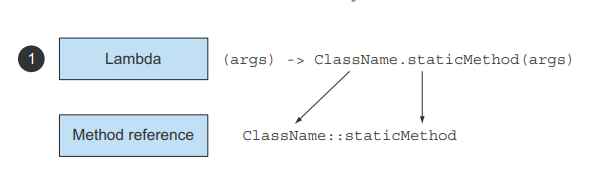


|  |
| --- |
| **package** org.iesvdm;  **public** **interface** **IFuncionOperacion** {  **public** **double** **operacion**(**double** operando1, **double** operando2);  } |

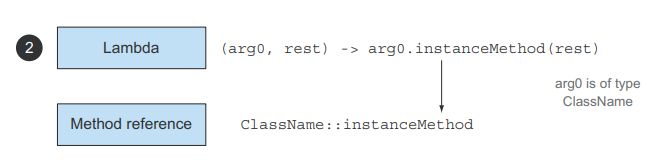
|  |
| --- |
| **import** org.iesvdm.IFuncionOperacion; **import** org.junit.jupiter.api.Assertions; **import** org.junit.jupiter.api.Test;  **public** **class** **JavaFunctionalTest** {    **@Test**  **public** **void** **claseAnonimaFuncionesAntesDeJAVA8**() {   *// Una interfaz Java a priori no es instanciable -> MEDIA VERDAD S*  *// SÍ se puede instancia mediante el mecanismo de clase anónima interna (Anonymous Inner Class)*  *// Una especie implementación on-fly (in-line) de una clase que cumpla con la interfaz.*   *// Interfaz Clase Anónima Interna (se implementa una interfaz on-fly en el código)*  *// | |*  *// --------- --*  *// V V*  IFuncionOperacion iFuncionOperacionSuma = **new** IFuncionOperacion() {   *// Se realiza un @Override implementación del único método de la interfaz (interfaz funcional, por tanto)*  *// |*  *// -------------*  *// V |*  **@Override** *// V*  **public** **double** **operacion**(**double** operando1, **double** operando2) {  *//En esta clase anónima implementas la suma*  **return** operando1 + operando2;  }  };   *// Interfaz Clase Anónima Interna (se implementa una interfaz on-fly)*  *// | |*  *// --------- --*  *// V V*  IFuncionOperacion iFuncionOperacionProducto = **new** IFuncionOperacion() {  **@Override**  **public** **double** **operacion**(**double** operando1, **double** operando2) {  *//En esta clase anónima implementas el producto*  **return** operando1 \* operando2;  }  };   Assertions.assertEquals(4, iFuncionOperacionSuma.operacion(1, 3));  Assertions.assertEquals(3, iFuncionOperacionProducto.operacion(1, 3));   }   **@Test**  **public** **void** **closureEnClaseAnonima**() {   *// Dentro de una clase anónima se puede atrapar una variable en el ámbito externo*  *// Se necesita que esa variable sea declarada final (o virtualmente final)*  *// |*  *// -------------------------------------------*  *// |*  *// V*  **final** **double** constante = 1.234;   IFuncionOperacion iFuncOperProdYConstante = **new** IFuncionOperacion() {  **@Override**  **public** **double** **operacion**(**double** operando1, **double** operando2) {  *// Variable que se accede por closure desde el ámbito externo*  *// |*  *// V*  **return** operando1 \* operando2 \* constante;  }  };   Assertions.assertEquals(1234, iFuncOperProdYConstante.operacion(10.0, 100.0));   }   **@Test**  **public** **void** **closureEnClaseAnonimaConAtributo**() {   **final** **double** constante = 1.234;   IFuncionOperacion iFuncionOperacion = **new** IFuncionOperacion() {   *//Atributo inicializado a constante final de la closure (ámbito externo del cual se atrapa la variable)*  *// |*  *// V*  **private** **double** constanteInner = constante;  **@Override**  **public** **double** **operacion**(**double** operando1, **double** operando2) {  **return** operando1 \* operando2 \* constante;  }  };  Assertions.assertEquals(1234.0, iFuncionOperacion.operacion(10.0, 100.0));    }   **@Test**  **public** **void** **pasoDeArgumentosAClaseAnonimaSinClosure**() {   IFuncionOperacion iFuncionOperacion = **new** IFuncionOperacion() {   *// El atributo interno no se enlaza con variable externa mediante closure, sino que establece mediante cadena de método en setter*  *// |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*  *// V |*  **private** **double** constante; *// |*  *// |*  *// |*  *// ----------------------------------------|*  *// V V*  **public** IFuncionOperacion **setConstante**(**double** constante) {  **this**.constante = constante;  **return** **this**;  }   **@Override**  **public** **double** **operacion**(**double** operando1, **double** operando2) {  **return** operando1 \* operando2 \* **this**.constante;  }  }.setConstante(1.234);   Assertions.assertEquals(1234, iFuncionOperacion.operacion(10.0, 100.0));   }   **@Test**  **public** **void** **REVOLUCION\_JAVA\_8\_LAMBDAS**() {   **final** **double** constante = 1.234;   *//SE REEMPLAZA TODO EL CÓDIGO DE CLASE ANÓNIMA INTERNA POR UN LAMBDA (O DECLARACIÓN DE FUNCIÓN ANÓNIMA)*   *//LAMBDA -------------------------*  *// | |*  *// SE BASA EN UNA INTERFAZ |*  *// FUNCNIONAL |*  *// | |*  *// INTERFAZ CON UN |*  *// SOLO MÉTODO |*  *// ABSTRACTO |*  *// | |*  *// V V*  IFuncionOperacion ifo = (o1, o2) -> o1 \* o2 \* constante;   Assertions.assertEquals(1234, ifo.operacion(10.0, 100.0));   }  } |

## 1.3.4 Reemplazo de lambda por referencia a método

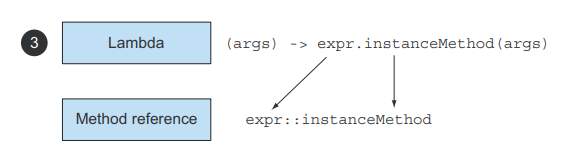
### 1.3.4.1 Por método estático



1.3.4.2 Por método de instancia de un objeto arbitrario con paso de argumentos



### 1.3.4.3 Por método de instancia de un objeto particular (o expresión que devuelve objeto particular)



### 1.3.4.4 Resumen transformación lambda a referencia a método

Observa que método estático e instancia de objeto particular tienen el mismo tipo de declaración: Clase::método.

* La diferencia estriba en que en estático el método es estático y todos los argumentos/parámetros (args) de la lambda se trasladan como argumentos/parámetros del método estático.
* Mientras que en objeto arbitrario el método no es estático, sino que necesita de una instancia de objeto que se toma del primer argumento/parámetro (arg0) de la lambda, pasando los restantes parámetros (rest) como parámetros del método.

Por último, en objeto particular se tiene el método aplicado a un objeto en el cuerpo del lambda.

### 1.3.4.5 Tabla de ejemplos de transformaciones a referencias métodos

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo de Referencia a Método** | **Referencia a Método** | **Expresión Lambda** | **Interfaz Funcional Java 8** |
| **Método estático** | String::valueOf | (int num) → String.valueOf(num) | Function<T, R>  (T) → R  *Function<Integer, String>* |
| **Método de instancia de objeto arbitrario o expresión** | String::equals | (String a. String b) → a.equals(b) | BiFunction<T, U, Boolean>  (T, U) → Boolean  BiPredicate<T, U>  (T, U) → Boolean  *BiPredicate<String, String>* |
| JLabel::getIcon | (JLabel jl) → jl.getIcon() | Function<T, R>  (T) → R  Function<JLabel, Icon> |
| **Método de instancia de objeto particular** | s::substring | (int ini, int fin) → s.substring(ini, fin) | BiFunction<T, U, R>  (T, U) → R  *BiFunction<Integer, Integer, String>* |
| **Constructor** | String::new | () → new String() | Supplier<T>  () → T  Supplier<String> |

Ejemplo:

|  |
| --- |
| **interface** **F0** {  **void** **f0**(); } **interface** **F1** {  **void** **f1**(MetRef i1); } **interface** **F2** {  **void** **f2**(MetRef i1, MetRef i2); }  **public** **class** **MetRef** {  **public** **static** **void** **stat0**() {;}  **public** **static** **void** **stat1**(MetRef a) {;}  **public** **static** **void** **stat2**(MetRef a, MetRef b) {;}   **public** **void** **inst0**() {;}  **public** **void** **inst1**(MetRef a) {;}  **public** **void** **inst2**(MetRef a, MetRef b) {;} } |

|  |
| --- |
| **public** **class** **Test** {  **public** **static** **void** **main**(String[] args) {  **final** MetRef mr = **new** MetRef();   **final** F0 mr01 = MetRef::stat0; *// 1: f0() ~ MetRef.stat0()*   **final** F0 mr02 = mr::inst0; *// 2: f0() ~ mr.inst0()*  **final** F0 mr04 = MetRef::**new**; *// 4: f0() ~ new MetRef()*   **final** F1 mr11 = MetRef::stat1; *// 1: f1(i1) ~ MetRef.stat1(i1)*  **final** F1 mr12 = mr::inst1; *// 2: f1(i1) ~ mr.inst1(i1)*  **final** F1 mr13 = MetRef::inst0; *// 3: f1(i1) ~ i1.inst0() <==!*   **final** F2 mr21 = MetRef::stat2; *// 1: f2(i1, i2) ~ MetRef.stat2(i1, i2)*  **final** F2 mr22 = mr::inst2; *// 2: f2(i1, i2) ~ mr.inst2(i1, i2)*  **final** F2 mr23 = MetRef::inst1; *// 3: f2(i1, i2) ~ i1.inst1(i2) <==!*  } } |

## 1.3.5 Closure: uso de variables locales del ámbito de definición de la lambda de Java

Un Closure es una instancia de una función (lambda) que puede hacer referencia a variables no locales de esa función. (es decir que no estan dentro de la lambda)

por ejemplo, El cierre podría pasarse como argumento a otra función. (una variable que se pasa como argumento en un lambda y la variable queda atrapada como final)

También podría acceder y modificar variables definidas fuera de su alcance. Ahora, las lambdas de Java 8 y las clases anónima hacen algo similar a los Closures: se pueden pasar como argumento a métodos y pueden acceder a variables fuera de su alcance.

Pero tienen una restricción: no pueden modificar el contenido de las variables locales de un método en el que la lambda es definido. (queda final virtualmente si no se declara final)

Esas variables tienen que ser implícitamente finales. Puedes pensar que las lambdas cierran los valores de las variables.

Este código sería correcto puesto que implícitamente en el ámbito local no se modifica la variable sobre la que se cierra la lambda *portNumber*

|  |
| --- |
| **int** portNumber = 1337; Runnable r = () -> System.out.println(portNumber); |

En cambio, en el siguiente bloque de código se rompe la naturaleza final de portNumber al haber una segunda asignación:

|  |
| --- |
| **int** portNumber = 1337; Runnable r = () -> System.out.println(portNumber);  portNumber = 31337; |

Esta restricción existe porque las variables locales se van a encontrar en la pila (STACK) y están implícitamente confinadas al hilo (thread) en el que están, que desaparecerán cuando la ejecución abandone el ámbito de definición de estas variables locales.

Permitir la captura de variables locales mutables abre posibilidades de inseguridad en un entorno multi-hilo (multi-thread) para el cual está pensado Java, que no son deseables.

En cambio, las variables de instancia no presentan este problema porque se encuentran en el heap, que se comparte entre hilos (threads).

### 1.3.5.1 Repaso de la memoria Java: Stack vs Heap

HEAP: Espacio de almacenamiento dinámico de Java

En tiempo de ejecución de Java se utiliza el espacio de la pila de memoria Java para asignar memoria a objetos y clases de JRE.

Cada vez que creamos un objeto, siempre se crea en el espacio Heap. Existe un recolector de basura (GC) que se ejecuta en la memoria del HEAP para liberar la memoria utilizada por los objetos que no tienen ninguna referencia.

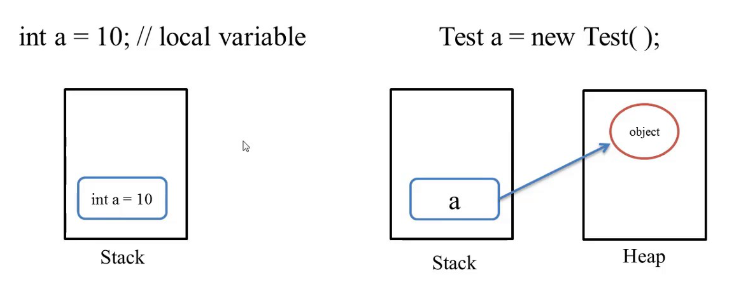
Cualquier objeto creado en el espacio de almacenamiento dinámico tiene acceso global y se puede hacer referencia a él desde cualquier parte de la aplicación.

STACK: Memoria de pila de Java (Pila)

La memoria de Java Stack se utiliza para la ejecución de un hilo. Contiene valores específicos del método que son de corta duración y referencias a otros objetos en el HEAP a los que se hace referencia desde el método.

Se trata de una memoria tipo LIFO (Último en entrar, primero en salir). Cada vez que se invoca un método, se crea un nuevo bloque en la memoria de la pila para que el método contenga valores primitivos locales y haga referencia a otros objetos en el método. Tan pronto como finaliza el método, el bloque deja de utilizarse y queda disponible para el siguiente método.

El tamaño de la memoria de pila es mucho menor en comparación con la memoria Heap.



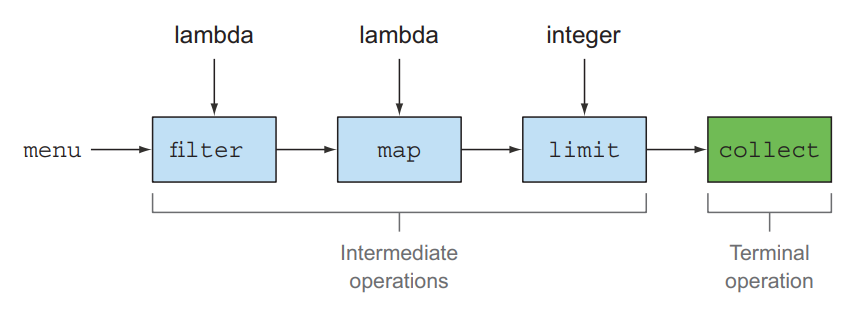
## 1.3.6 Operaciones con Streams

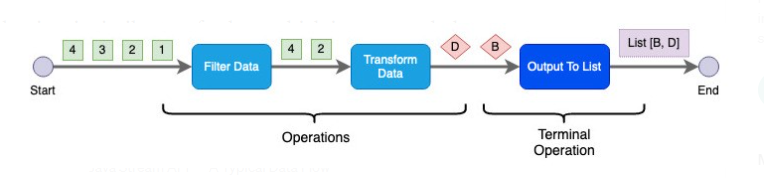
La interfaz de *streams* en java.util.stream.Stream define muchas operaciones.

Éstas se dividen en dos categorías:

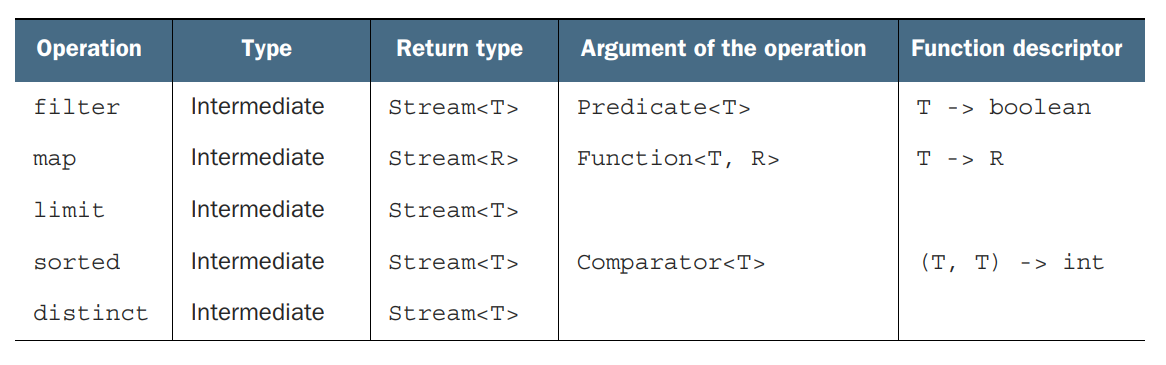
* Operaciones intermedias.
* Operaciones terminales.

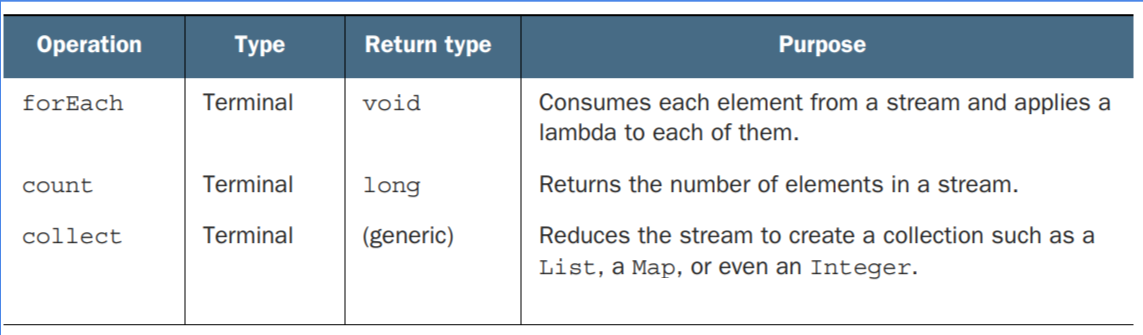
|  |
| --- |
| List<String> names = menu.stream()  .filter(dish -> dish.getCalories() > 300)   .map(Dish::getName)  .limit(3)  .collect(toList()); |





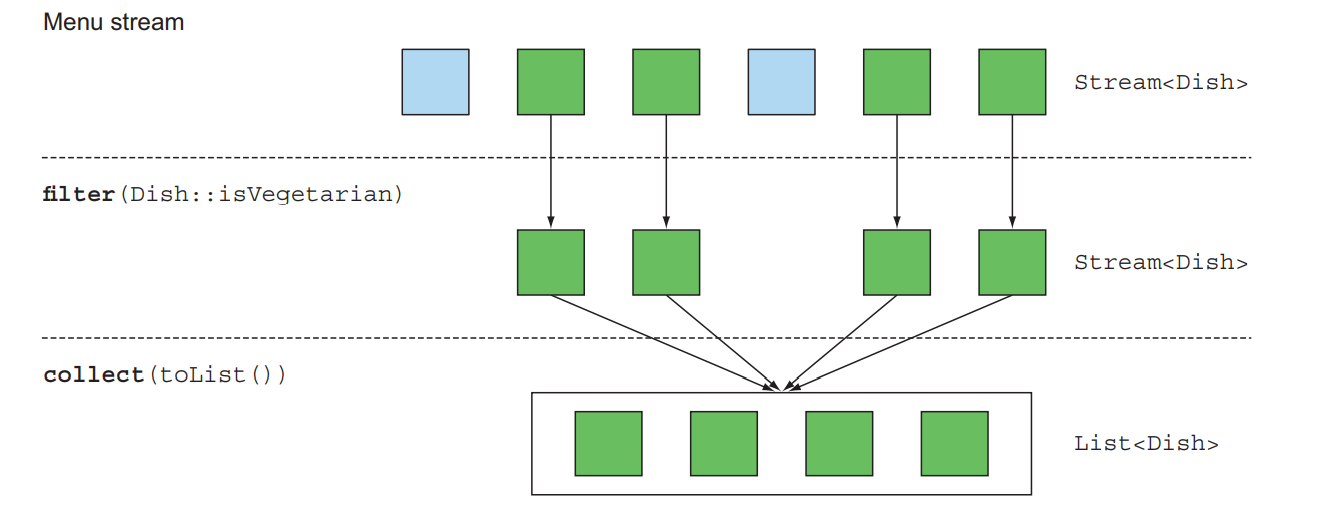
\_\_\_ Operaciones intermedias *filter, map, limit*,... devuelven siempre un stream y se pueden encadenar como *pipeline*



\_\_\_ Operación terminal *collect* produce un resultado final de salida del stream

## 1.3.7 filter

|  |  |
| --- | --- |
| **Recoger Platos Vegetarianos del Menú** | |
| **Mediante Bucle** | **Mediante Stream** |
| List<Dish> vegetarianDishes = **new** ArrayList<>();  **for**(Dish d: menu) {  **if**(d.isVegetarian()) {  vegetarianDishes.add(d);  }  } | **import** **static** java.util.stream.Collectors.toList;  List<Dish> vegetarianDishes = menu.stream()  .filter(Dish::isVegetarian) // <-> .filter( dish -> dish.isVegetarian() )  // en filter los elementos del stream que no cumplen el predicado se eliminan  .collect(toList()); |

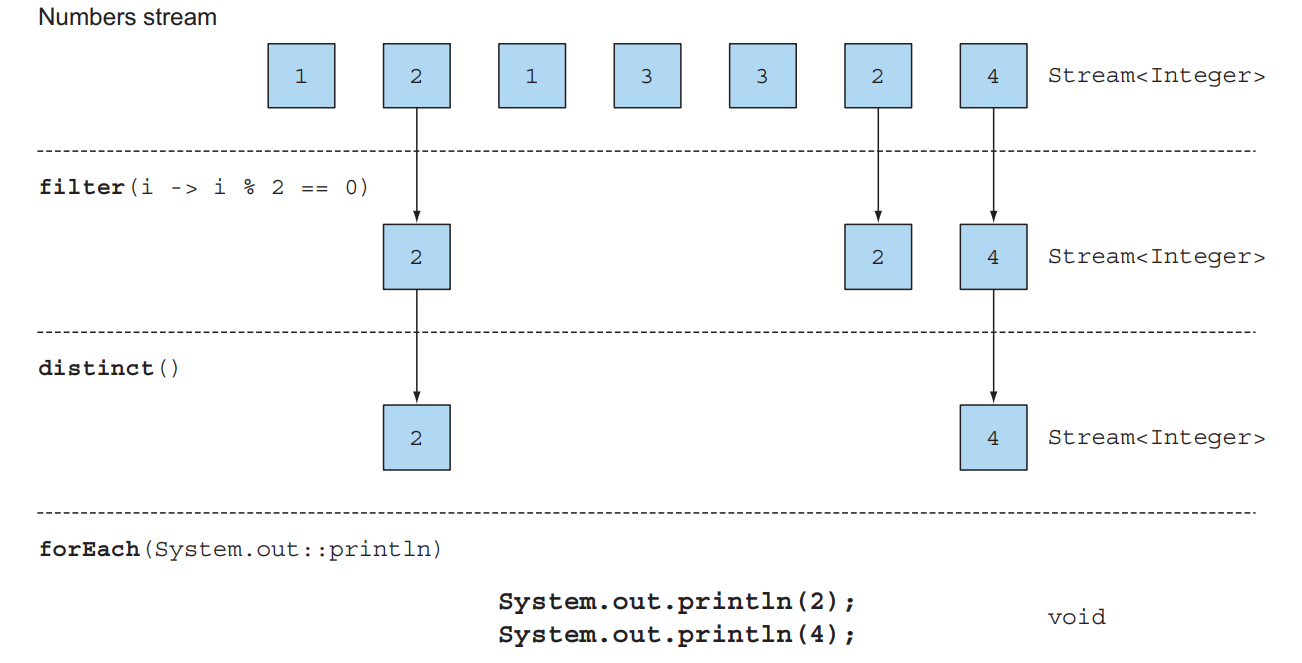


1.3.8 sorted

|  |
| --- |
| List<Dish> lowCaloricDishesName = menu  .parallelStream()  .filter(d -> d.getCalories() < 400)  .sorted(comparator.comparing(Dish::getCalories))  // comparing genera una lambda de tipo Comparator<T> sobre el método de getCalories de Dish  //.sort(Comparator.comparing((Dish d) -> d.getCalories));  //.sort(  .collect(toList()); |

## 1.3.9 distinct

|  |
| --- |
| List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);   numbers.stream()  .filter(i -> i % 2 == 0)   .distinct()   .forEach(System.out::println); |



## 1.3.10 takeWhile (Java 9)

|  |
| --- |
| List<Dish> specialMenu = Arrays.asList( **new** Dish("seasonal fruit", **true**, 120, Dish.Type.OTHER)  , **new** Dish("prawns", **false**, 300, Dish.Type.FISH)  , **new** Dish("rice", **true**, 350, Dish.Type.OTHER)  ,**new** Dish("chicken", **false**, 400, Dish.Type.MEAT)  , **new** Dish("french fries", **true**, 530, Dish.Type.OTHER));  //Fíjate que specialMenu está ordenado de menor a mayor calorías..  List<Dish> filteredMenu = specialMenu.stream()  .takeWhile(dish -> dish.getCalories() < 320) //Selecciona hasta que deja de cumplirse por 1a vez el predicado.  //tomaMientras (secuencialmente) -sólo con sentido en colecciones ordenadas.  .collect(toList());  // En filteredMenu tendremos solo: seasonal fruit, prawns |

## 1.3.11 dropWhile (Java 9)

|  |
| --- |
| List specialMenu = Arrays.asList( **new** Dish("seasonal fruit", **true**, 120, Dish.Type.OTHER)  , **new** Dish("prawns", **false**, 300, Dish.Type.FISH)  , **new** Dish("rice", **true**, 350, Dish.Type.OTHER)  ,**new** Dish("chicken", **false**, 400, Dish.Type.MEAT)  , **new** Dish("french fries", **true**, 530, Dish.Type.OTHER));  //Fíjate que specialMenu está ordenado de menor a mayor calorías  List filteredMenu = specialMenu.stream()  .dropWhile(dish -> dish.getCalories() < 320) //Descarta hasta que deja de cumplirse por 1a vez el predicado, a partir de ahí devuelve todo.  //descartaMientras (secuencialmente) -sólo con sentido en colecciones ordenadas  .collect(toList());  // En filteredMenu tendremos solo: seasonal fruit, prawns |

## 1.3.12 limit

|  |
| --- |
| List dishes = specialMenu .stream()   .filter(dish -> dish.getCalories() > 300)   .limit(3) //Se queda con los tres primeros del flujo, en este caso, que hayan pasado por el predicado de filter  .collect(toList()); |

## 1.3.13 skip

|  |
| --- |
| List dishes = menu.stream()  .filter(d -> d.getCalories() > 300)  .skip(2) //Descarta los 2 primeros del flujo, en este caso, que hayan pasado por el predicado de filter  .collect(toList()); |

## 1.3.14 map

|  |
| --- |
| List<String> dishNames = menu.stream()  .map(Dish::getName) //Aplica a cada elemento del flujo una función, en este caso, Dish::getName  // Mapear se puede interpretar por transformar, el elemento se mapea con el resultado de la función (se transforma)  .collect(toList()); |

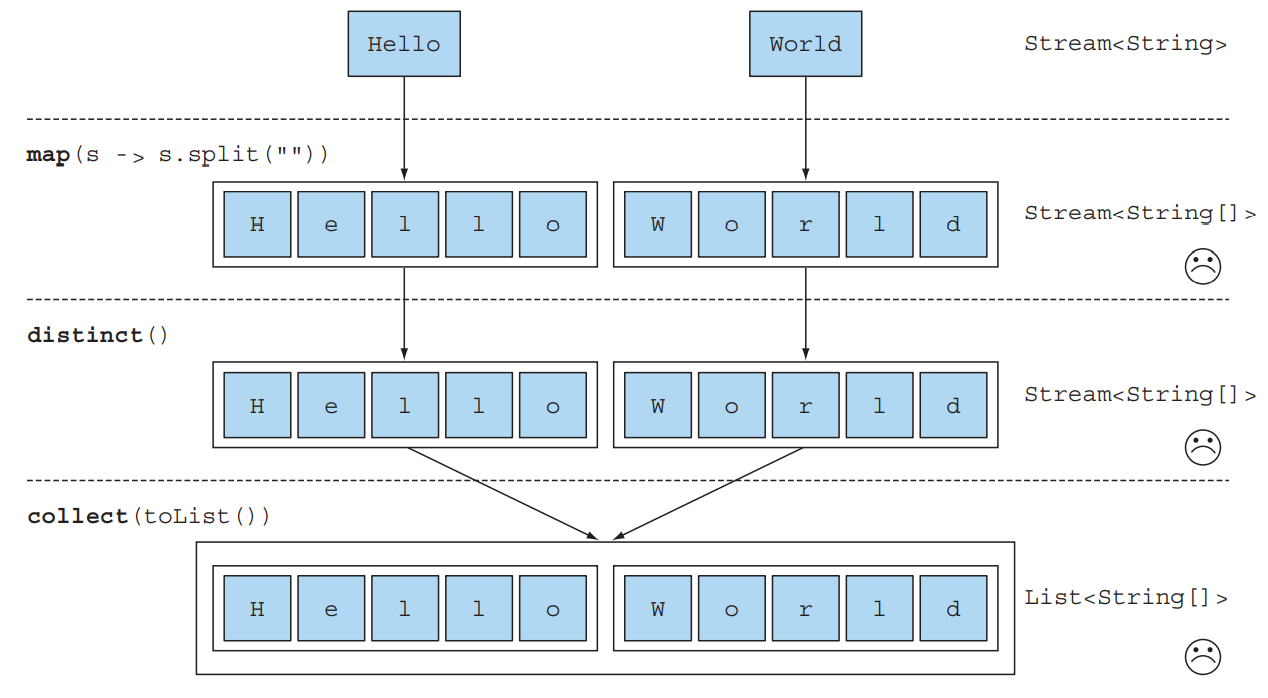
## 

|  |
| --- |
| List<String> words = Arrays.asList("Modern", "Java", "In", "Action");  List<Integer> wordLengths = words.stream()  .map(String::length) //Aplica a cada elemento del flujo  .collect(toList()); |

## 1.3.15 flatmap

Dado un array de palabras se quiere obtener un flujo sobre las letras de las palabras que las componen.

|  |
| --- |
| String[] words = **new** String[]{ "Hello", "World" }  List<String[]> list = words.stream()   .map(word -> word.split("")) //Aplica a cada palabra del array, pero word.split devuelve un array de String, de modo que  // map ha transformado el flujo de Stream<String> a Stream<String[]>  .distinct()   .collect(toList()); |



Existe la posibilidad de uso de Arrays.stream() que convierte un array de elementos en un flujo directo de esos elementos.

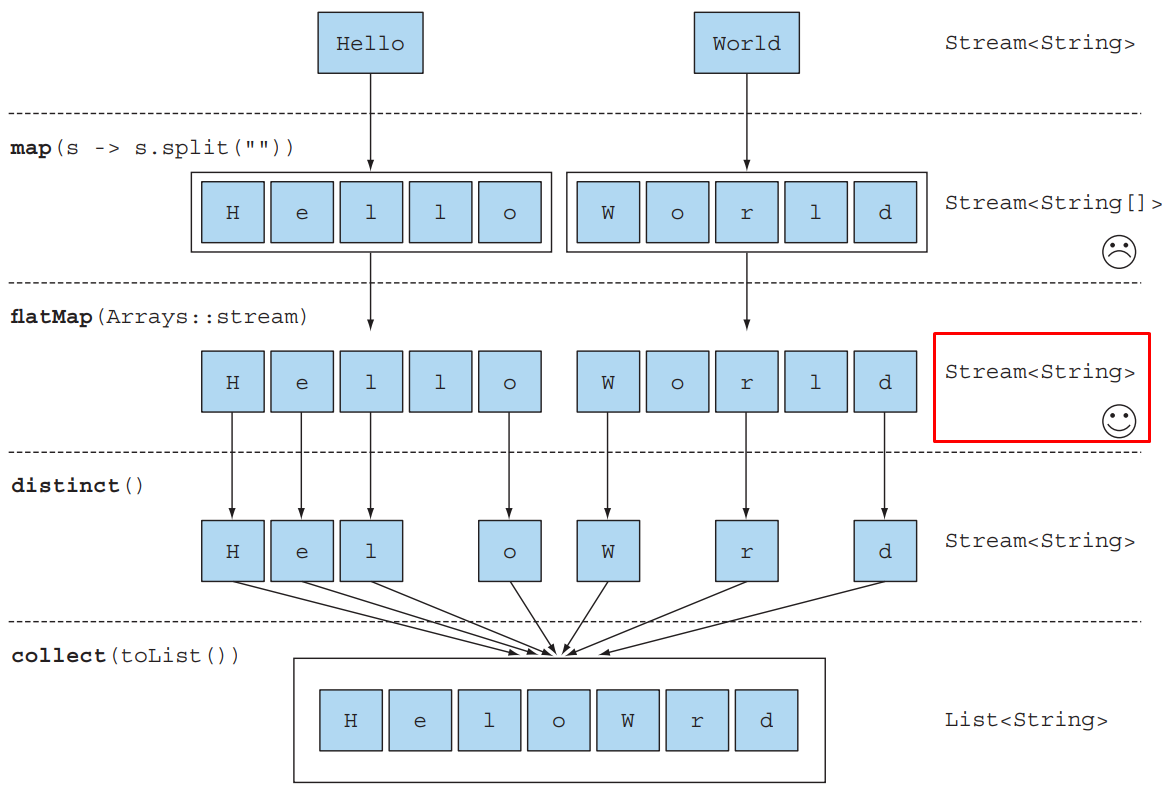
|  |
| --- |
| String[] arrayOfWords = {"Goodbye", "World"};  Stream<String> streamOfwords = Arrays.stream(arrayOfWords); |

aplicando esta función al resultado del map de split tendríamos:

|  |
| --- |
| words.stream()   .map(word -> word.split(""))   .map(Arrays::stream) //Va a convertir cada elemento del stream de tipo array en un stream. Tendremos streams Stream<String> dentro del stream principal.  .distinct()   .collect(toList()); // El resultado será List<Stream<String>> |

El método flatMap le permite concatenar todos los flujos generados en el flujo principal.

|  |
| --- |
| List<String> uniqueCharacters = words.stream()   .map(word -> word.split(""))   .flatMap(Arrays::stream)   .distinct()   .collect(toList()); |



## 1.3.16 allMatch, anyMatch, noneMatch

|  |
| --- |
| **if**(menu.stream().anyMatch(Dish::isVegetarian)) { // anyMatch comprueba que algún elemento cumpla con el predicado devolviendo true en ese caso  //Predicado por referencia a método Dish::isVegetarian  System.out.println("The menu is (somewhat) vegetarian friendly!!");  } |

|  |
| --- |
| **boolean** isHealthy = menu.stream()  .allMatch(dish -> dish.getCalories() < 1000); //allMatch comprueba que todos los elementos cumplan con el predicado devolviendo true en ese caso |

|  |
| --- |
| **boolean** isHealthy = menu.stream()  .noneMatch(d -> d.getCalories() >= 1000); //noneMatch comprueba que ningún elemento cumpla con el predicado, devolviendo true en ese caso |

## 1.3.17 findAny, findFirst (son Finales) y devuelven algo(boolean / obj)

|  |
| --- |
| Optional<Dish> dish = menu.stream()   .filter(Dish::isVegetarian)   .findAny(); //Devuelve alguno, de tipo Optional<T> |

Devuelve un valor de tipo Optional<T> donde T es el tipo del flujo, para prevenir la asignación de null.

<https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html>

Optional<T> presenta los siguientes métodos:

* isPresent() devuelve true si Optional contiene un valor; de lo contrario, devuelve false.
* ifPresent(Consumer<T> block) ejecuta el bloque dado si hay un valor presente (ver Apéndice A: Interfaces funcionales de Java 8)
* T get() devuelve el valor si está presente; de lo contrario, lanza una excepción NoSuchElement.
* T orElse(T otro) devuelve el valor si está presente; de lo contrario, devuelve un valor predeterminado

|  |
| --- |
| menu.stream()  .filter(Dish::isVegetarian)  .findAny() // Devuelve un wrap Optional<Dish>  .ifPresent(dish -> System.out.println(dish.getName());  //Si hay valor ejecuta el lambda de tipo Consumer |

|  |
| --- |
| List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);  Optional<Integer> firstSquareDivisibleByThree = someNumbers.stream()  .map(n -> n \* n)  .filter(n -> n % 3 == 0)  .findFirst(); *// 9* |

findFirst te asegura el primero en streams con paralelización.

## 1.3.18 reduce

### 1.3.18.1 reduce con valor inicial (reduces a un solo valor un conjunto de valores)

|  |  |  |
| --- | --- | --- |
| int sum = 0;  for (int x : numbers) {  sum += x;  }  Bucle de acumulacion | **int** sum = numbers.stream()  .reduce(0, (a, b) -> a + b);  // 0 → valor inicial  // (a, b) -> a + b  operación de reducción | **int** sum = numbers.stream()  .reduce(0, Integer::sum);  //Integer::sum método estático para suma a + b |
| int prod = 1;  for (int x : numbers) {  prod \*= x;  } | **int** product = numbers.stream()  .reduce(1, (a, b) -> a \* b);  // 1 → valor inicial  // (a, b) -> a \* b  operación de reducción | **//**Integer no implementa método estático producto  // a \*b |

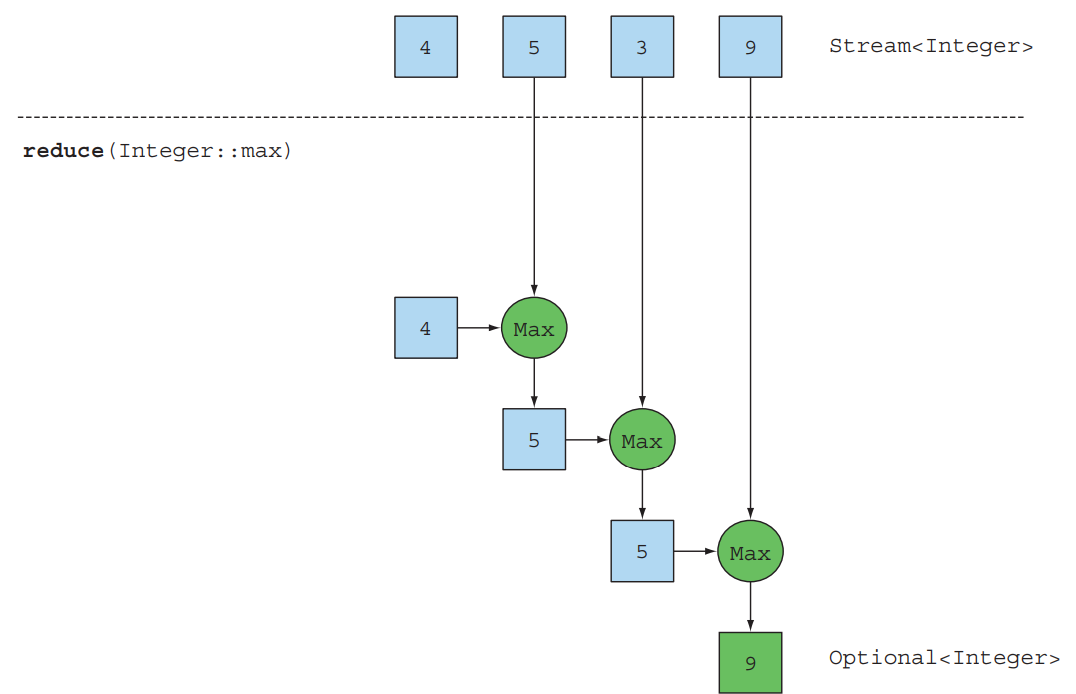
## 

### 1.3.18.2 reduce sin valor inicial

|  |
| --- |
| Optional<Integer> sum = numbers.stream()  .reduce((a, b) -> (a + b));  //Devuelve un Optional<Integer> para el caso en el que el stream no tenga elementos, por carecer de valor inicial |

### 1.3.18.3 reduce a max y min

|  |  |
| --- | --- |
| Optional<Integer> max = numbers.stream()  .reduce((a, b) -> a > b ? a : b); | Optional<Integer> max = numbers.stream()  .reduce(Integer::max); |
| Optional<Integer> max = numbers.stream()  .reduce((a, b) -> a < b ? a : b); | Optional<Integer> min = numbers.stream()  .reduce(Integer::min); |
| Optional<Integer> max = numbers.stream()  .reduce(Integer::max);  //.reduce((a,b) -> Integer.max(a,b)); |  |



## 1.3.19 Streams numéricos primitivos: mapToInt, mapToDouble y mapToLong

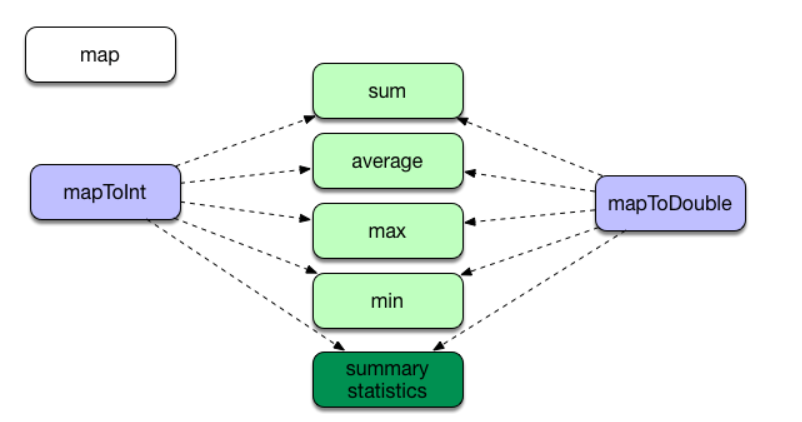
Previene frente al coste de auto-boxing ( conversión automática entre tipo primitivo y objeto) y permite mayor expresividad.

|  |  |
| --- | --- |
| **int** calories = menu.stream()  .map(Dish::getCalories)  //map devuelve un Stream<Integer> general  //en el siguiente paso debe reducirse de forma genérica  .reduce(0, Integer::sum); | **int** calories = menu.stream()  .mapToInt(Dish::getCalories)  //mapToInt devuelve IntStream específico que viene con función sum  //no hace falta reducirla, puede expresarse de forma más declarativa  .sum(); |

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html>

Otras funciones que soporta IntStream aparte de sum son: max, min y average.

|  |
| --- |
| OptionalInt maxCalories = menu.stream()   .mapToInt(Dish::getCalories)  .max(); *//OptionalInt va a evaluar directamente a tipo primitivo int*  **int** max = maxCalories.orElse(1); |



## 1.3.20 Rangos numéricos

|  |  |
| --- | --- |
| int evenNumbers = Stream.iterate(0, n -> n + 1)  .limit(100)  .filter(n -> n % 2 == 0)  .count(); | int evenNumbers = IntStream.rangeClosed(1, 100)  .filter(n -> n % 2 == 0)  .count(); |

## 1.3.21 Ejemplos

|  |
| --- |
| **public** **class** **Trader** {   **private** **final** String nombre;   **private** **final** String ciudad;   **public** **Trader**(String n, String c) {  **this**.nombre = n;  **this**.ciudad = c;  }   **public** String **getNombre**() {  **return** nombre;  }   **public** String **getCiudad**() {  **return** ciudad;  }   **public** String **toString**() {  **return** "Trader:" + **this**.nombre + " in " + **this**.ciudad;  } } |

|  |
| --- |
| **public** **class** **Transaction** {  **private** **final** Trader trader;  **private** **final** **int** anio;   **private** **final** **int** valor;   **public** **Transaction**(Trader trader, **int** anio, **int** valor) {  **this**.trader = trader;  **this**.anio = anio;  **this**.valor = valor;  }   **public** Trader **getTrader**() {  **return** **this**.trader;  }   **public** **int** **getAnio**() {  **return** anio;  }   **public** **int** **getValor**() {  **return** valor;  }   **public** String **toString**() {  **return** "{" + **this**.trader + ", " + "año: " + **this**.anio + ", " + "valor:" + **this**.valor + "}";  } } |

|  |
| --- |
| **import** **static** java.util.stream.Collectors.\*; **import** **static** java.util.Comparator.\*; **import** java.util.Arrays; **import** java.util.List; **import** java.util.Optional; **import** java.util.Set;  **public** **class** **Main** {   **public** **static** **void** **main**(String[] args) {  Trader raoul = **new** Trader("Raul", "Cambridge");  Trader mario = **new** Trader("Mario", "Milan");  Trader alan = **new** Trader("Alan", "Cambridge");  Trader brian = **new** Trader("Brian", "Cambridge");  List<Transaction> transactions = Arrays.asList(**new** Transaction(brian, 2011, 300),  **new** Transaction(raoul, 2012, 1000), **new** Transaction(raoul, 2011, 400),  **new** Transaction(mario, 2012, 710), **new** Transaction(mario, 2012, 700), **new** Transaction(alan, 2012, 950));   *// 1 Encuentre todas las transacciones del año 2011 y ordénelas por valor (menor* *a mayor).*   List<Transaction> tr2011 = transactions.stream()  .filter(transaction -> transaction.getAnio() == 2011)  .sorted(comparing(Transaction::getValor))  .collect(toList());   *// 2 ¿Cuáles son todas las ciudades (sin repetición) donde trabajan los traders?*   List<String> cities = transactions.stream()  .map(transaction -> transaction.getTrader().getCiudad())  .distinct()  .collect(toList());   *// O mediante toSet*  Set<String> ciudades = transactions.stream()  .map(transaction -> transaction.getTrader().getCiudad())  .collect(toSet());   *// 3 Encuentre todos los traders de Cambridge y ordénelos por nombre.*  List<Trader> traders = transactions.stream()  .map(Transaction::getTrader)  .filter(trader -> trader.getCiudad().equals("Cambridge")).distinct()  .sorted(comparing(Trader::getNombre))  .collect(toList());  *// 4 Devuelva los nombres de todos los traders ordenados alfabéticamente en una*  *// sola cadena*   String tradersStr = transactions.stream()  .map(transaction -> transaction.getTrader().getNombre())  .distinct()  .sorted()  .reduce("", (n1, n2) -> n1 + n2);   *// O mediante joining*  tradersStr = transactions.stream()  .map(transaction -> transaction.getTrader().getNombre())  .distinct()  .sorted()  .collect(joining());   *// 5 ¿Hay traders con sede en Milán? Sí o no comprueba el null*  **boolean** milanBased = transactions.stream()  .anyMatch(t ->  If(t.getTrader().getCiudad() == null{  return false;  }else{  return t.getTrader.getCiudad().equals("Milan"));  });  *// 6 Imprime los valores de todas las transacciones de los traders que viven en*  *// Cambridge.*  transactions.stream()  .filter(t -> "Cambridge".equals(t.getTrader().getCiudad()))  .map(Transaction::getValor)  .forEach(System.out::println);   *// 7 ¿Cuál es el valor más alto de todas las transacciones?*  Optional<Integer> highestValue = transactions.stream()  .map(Transaction::getValor)  .reduce(Integer::max);    *// 8 Encuentra la transacción con el valor más pequeño.*  Optional<Transaction> smallestTransaction = transactions.stream()  .reduce((t1, t2) -> t1.getValor() < t2.getValor() ? t1 : t2);    smallestTransaction = transactions.stream()  .min(comparing(Transaction::getValor));   }  } |

## 1.4 collect y agregación

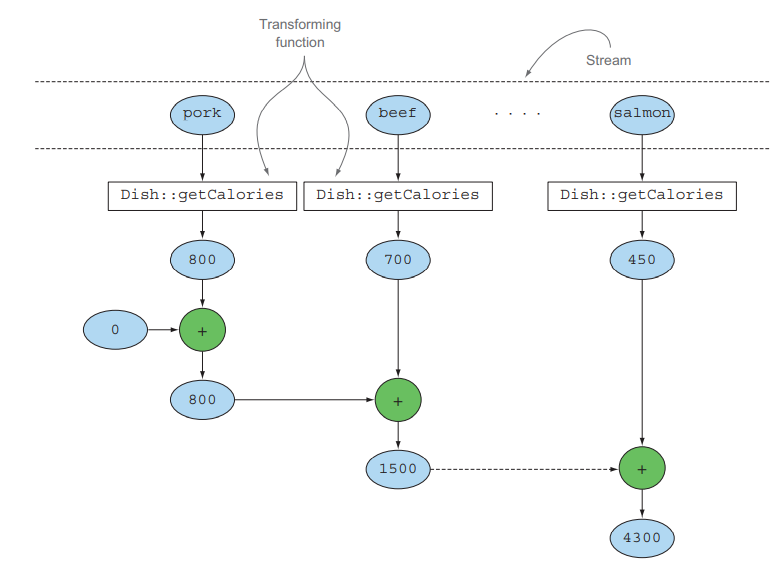
### 1.4.1 collect max/min

|  |
| --- |
| Comparator dishCaloriesComparator = Comparator.comparingInt(Dish::getCalories);  Optional<Dish> mostCalorieDish = menu.stream().collect(maxBy(dishCaloriesComparator)); |

La devolución de Optional es debido a la consideración de streams vacíos sobre los que se aplicará collect.

### 1.4.2 collect sumarización

|  |
| --- |
| **int** totalCalories = menu.stream().collect(summingInt(Dish::getCalories));  double avgCalories = menu.stream().collect(averagingInt(Dish::getCalories));  IntSummaryStatistics menuStatistics = menu.stream()  .collect(summarizingInt(Dish::getCalories));  System.out.println(menuStatistics);  // IntSummaryStatistics{count=9, sum=4300, min=120, average=477.777778, max=800}  String shortMenu = menu.stream().map(Dish::getName).collect(joining(", ")); |



## 1.4.3 groupingBy

## 

|  |
| --- |
| Map<Dish.Type, List<Dish>> dishesByType = menu.stream().collect(groupingBy(Dish::getType)); |

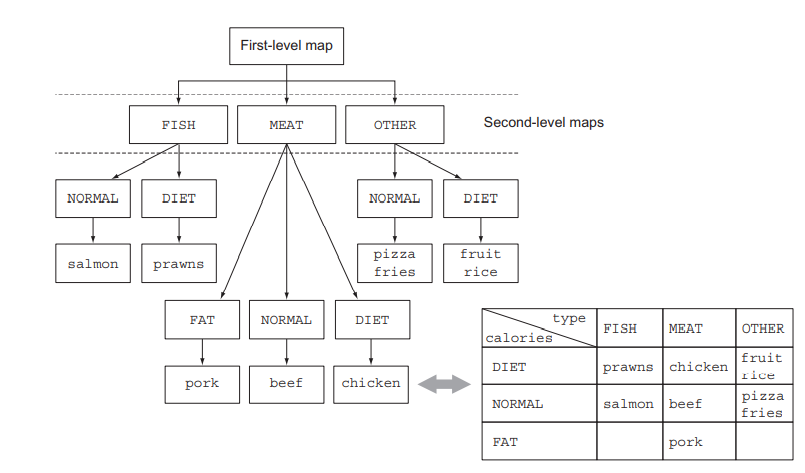
## 

### 1.4.3.1 groupingby lambda

|  |
| --- |
| **public** **enum** CaloricLevel { DIET, NORMAL, FAT };  Map<CaloricLevel, Lis<Dish>> dishesByCaloricLevel = menu.stream()  .collect( groupingBy(dish -> { **if** (dish.getCalories() <= 400) **return** CaloricLevel.DIET;  **else** **if** (dish.getCalories() <= 700) **return** CaloricLevel.NORMAL;  **else** **return** CaloricLevel.FAT; } )); |

### 1.4.2.2 goupingby multinivel

|  |
| --- |
| Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel = menu.stream()  .collect( groupingBy(Dish::getType , groupingBy(dish ->   {  **if** (dish.getCalories() <= 400) **return** CaloricLevel.DIET;  **else** **if** (dish.getCalories() <= 700) **return** CaloricLevel.NORMAL;   **else** **return** CaloricLevel.FAT;   } ) ) );  /\* Resultado mapa a 2 niveles: 1er nivel TYPE y 2o nivel CaloricLevel  {MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]}  , FISH={DIET=[prawns], NORMAL=[salmon]}  , OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}  \*/ |



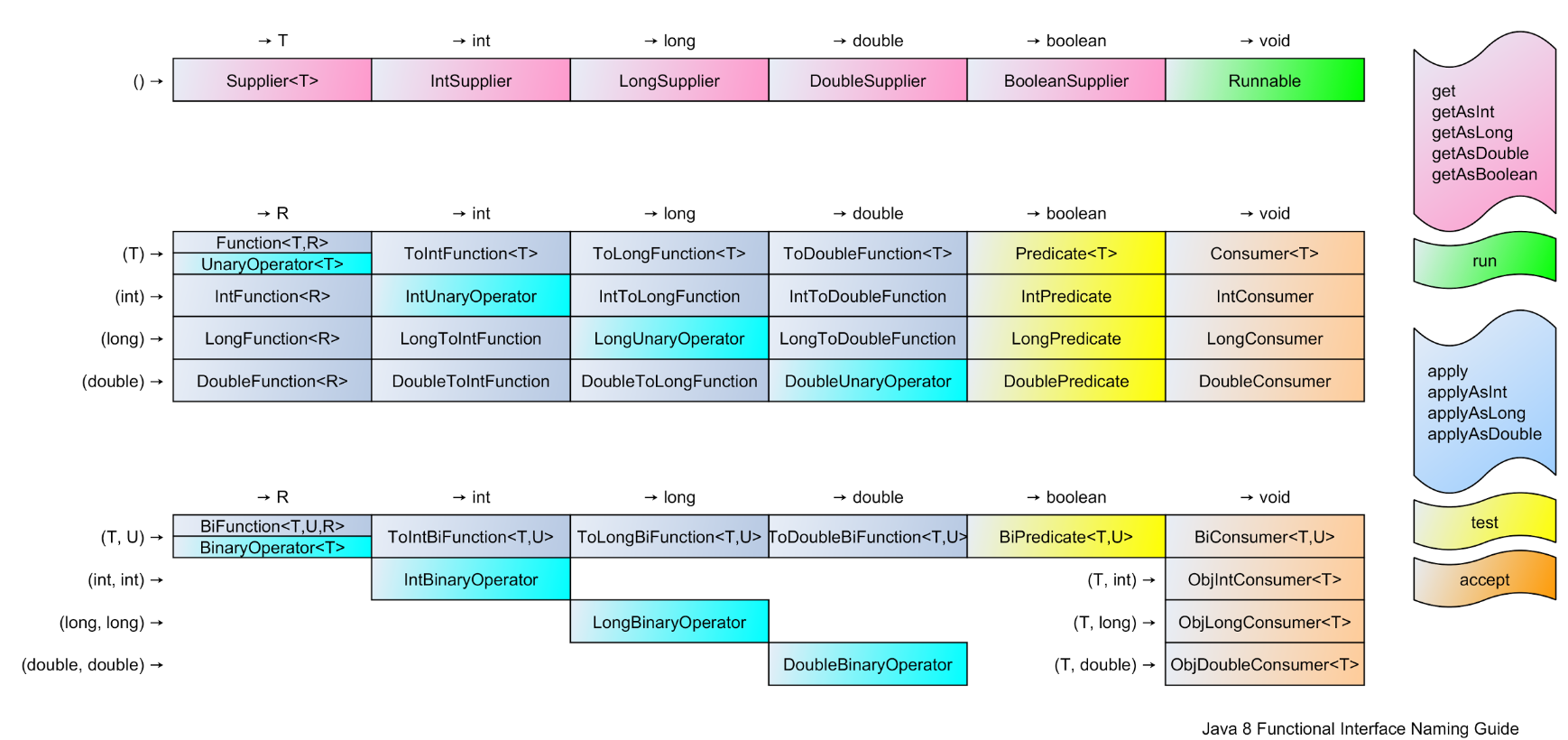
## 1.4.2.3. groupingby y sumarización

Los métodos de agregación de collect se trasladan a groupingBy aplicando su función sobre cada subgrupo.

|  |
| --- |
| Map<Dish.Type, Long> typesCount = menu.stream().collect( groupingBy(Dish::getType, counting()));  */\* {MEAT=3, FISH=2, OTHER=4} \*/*  Map<Dish.Type, Integer> totalCaloriesByType = menu.stream().collect(groupingBy(Dish::getType, summingInt(Dish::getCalories))); |

|  |
| --- |
| Map<Dish.Type, Optional<Dish>> mostCaloricByType = menu.stream()  .collect(groupingBy(Dish::getType, maxBy(comparingInt(Dish::getCalories))));  */\* {FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]} \*/*  *//Se puede eliminar el Optional<Dish> mediante collectingAndThen envolviendo a la función de collect* *//y operando sobre cada uno de los elementos antes de devolverlo*  Map<Dish.Type, Dish> mostCaloricByType = menu.stream() .collect(groupingBy(Dish::getType, collectingAndThen(   maxBy(comparingInt(Dish::getCalories)),   Optional::get))); */\* {FISH=salmon, OTHER=pizza, MEAT=pork} \*/* |

## Apéndice A: Interfaces funcionales de Java 8



## Interfaz de usuario gráfica Descripción generada automáticamenteApéndice B: Operaciones intermedias y terminales

## Apéndice C: sorted & Comparator

|  |
| --- |
| Recuerda que los métodos List.sort y Collections.sort son destructivos con la colección original.  En cambio el método sorted de un stream tan sólo devuelve un stream ordenado. |

|  |
| --- |
| Comparator<Apple> c = Comparator.comparing(Apple::getWeight); *//Comparador de manzanas por peso en orden ascendente*  appleList.sort(c); *//Lista inventario de manzanas ordenada por peso ascendente* |

De forma más compacta si quiero ordenar la lista de manzanas por orden descendente:

|  |
| --- |
| appleList.sort( comparing(Apple::getWeight).reversed() ); |

Por último si quiero ordenar la lista de manzanas por orden descendente de peso y además por país de origen para manzanas del mismo peso:

|  |
| --- |
| inventory.sort(comparing(Apple::getWeight) *//Comparator inicial por orden ascendente (por defecto) de peso*  .reversed()  *//Se invierte el orden a descendente*  .thenComparing(Apple::getCountry)); *//Se añade otro campo de ordenación por orden ascendente (por defecto).* |

## Apéndice D: Composición de predicados

|  |
| --- |
| Predicate<Apple> redApples = (Apple apple) -> RED.equals(apple.getColor()); // Se define una función predicado (T) -> boolean y a partir de ella se pueden realizar las operaciones lógicas habituales añadiendo más predicados  Predicate<Apple> notRedApples = redApples.negate();  Predicate redAndHeavyApples = redApples.and(apple -> apple.getWeight() > 150);  Predicate redAndHeavyAppleOrGreen = redApple.and(apple -> apple.getWeight() > 150)   .or(apple -> GREEN.equals(a.getColor())); |

Reglas de paréntesis de composición de predicados:

* a.or(b).and(c) se lee (a || b) && c
* a.and(b).or(c) se lee (a && b) || c

## Apéndice F: Código base para probar los ejemplos

Dish.java

|  |
| --- |
| **package** org.iesvegademijas.ejercicios\_stream;  **public** **class** **Dish** {  **private** **final** String name;  **private** **final** **boolean** vegetarian;  **private** **final** **int** calories;  **private** **final** Type type;   **public** **Dish**(String name, **boolean** vegetarian, **int** calories, Type type) {  **this**.name = name;  **this**.vegetarian = vegetarian;  **this**.calories = calories;  **this**.type = type;  }   **public** String **getName**() {  **return** name;  }   **public** **boolean** **isVegetarian**() {  **return** vegetarian;  }   **public** **int** **getCalories**() {  **return** calories;  }   **public** Type **getType**() {  **return** type;  }   **@Override**  **public** String **toString**() {  **return** name;  }   **public** **enum** Type {  MEAT, FISH, OTHER  } } |

Main.java

|  |
| --- |
| **package** org.iesvegademijas.ejercicios\_stream;  **import** java.util.Arrays; **import** java.util.List; **import** java.util.Optional;  **import** **static** java.util.Comparator.\*; **import** **static** java.util.stream.Collectors.\*;  **public** **class** **Main** {   **public** **static** **void** **main**(String[] args) {    List<Dish> menu = Arrays.asList(   **new** Dish("pork", **false**, 800, Dish.Type.MEAT),  **new** Dish("beef", **false**, 700, Dish.Type.MEAT),  **new** Dish("chicken", **false**, 400, Dish.Type.MEAT),  **new** Dish("french fries", **true**, 530, Dish.Type.OTHER),  **new** Dish("rice", **true**, 350, Dish.Type.OTHER),  **new** Dish("season fruit", **true**, 120, Dish.Type.OTHER),  **new** Dish("pizza", **true**, 550, Dish.Type.OTHER),  **new** Dish("prawns", **false**, 300, Dish.Type.FISH),  **new** Dish("salmon", **false**, 450, Dish.Type.FISH) );      *//Prueba aquí tus streams:*  List<String> lowCaloricDishesName = menu  .stream()  .filter(d -> d.getCalories() < 400)  .sorted(comparing(Dish::getCalories))  .map(Dish::getName)  .collect(toList());    *//Visualiza el resultado aquí*  System.out.println(lowCaloricDishesName);    }   } |