Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

Lambdas y flujos de Java SE 8

17

Oh, podría fluir como tú,
y hacer de tu corriente
mi gran ejemplo,
¡puesto que es mi tema!

—Sir John Denham

Objetivos

En este capítulo aprenderá:

- Lo que es la programación funcional y cómo complementa la programación orientada a objetos.
- A utilizar la programación funcional para simplificar las tareas de programación que ha realizado con otras técnicas.
- A escribir expresiones lambda que implementen interfaces funcionales.
- Lo que son los flujos y cómo se forman las canalizaciones de flujo a partir de los orígenes de flujos, operaciones intermedias y operaciones terminales.
- A realizar operaciones con objetos `IntStream`, incluyendo `forEach`, `count`, `min`, `max`, `sum`, `average`, `reduce`, `filter` y `sorted`.
- A realizar operaciones con objetos `Stream`, incluyendo `filter`, `map`, `sorted`, `collect`, `forEach`, `findFirst`, `distinct`, `mapToDouble` y `reduce`.
- A crear flujos que representen rangos de valores `int` y valores `int` aleatorios.

Plan general

17.1 Introducción

La forma en que piensa sobre la programación en Java está a punto de cambiar drásticamente. Antes de Java SE 8, el lenguaje Java soportaba tres paradigmas de programación: *programación por procedimientos*, *programación orientada a objetos* y *programación genérica*. Java SE 8 agrega la *programación funcional*. El nuevo lenguaje y las herramientas de biblioteca que soportan este paradigma se agregaron a Java como parte del *proyecto Lambda*:

```
http://openjdk.java.net/projects/lambda
```

En este capítulo definiremos la programación funcional y mostraremos cómo usarla para escribir programas de una manera más rápida, concisa y con menos errores que los programas escritos con las técnicas anteriores. En el capítulo 23 (en inglés) verá que los programas funcionales son más fáciles de *paralelizar* (es decir, realizar varias operaciones al mismo tiempo), de modo que sus programas puedan aprovechar las arquitecturas multinúcleo para mejorar el rendimiento. Antes de leer este capítulo le recomendamos que repase la

sección 10.10 en donde se introdujeron las nuevas características de las interfaces de Java SE 8 (la habilidad de incluir métodos `default` y `static`) y se habló sobre el concepto de las interfaces funcionales.

En este capítulo presentamos muchos ejemplos de programación funcional que a menudo muestran formas más simples de implementar las tareas que ya se programaron en capítulos anteriores (figura 17.1).

Temas previos a Java SE 8	Explicaciones y ejemplos correspondientes de Java SE 8
Capítulo 7, Arreglos y objetos `ArrayList`	Las secciones 17.3 y 17.4 introducen las herramientas básicas de lambdas y flujos que procesan arreglos unidimensionales.
Capítulo 10, Programación orientada a objetos: polimorfismo e interfaces	La sección 10.10 introdujo las nuevas características de las interfaces de Java SE 8 (métodos `default`, métodos `static` y el concepto de las interfaces funcionales) que soportan la programación funcional.
Capítulo 12, Componentes de la GUI: parte 1	La sección 17.9 muestra cómo usar un lambda para implementar una interfaz funcional de escucha de eventos en Swing.
Capítulo 14, Cadenas, caracteres y expresiones regulares	La sección 17.5 muestra cómo usar lambdas y flujos para procesar colecciones de objetos `String`.
Capítulo 15, Archivos, flujos y serialización de objetos	La sección 17.7 muestra cómo usar lambdas y flujos para procesar líneas de texto de un archivo.
Capítulo 22 (en inglés, en el sitio web del libro), GUI Components: Part 2	Habla sobre el uso de lambdas para implementar interfaces funcionales de escucha de eventos en Swing.
Capítulo 23 (en inglés, en el sitio web del libro), Concurrency	Muestra que los programas funcionales son más fáciles de paralelizar, de modo que puedan aprovechar las arquitecturas multinúcleo para mejorar el rendimiento. Demuestra el procesamiento de flujos en paralelo. Muestra que el método `parallelSort` de `Arrays` mejora el desempeño en arquitecturas multinúcleo al almacenar arreglos grandes.
Capítulo 25 (en inglés, en el sitio web del libro), Java FX GUI: Part 1	Habla sobre el uso de lambdas para implementar las interfaces funcionales de escucha de eventos en JavaFX.

Fig. 17.1 | Explicaciones y ejemplos sobre lambdas y flujos de Java SE 8.

17.2 Generalidades acerca de las tecnologías de programación funcional

En los capítulos anteriores aprendió varias técnicas de programación por procedimientos, orientada a objetos y genérica. Aunque usó con frecuencia clases e interfaces de la biblioteca de Java para realizar varias tareas, por lo general debía determinar *qué* deseaba lograr en una tarea y luego especificar de manera precisa *cómo* lograrlo. Por ejemplo, vamos a suponer que lo *que* desea lograr es sumar los elementos de un arreglo llamado `valores` (el *origen de datos*). Podría usar el siguiente código:

```
int suma = 0;
for (int contador = 0; contador < valores.length; contador++)
   suma += valores[contador];
```

Este ciclo especifica *cómo* nos gustaría sumar el valor de cada elemento del arreglo a `suma`: con una instrucción de repetición `for` que procese cada elemento a la vez, sumando el valor de cada elemento a la variable `suma`. Esta técnica de iteración se conoce como **iteración externa** (porque especifica cómo iterar,

no sólo la biblioteca) y requiere que se acceda a los elementos en forma secuencial de principio a fin en un solo hilo de ejecución. Para realizar la tarea anterior también hay que crear dos variables (`suma` y `contador`) que *muten* repetidas veces (es decir, que sus valores cambien) mientras se realiza la tarea. Ya ha realizado muchas tareas similares con arreglos y colecciones, como visualizar los elementos de un arreglo, sintetizando las caras de un dado que se tiró 6,000,000 de veces, calcular el promedio de los elementos de un arreglo y más.

La iteración externa es propensa a errores

La mayoría de los programadores de Java se sienten cómodos con la iteración externa. Sin embargo existen en ésta varias oportunidades de error. Por ejemplo, podría inicializar la variable `suma` de manera incorrecta, inicializar la variable de control `contador` de manera incorrecta, usar la condición de continuación de ciclo equivocada, incrementar la variable de control `contador` de manera incorrecta o sumar incorrectamente cada valor en el arreglo a la `suma`.

Iteración interna

En la **programación funcional**, el programador especifica *qué* quiere realizar en una tarea, pero *no cómo* lograrlo. Como veremos en este capítulo, para sumar los elementos de un origen de datos numérico (como los de un arreglo o colección), puede usar las nuevas herramientas de la biblioteca de Java SE 8 que le permiten decir, "he aquí un origen de datos, dame la suma de sus elementos". *No* necesita especificar *cómo* iterar a través de los elementos *ni* declarar y usar variables mutables. Esto se conoce como **iteración interna**, ya que la *biblioteca* determina cómo acceder a todos los elementos para realizar la tarea. Con la iteración interna, se puede decir fácilmente a la biblioteca que desea realizar esta tarea con *procesamiento paralelo* para aprovechar la arquitectura multinúcleo de su computadora; esto puede mejorar de manera considerable el rendimiento de la tarea. Como veremos en el capítulo 23, es difícil crear tareas paralelas que operen correctamente si esas tareas modifican la información del estado de un programa (es decir, los valores de sus variables). Por ende, las herramientas de programación funcional que aprenderá a usar aquí se enfocan en la **inmutabilidad** y no en modificar el origen de datos que se está procesando o cualquier otro estado del programa.

17.2.1 Interfaces funcionales

En la sección 10.10 se introdujeron las nuevas características de interfaces de Java SE 8 (métodos `default` y métodos `static`) y se vio el concepto de una *interfaz funcional*: una interfaz que contiene sólo un método `abstract` (también puede contener métodos `static` y `default`). Dichas interfaces se conocen también como interfaces de *un solo método abstracto* (SAM). Las interfaces funcionales se usan mucho en la programación funcional, ya que actúan como un modelo orientado a objetos para una función.

Interfaces funcionales en el paquete `java.util.function`

El paquete `java.util.function` contiene varias interfaces funcionales. En la figura 17.2 se muestran las seis interfaces funcionales genéricas básicas. En la tabla, `T` y `R` son nombres de tipos genéricos que representan el tipo del objeto con el que opera la interfaz funcional y el tipo de valor de retorno de un método, respectivamente. Hay muchas otras interfaces funcionales en el paquete `java.util.function` que son versiones especializadas de las de la figura 17.2. La mayoría son para usarse con valores primitivos `int`, `long` y `double`, pero también hay personalizaciones genéricas de `Consumer`, `Function` y `Predicate` para operaciones binarias; es decir, métodos que reciben dos argumentos.

Interfaz	Descripción
`BinaryOperator<T>`	Contiene el método `apply` que recibe dos argumentos, realiza una operación sobre ellos (como un cálculo) y devuelve un valor de tipo `T`. En la sección 17.3 verá varios ejemplos de `BinaryOperator`.
`Consumer<T>`	Contiene el método `accept` que recibe un argumento `T` y devuelve `void`. Realiza una tarea con su argumento `T`, como mostrar el objeto en pantalla, invocar a un método del objeto, etc. Verá varios ejemplos de `Consumer` a partir de la sección 17.3.
`Function<T,R>`	Contiene el método `apply` que recibe un argumento `T` y devuelve el resultado de ese método. Verá varios ejemplos de `Function` a partir de la sección 17.5.
`Predicate<T>`	Contiene el método `test` que recibe un argumento `T` y devuelve un `boolean`. Verá varios ejemplos de `Predicate` a partir de la sección 17.3.
`Supplier<T>`	Contiene el método `get` que no recibe argumentos y produce un valor de tipo `T`. A menudo se usa para crear un objeto colección en donde se colocan los resultados de la operación de un flujo. Verá varios ejemplos de `Supplier` a partir de la sección 17.7.
`UnaryOperator<T>`	Contiene el método `get` que no recibe argumentos y devuelve un valor de tipo `T`. Verá varios ejemplos de `UnaryOperator` a partir de la sección 17.3.

Fig. 17.2 | Las seis interfaces funcionales genéricas básicas en el paquete `java.util.function`.

17.2.2 Expresiones lambda

La programación funcional se logra con las expresiones lambda. Una **expresión lambda** representa a un *método anónimo*; es decir, una notación abreviada para implementar una interfaz funcional, similar a una clase interna anónima (sección 12.11). El tipo de una expresión lambda es el tipo de la interfaz funcional que implementa esa expresión lambda. Las expresiones lambda pueden usarse en cualquier parte en donde se esperan interfaces funcionales. De aquí en adelante nos referiremos a las expresiones lambda simplemente como lambdas. Le mostraremos la sintaxis básica de las lambdas en esta sección y hablaremos sobre sus características adicionales a medida que las utilicemos en este capítulo y en los capítulos posteriores.

Sintaxis de una lambda

Una lambda consiste en una *lista de parámetros* seguida del **token flecha (->)** y un cuerpo, como en:

```
(listaParámetros) -> {instrucciones}
```

La siguiente lambda recibe dos valores `int` y devuelve su suma:

```
(int x, int y) -> {return x + y;}
```

En este caso, el cuerpo es un *bloque de instrucciones* que puede contener *una o más* instrucciones encerradas entre llaves. Hay diversas variaciones de esta sintaxis. Por ejemplo, por lo general pueden omitirse los tipos de parámetros, como en:

```
(x, y) -> {return x + y;}
```

en cuyo caso, el compilador determina los tipos de los parámetros y del valor de retorno según el contexto de la lambda; hablaremos más sobre esto después.

Cuando el cuerpo contiene sólo una expresión, se pueden omitir la palabra clave `return` y las llaves, como en:

```
(x, y) -> x + y
```

en este caso, el valor de la expresión se devuelve *implícitamente*. Cuando la lista de parámetros contiene sólo un parámetro, se pueden omitir los paréntesis, como en:

```
valor -> System.out.printf("%d ", valor)
```

Para definir una lambda con una lista de parámetros vacía, especifique la lista de parámetros como paréntesis vacíos a la izquierda del token flecha (->), como en:

```
() -> System.out.println("Bienvenido a los lambdas!")
```

Además de la sintaxis anterior de las lambdas, hay formas abreviadas especializadas de lambdas que se conocen como *referencias a métodos*, las cuales presentaremos en la sección 17.5.1.

17.2.3 Flujos

Java SE 8 introduce el concepto de **flujos**, que son similares a los iteradores que vimos en el capítulo 16. Los flujos son objetos de clases que implementan a la interfaz `Stream` (del paquete `java.util.stream`) o una de las interfaces de flujo especializadas para procesar colecciones de valores `int`, `long` o `double` (que presentaremos en la sección 17.3). En conjunto con las lambdas, los flujos le permiten realizar tareas sobre colecciones de elementos, a menudo de un objeto arreglo o colección.

Canalizaciones de flujo

Los flujos desplazan elementos a través de una secuencia de pasos de procesamiento (lo que se conoce como una **canalización de flujo**) la cual comienza con un *origen de datos* (como un arreglo o colección), realiza varias *operaciones intermedias* sobre los elementos del origen de datos y finaliza con una *operación terminal*. Una canalización de flujo se forma mediante el *encadenamiento* de llamadas a métodos. A diferencia de las colecciones, los flujos *no* tienen su propio almacenamiento; una vez que se procesa un flujo, no puede reutilizarse debido a que no mantiene una copia del origen de datos original.

Operaciones intermedias y terminal

Una **operación intermedia** especifica las tareas a realizar sobre los elementos del flujo y siempre produce un nuevo flujo. Las operaciones intermedias son **perezosas**; es decir, no se ejecutan sino hasta que se invoque a una operación terminal. Esto permite a los desarrolladores de bibliotecas optimizar el rendimiento del procesamiento de flujos. Por ejemplo, si tiene una colección de 1,000,000 de objetos `Persona` y busca el *primero* con el apellido "`Jones`", el procesamiento del flujo puede terminar tan pronto como se encuentre dicho objeto `Persona`.

Una **operación terminal** inicia el procesamiento de las operaciones intermedias de una canalización de flujo y produce un resultado. Las operaciones terminales son **ansiosas**, ya que realizan la operación solicitada cuando se les invoca. Hablaremos más sobre las operaciones perezosas y ansiosas a media que las veamos en el capítulo; el lector verá cómo es que las operaciones perezosas pueden mejorar el rendimiento. La figura 17.3 muestra algunas operaciones intermedias comunes. La figura 17.4 muestra algunas operaciones terminales comunes.

Operaciones intermedias con flujos	
`filter`	Produce un flujo que contiene sólo los elementos que satisfacen una condición.
`distinct`	Produce un flujo que contiene sólo los elementos únicos.
`limit`	Produce un flujo con el número especificado de elementos a partir del inicio del flujo original.
`map`	Produce un flujo en el que cada elemento del flujo original está asociado a un nuevo valor (posiblemente de un tipo distinto); por ejemplo, asociar valores numéricos a los cuadrados de los valores numéricos. El nuevo flujo tiene el mismo número de elementos que el flujo original.
`sorted`	Produce un flujo en el que los elementos están ordenados. El nuevo flujo tiene el mismo número de elementos que el flujo original.

Fig. 17.3 | Operaciones intermedias comunes con `Stream`.

Operaciones terminales con `Stream`	
`forEach`	Realiza un procesamiento sobre cada elemento en un flujo (por ejemplo, mostrar cada elemento en pantalla).
***Operaciones de reducción**: toman todos los valores en el flujo y devuelven un solo valor*	
`average`	Calcula el *promedio* de los elementos en un flujo numérico.
`count`	Devuelve el *número de elementos* en el flujo.
`max`	Localiza el valor *más grande* en un flujo numérico.
`min`	Localiza el valor *más pequeño* en un flujo numérico.
`reduce`	Reduce los elementos de una colección a un *solo valor* mediante el uso de una función de acumulación asociativa (por ejemplo, una lambda que suma dos elementos).
***Operaciones de reducción mutables:** crean un contenedor (como una colección o un `StringBuilder`)*	
`collect`	Crea una *nueva colección* de elementos que contienen los resultados de las operaciones anteriores del flujo.
`toArray`	Crea un *arreglo* que contiene los resultados de las operaciones anteriores del flujo.
Operaciones de búsqueda	
`findFirst`	Encuentra el *primer* elemento del flujo con base en las operaciones intermedias; termina inmediatamente el procesamiento de la canalización de flujo una vez que se encuentra dicho elemento.
`findAny`	Encuentra *cualquier* elemento de flujo con base en las operaciones intermedias anteriores; termina de inmediato el procesamiento de la canalización de flujo una vez que se encuentra dicho elemento.
`anyMatch`	Determina si *alguno* de los elementos del flujo coincide con una condición especificada; cuando un elemento coincide, ésta termina de inmediato el procesamiento de la canalización de flujo.
`allMatch`	Determina si *todos* los elementos en el flujo coinciden con una condición especificada.

Fig. 17.4 | Operaciones terminales comunes con `Stream`.

Flujo en procesamiento de archivos comparado con flujo en programación funcional

En este capítulo usamos el término *flujo* en el contexto de la programación funcional; éste no es el mismo concepto que el de los flujos de E/S que vimos en el capítulo 15 en donde un programa lee un flujo de bytes

de un archivo o envía un flujo de bytes a un archivo. Como veremos en la sección 17.7, también puede usar la programación funcional para manipular el contenido de un archivo.

17.3 Operaciones con `IntStream`

[Esta sección demuestra cómo pueden usarse las lambdas y los flujos para simplificar las tareas de programación que vio en el capítulo 7, Arreglos y objetos `ArrayList`].

La figura 17.5 demuestra las operaciones sobre un `IntStream` (paquete `java.util.stream`): un flujo especializado para manipular valores `int`. Las técnicas que se muestran en este ejemplo también se aplican a los flujos `LongStream` y `DoubleStream` para valores `long` y `double`, respectivamente.

```
1  // Fig. 17.5: OperacionesIntStream.java
2  // Demostración de las operaciones con IntStream.
3  import java.util.Arrays;
4  import java.util.stream.IntStream;
5
6  public class OperacionesIntStream
7  {
8     public static void main(String[] args)
9     {
10       int[] valores = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
11
12       // muestra los valores originales
13       System.out.print("Valores originales: ");
14       IntStream.of(valores)
15             .forEach(valor -> System.out.printf("%d ", valor));
16       System.out.println();
17
18       // cuenta, min, max, suma y promedio de los valores
19       System.out.printf("%nCuenta: %d%n", IntStream.of(valores).count());
20       System.out.printf("Min: %d%n",
21          IntStream.of(valores).min().getAsInt());
22       System.out.printf("Max: %d%n",
23          IntStream.of(valores).max().getAsInt());
24       System.out.printf("Suma: %d%n", IntStream.of(valores).sum());
25       System.out.printf("Promedio: %.2f%n",
26          IntStream.of(valores).average().getAsDouble());
27
28       // suma de valores con el método reduce
29       System.out.printf("%nSuma mediante el metodo reduce: %d%n",
30          IntStream.of(valores)
31             .reduce(0, (x, y) -> x + y));
32
33       // suma de cuadrados de los valores con el método reduce
34       System.out.printf("Suma de cuadrados mediante el metodo reduce: %d%n",
35          IntStream.of(valores)
36             .reduce(0, (x, y) -> x + y * y));
```

Fig. 17.5 | Demostración de las operaciones con `IntStream` (parte 1 de 2).

```
37
38          // producto de los valores con el método reduce
39          System.out.printf("Producto mediante el metodo reduce: %d%n",
40             IntStream.of(valores)
41                   .reduce(1, (x, y) -> x * y));
42
43          // valores pares mostrados en orden
44          System.out.printf("%nValores pares mostrados en orden: ");
45          IntStream.of(valores)
46                .filter(valor -> valor % 2 == 0)
47                .sorted()
48                .forEach(valor -> System.out.printf("%d ", valor));
49          System.out.println();
50
51          // valores impares multiplicados por 10 y mostrados en orden
52          System.out.printf(
53             "Valores impares multiplicados por 10 y mostrados en orden: ");
54          IntStream.of(valores)
55                .filter(valor -> valor % 2 != 0)
56                .map(valor -> valor * 10)
57                .sorted()
58                .forEach(valor -> System.out.printf("%d ", valor));
59          System.out.println();
60
61          // suma el rango de enteros del 1 al 10, exclusivo
62          System.out.printf("%nSuma de enteros del 1 al 9: %d%n",
63             IntStream.range(1, 10).sum());
64
65          // suma el rango de enteros del 1 al 10, inclusivo
66          System.out.printf("Suma de enteros del 1 al 10: %d%n",
67             IntStream.rangeClosed(1, 10).sum());
68       }
69    } // fin de la clase OperacionesIntStream
```

```
Valores originales: 3 10 6 1 4 8 2 5 9 7

Cuenta: 10
Min: 1
Max: 10
Suma: 55
Promedio: 5.50

Suma mediante el metodo reduce: 55
Suma de cuadrados mediante el metodo reduce: 385
Producto mediante el metodo reduce: 3628800

Valores pares mostrados en orden: 2 4 6 8 10
Valores impares multiplicados por 10 y mostrados en orden: 10 30 50 70 90

Suma de enteros del 1 al 9: 45
Suma de enteros del 1 al 10: 55
```

Fig. 17.5 | Demostración de las operaciones con `IntStream` (parte 2 de 2).

17.3.1 Creación de un `IntStream` e impresión en pantalla de sus valores con la operación terminal `forEach`

El método `static` **`of`** de `IntStream` (línea 14) recibe un arreglo `int` como argumento y devuelve un `IntStream` para procesar los valores del arreglo. Una vez que creamos un flujo, es posible *encadenar* varias llamadas a métodos para crear una *canalización de flujo*. La instrucción en las líneas 14 y 15 crea un `IntStream` para el arreglo `valores` y luego usa el método **`forEach`** de `IntStream` (una operación terminal) para ejecutar una tarea sobre cada elemento del flujo. El método `forEach` recibe como argumento un objeto que implementa a la interfaz funcional **`IntConsumer`** (paquete **`java.util.function`**); ésta es una versión específica para valores `int` de la interfaz funcional genérica `Consumer`. El método **`accept`** de esta interfaz recibe un valor `int` y ejecuta una tarea con él; en este caso, muestra en pantalla el valor y el espacio. Antes de Java SE 8, por lo general se implementaba la interfaz `IntConsumer` mediante el uso de una clase interna anónima como:

```
new IntConsumer()
{
   public void accept(int valor)
   {
      System.out.printf("%d ", valor);
   }
}
```

pero en Java SE 8 simplemente se escribe la lambda

```
valor -> System.out.printf("%d ", valor)
```

El nombre del parámetro (`valor`) del método `accept` se convierte en el parámetro de la lambda y la instrucción del cuerpo del método `accept` se convierte en el cuerpo de la expresión lambda. Como puede ver, la sintaxis de la lambda es más clara y concisa que la clase interna anónima.

Inferencia de tipos y el tipo de destino de una lambda

Por lo general el compilador de Java puede *inferir* los tipos de los parámetros de una lambda y el tipo devuelto por una lambda a partir del contexto en el que se utilice. Esto se determina mediante el **tipo de destino** de la lambda, que es la interfaz funcional que se espera en donde aparece la lambda en el código. En la línea 15 el tipo de destino es `IntConsumer`. En este caso se *infiere* que el tipo del parámetro de la lambda es `int`, ya que el método `accept` de la interfaz `IntConsumer` espera recibir un `int`. Puede declarar *explícitamente* el tipo del parámetro, como en:

```
(int valor) -> System.out.printf("%d ", valor)
```

Al hacerlo, la lista de parámetros de la lambda *debe* encerrarse entre paréntesis. Por lo general dejamos que el compilador *infiera* el tipo del parámetro de la lambda en nuestros ejemplos.

Variables locales `final`, variables locales efectivamente `final` y lambdas de captura

Antes de Java SE 8, al implementar una clase interna anónima era posible usar variables locales del método circundante (lo que se conoce como *alcance léxico*), pero había que declarar esas variables locales como `final`. Las lambdas también pueden usar variables locales `final`. En Java SE 8, las clases internas anónimas y las lambdas también pueden usar **variables locales efectivamente `final`**; es decir, variables locales que *no* se modifican después de declararse por primera vez e inicializarse. Una lambda que hace referencia a una variable local en el alcance léxico circundante se conoce como **lambda de captura**. El compilador captura el valor de la variable local y se asegura de que el valor pueda usarse cuando se ejecute la lambda *en un momento dado*, lo cual puede ser *después* de que su alcance léxico *deje de existir*.

Uso de this en una lambda que aparece en un método de instancia

Como en una clase interna anónima, una lambda puede usar la referencia `this` de la clase externa. En una clase interna anónima hay que usar la sintaxis *NombreClaseExterna*`.this`; de lo contrario, la referencia `this` se referiría al *objeto de la clase interna anónima*. En una lambda, se hace referencia al objeto de la clase externa simplemente como `this`.

Nombres de parámetros y variables en una lambda

Los nombres de los parámetros y las variables que se utilizan en las lambdas no pueden ser iguales a los de cualquier otra variable local en el alcance léxico de la lambda; de lo contrario, se produce un error de compilación.

17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average`

La clase `IntStream` proporciona varias operaciones terminales para reducciones de flujo comunes en flujo de valores `int`. Las operaciones terminales son *ansiosas*, ya que procesan de inmediato los elementos en el flujo. Las operaciones de reducción comunes para los `IntStream` son:

- **`count`** (línea 19) devuelve el número de elementos en el flujo.
- **`min`** (línea 21) devuelve el `int` más pequeño en el flujo.
- **`max`** (línea 23) devuelve el `int` más grande en el flujo.
- **`sum`** (línea 24) devuelve la suma de todos los valores `int` en el flujo.
- **`average`** (línea 26) devuelve un **`OptionalDouble`** (paquete `java.util`) que contiene el promedio de los valores `int` en el flujo como un valor de tipo `double`. Para cualquier flujo, es posible que *no* haya *elementos* en el flujo. Al devolver `OptionalDouble` el método `average` puede devolver el promedio si el flujo contiene *al menos un elemento*. En este ejemplo sabemos que el flujo tiene 10 elementos, por lo que llamamos al método **`getAsDouble`** de la clase `OptionalDouble` para obtener el promedio. Si no hubiera *elementos*, el `OptionalDouble` no contendría el promedio y `getAsDouble` lanzaría una excepción `NoSuchElementException`. Para evitar esta excepción se puede llamar en su lugar al método **`orElse`**, el cual devuelve el valor `OptionalDouble` si hay uno, o el valor que pasa a `orElse`, en caso contrario.

La clase `IntStream` también proporciona el método `summaryStatistics` que ejecuta las operaciones `count`, `min`, `max`, `sum` y `average` *en una pasada* de los elementos de un `IntStream` y devuelve los resultados como un objeto `IntSummaryStatistics` (paquete `java.util`). Esto proporciona un incremento considerable en el rendimiento, en comparación con la acción de reprocesar repetidas veces un `IntStream` para cada operación individual. Este objeto tiene métodos para obtener cada resultado y un método `toString` que sintetiza todos los resultados. Por ejemplo, la instrucción:

```
System.out.println(IntStream.of(valores).summaryStatistics());
```

produce:

```
IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000,
max=10}
```

para los valores del arreglo en la figura 17.5.

17.3.3 Operación terminal `reduce`

Puede definir sus propias reducciones para un `IntStream` mediante la invocación de su método **`reduce`** como se muestra en las líneas 29 a 31 de la figura 17.5. Cada una de las operaciones terminales en la sección

17.3.2 es una implementación especializada de `reduce`. Por ejemplo, en la línea 31 se muestra cómo sumar los valores de un `IntStream` mediante el uso de `reduce`, en vez de `sum`. El primer argumento (`0`) es un valor que le ayuda a comenzar la operación de reducción y el segundo argumento es un objeto que implementa la interfaz funcional **`IntBinaryOperator`** (paquete `java.util.function`). La lambda:

```
(x, y) -> x + y
```

implementa el método **`applyAsInt`** de la interfaz, que recibe dos valores `int` (los cuales representan a los operandos izquierdo y derecho de un operador binario) y realiza un cálculo con los valores; en este caso, se suman los valores. Una lambda con dos o más parámetros *debe* encerrarlos entre paréntesis. La evaluación del proceso de reducción es la siguiente:

- En la primera llamada a `reduce`, el valor del parámetro `x` de la lambda es el valor de identidad (`0`) y el valor del parámetro `y` de la lambda es el *primer* `int` en el flujo (`3`), lo que produce la suma `3` (0 + 3).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del primer cálculo (`3`) y el valor del parámetro `y` de la lambda es el *segundo* `int` en el flujo (`10`), lo que produce la suma `13` (3 + 10).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del cálculo anterior (`13`) y el valor del parámetro `y` de la lambda es el *tercer* `int` en el flujo (`6`), lo que produce la suma `19` (13 + 6).

Este proceso continúa produciendo un total actualizado de los valores del `IntStream` hasta que se hayan utilizado todos; en ese punto se devuelve la suma final.

Argumento valor de identidad del método `reduce`

El primer argumento del método `reduce` se conoce formalmente como **valor de identidad**: un valor que cuando se combina con un elemento de flujo mediante el `IntBinaryOperator` produce el valor original de ese elemento. Por ejemplo, al sumar los elementos el valor de identidad es `0` (cualquier valor `int` que se suma a `0` produce como resultado el valor original) y al obtener el producto de los elementos el valor de identidad es 1 (cualquier valor `int` multiplicado por 1 produce como resultado el valor original).

Suma de los cuadrados de los valores con el método `reduce`

En las líneas 34 a 36 de la figura 17.5 se usa el método `reduce` para calcular las sumas de los cuadrados de los valores del `IntStream`. La lambda en este caso suma el *cuadrado* del valor actual al total actualizado. La evaluación de la reducción procede de la siguiente manera:

- En la primera llamada a `reduce`, el valor del parámetro `x` de la lambda es el valor de identidad (`0`) y el valor del parámetro `y` de la lambda es el *primer* `int` en el flujo (`3`), lo que produce el valor `9` ($0 + 3^2$).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del primer cálculo (`9`) y el valor del parámetro `y` de la lambda es el *segundo* `int` en el flujo (`10`), lo que produce la suma `109` ($9 + 10^2$).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del cálculo anterior (`109`) y el valor del parámetro `y` de la lambda es el *tercer* `int` en el flujo (`6`), lo que produce la suma `145` ($109 + 6^2$).

Este proceso continúa produciendo un total actualizado de los cuadrados de los valores del `IntStream` hasta que se hayan usado todos; en este punto se devuelve la suma final.

Cálculo del producto de los valores con el método reduce

En las líneas 39 a 41 de la figura 17.5 se usa el método `reduce` para calcular el producto de los valores del `IntStream`. En este caso, la lambda multiplica sus dos argumentos. Como vamos a producir un producto, comenzamos con el valor de identidad `1` en este caso. La evaluación de la reducción procede de la siguiente forma:

- En la primera llamada a `reduce`, el valor del parámetro `x` de la lambda es el valor de identidad (`1`) y el valor del parámetro `y` de la lambda es el *primer* `int` en el flujo (`3`), lo que produce el valor `3` (1 * 3).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del primer cálculo (`3`) y el valor del parámetro `y` de la lambda es el *segundo* `int` en el flujo (`10`), lo que produce la suma `30` (3 * 10).
- En la siguiente llamada a `reduce`, el valor del parámetro `x` de la lambda es el resultado del cálculo anterior (`30`) y el valor del parámetro `y` de la lambda es el *tercer* `int` en el flujo (`6`), lo que produce la suma `180` (30 * 6).

Este proceso continúa produciendo un producto actualizado de los valores del `IntStream` hasta que se hayan usado todos; en este punto se devuelve el producto final.

17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores IntStream

En las líneas 45 a 48 de la figura 17.5 se crea una canalización de flujo que *localiza* los enteros pares en un `IntStream`, los *ordena* en forma ascendente y *muestra en pantalla* cada valor seguido de un espacio.

Operación intermedia filter

Un programador *filtra* elementos para producir un flujo de resultados inmediatos que coincidan con una condición; a esto se le conoce como *predicado*. El método **`filter`** de `IntStream` (línea 46) recibe un objeto que implementa a la interfaz funcional **`IntPredicate`** (paquete `java.util.function`). La lambda en la línea 46:

```
valor -> valor % 2 == 0
```

implementa el método **`test`** de la interfaz, el cual recibe un `int` y devuelve un `boolean` para indicar si el `int` satisface al predicado; en este caso, el `IntPredicate` devuelve `true` si el valor que recibe puede dividirse entre 2. Las llamadas a `filter` y a otros flujos intermedios son *perezosas*, ya que no se evalúan sino hasta que se realice una *operación terminal* (la cual es *ansiosa*) y producen nuevos flujos de elementos. En las líneas 45 a 48, esto ocurre cuando se llama a `forEach` (línea 48).

Operación intermedia sorted

El método **`sorted`** de `IntStream` ordena en forma *ascendente* los elementos del flujo. Al igual que `filter`, `sorted` es una operación *perezosa*; sin embargo, cuando se llega a realizar el ordenamiento todas las operaciones intermedias anteriores en la canalización de flujo deben completarse, de modo que el método `sorted` sepa qué elementos ordenar.

Procesamiento de la canalización de flujo y comparación entre operaciones intermedias con estado y sin estado

Cuando se llama a `forEach`, se procesa la canalización de flujo. En la línea 46 se produce un `IntStream` intermedio que contiene sólo los enteros pares; luego en la línea 47 se ordenan y en la línea 48 se muestra en pantalla cada elemento.

El método `filter` es una **operación intermedia sin estado**, ya que no requiere información sobre otros elementos en el flujo para probar si el elemento actual satisface al predicado. De manera similar, el método `map` (que veremos en breve) es una operación intermedia sin estado. El método `sorted` es una **operación intermedia con estado** que requiere información sobre *todos* los demás elementos en el flujo para poder ordenarlos. De manera similar, el método `distinct` es una operación intermedia con estado. La documentación en línea para cada operación de flujo intermedia especifica si es una operación con o sin estado.

Otros métodos de la interfaz funcional `IntPredicate`

La interfaz `IntPredicate` también contiene tres métodos `default`:

- **`and`** realiza un *AND lógico* con *evaluación de corto circuito* (sección 5.9) entre el `IntPredicate` sobre el cual se invoca y el `IntPredicate` que recibe como argumento.
- **`negate`** *invierte* el valor `boolean` del `IntPredicate` sobre el cual se invoca.
- **`or`** realiza un *OR lógico* con *evaluación de corto circuito* entre el `IntPredicate` sobre el cual se invoca y el `IntPredicate` que recibe como argumento.

Composición de expresiones lambda

Puede usar estos métodos y objetos `IntPredicate` para elaborar condiciones más complejas. Por ejemplo, considere los siguientes dos `IntPredicate`:

```
IntPredicate par = valor -> valor % 2 == 0;
IntPredicate mayorQue5 = valor -> valor > 5;
```

Para localizar todos los enteros pares mayores que 5, podría sustituir la lambda en la línea 46 con el `IntPredicate`

```
even.and(mayorQue5)
```

17.3.5 Operación intermedia: asignación

En las líneas 54 a 58 de la figura 17.5 se crea una canalización de flujo que *localiza* los enteros impares en un `IntStream`, *multiplica* cada entero impar por 10, *ordena* los valores en forma ascendente y *muestra* cada valor seguido de un espacio.

Operación intermedia `map`

La nueva característica aquí es la operación de *asignación* que recibe cada valor y lo multiplica por 10. La asignación es una *operación intermedia* que transforma los elementos de un flujo en nuevos valores y produce un flujo que contiene los elementos resultantes. Algunas veces son de tipos distintos a los elementos del flujo original.

El método **`map`** (línea 56) recibe un objeto que implementa a la interfaz funcional **`IntUnaryOperator`** (paquete `java.util.function`). La lambda en la línea 55:

```
valor -> valor * 10
```

implementa al método **`applyAsInt`** de la interfaz, el cual recibe un `int` y lo asigna a un nuevo valor `int`. Las llamadas a `map` son *perezosas*. El método `map` es una operación de flujo *sin estado*.

Procesamiento de la canalización de flujo
Cuando se invoca a `forEach` (línea 58), se procesa la canalización de flujo. Primero, la línea 55 produce un `IntStream` intermedio que contiene los valores impares. Luego, en la línea 56 se multiplica cada entero impar por `10`. Después, en la línea 57 se ordenan los valores y en la línea 58 se muestra cada elemento.

17.3.6 Creación de flujos de valores `int` con los métodos `range` y `rangeClosed` de `IntStream`

Si necesita una *secuencia ordenada* de valores `int`, puede crear un `IntStream` que contenga dichos valores con los métodos `range` (línea 63 de la figura 17.5) y `rangeClosed` (línea 67) de `IntStream`. Ambos métodos reciben dos argumentos `int` que representan el rango de valores. El método **`range`** produce una secuencia de valores desde su primer argumento hasta (pero *sin* incluir) su segundo argumento. El método **`rangeClosed`** produce una secuencia de valores, incluyendo sus *dos* argumentos. En las líneas 63 y 67 se demuestran estos métodos para producir secuencias de valores `int` del 1 al 9 y del 1 al 10, respectivamente.

17.4 Manipulaciones de objetos Stream<Integer>

[En esta sección se demuestra cómo pueden usarse las lambdas y los flujos para simplificar las tareas de programación que aprendió en el capítulo 7, arreglos y objetos `ArrayList`].
Así como el método `of` de la clase `IntStream` puede crear un `IntStream` a partir de un arreglo de valores `int`, el método **`stream`** de la clase `Array` puede usarse para crear un objeto `Stream` a partir de un arreglo de objetos. La figura 17.6 realiza el *filtrado* y *ordenamiento* en un `Stream<Integer>`, mediante el uso de las mismas técnicas que aprendió en la sección 17.3. El programa también muestra cómo *recolectar* los resultados de las operaciones de una canalización de flujo en una nueva colección que puede procesar en instrucciones subsiguientes. En este ejemplo usamos el arreglo `Integer valores` (línea 12) que se inicializa con valores `int`; el compilador *encierra* cada `int` en un objeto `Integer`. En la línea 15 se muestra el contenido de `valores` antes de realizar cualquier procesamiento de flujos.

```
1  // Fig. 17.6: ArreglosYFlujos.java
2  // Demostración de lambdas y flujos con un arreglo de enteros.
3  import java.util.Arrays;
4  import java.util.Comparator
5  import java.util.List;
6  import java.util.stream.Collectors;
7
8  public class ArreglosYFlujos
9  {
10    public static void main(String[] args)
11    {
12       Integer[] valores = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
13
14       // muestra los valores originales
15       System.out.printf("Valores originales: %s%n", Arrays.asList(valores));
```

Fig. 17.6 | Demostración de lambdas y flujos con un arreglo de objetos `Integer` (parte 1 de 2).

```
16
17          // ordena los valores en forma ascendente con flujos
18          System.out.printf("Valores ordenados: %s%n",
19             Arrays.stream(valores)
20                .sorted()
21                .collect(Collectors.toList()));
22
23          // valores mayores que 4
24          List<Integer> mayorQue4 =
25            Arrays.stream(valores)
26                .filter(value -> value > 4)
27                .collect(Collectors.toList());
28          System.out.printf("Valores mayores que 4: %s%n", mayorQue4);
29
30          // filtra los valores mayores que 4 y luego ordena los resultados
31          System.out.printf("Valores ordenados mayores que 4: %s%n",
32             Arrays.stream(valores)
33                .filter(value -> value > 4)
34                .sorted()
35                .collect(Collectors.toList()));
36
37          // objeto List mayorQue4 ordenado con flujos
38          System.out.printf(
39             "Valores mayores que 4 (ascendente con flujos): %s%n",
40             mayorQue4.stream()
41                .sorted()
42                .collect(Collectors.toList()));
43       }
44    } // fin de la clase ArreglosYFlujos
```

```
Valores originales: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
Valores ordenados: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Valores mayores que 4: [9, 5, 7, 8, 6]
Valores ordenados mayores que 4: [5, 6, 7, 8, 9]
Valores mayores que 4 (ascendente con flujos): [5, 6, 7, 8, 9]
```

Fig. 17.6 | Demostración de lambdas y flujos con un arreglo de objetos `Integer` (parte 2 de 2).

17.4.1 Creación de un `Stream<Integer>`

Al pasar un arreglo de objetos al método `static stream` de la clase `Arrays`, el método devuelve un `Stream` del tipo apropiado; por ejemplo, en la línea 19 se produce un `Stream<Integer>` a partir de un arreglo `Integer`. La interfaz **`Stream`** (paquete `java.util.stream`) es una interfaz genérica para realizar operaciones de flujos sobre cualquier tipo *no primitivo*. Los tipos de objetos que se procesan son determinados por el origen del `Stream`.

La clase `Arrays` también proporciona versiones sobrecargadas del método `stream` para crear objetos `IntStream`, `LongStream` y `DoubleStream` a partir de arreglos `int`, `long` y `double` completos o de rangos de elementos en los arreglos. Las clases `IntStream`, `LongStream` y `DoubleStream` especializadas proporcionan varios métodos para operaciones comunes sobre flujos numéricos, como vio en la sección 17.3.

17.4.2 Ordenamiento de un objeto Stream y recolección de los resultados

En la sección 7.5 aprendió a ordenar arreglos con los métodos `static sort` y `parallelSort` de la clase `Arrays`. A menudo tendrá que ordenar los resultados de las operaciones de flujos, por lo que en las líneas 18 a 21 ordenaremos el arreglo `valores` usando las técnicas de flujos y mostraremos en pantalla los valores ordenados. Primero, en la línea 19 se crea un `Stream<Integer>` a partir de `valores`. Después, en la línea 20 se hace una llamada al método `sorted` de `Stream` que ordena los elementos; esto produce un `Stream<Integers>` intermedio con los valores en orden *ascendente*.

Para mostrar en pantalla los resultados ordenados, podríamos producir cada valor usando la operación terminal `forEach` de `Stream` (como en la línea 15 de la figura 17.5). Sin embargo, al procesar flujos a menudo se crean *nuevas* colecciones que contienen los resultados, de modo que pueda realizar operaciones adicionales sobre ellos. Para crear una colección puede usar el método **`collect`** de `Stream` (figura 17.6, línea 21), que es una *operación terminal*. A medida que se procesa la canalización de flujo, el método `collect` realiza una operación de **reducción mutable** que coloca los resultados en un objeto que *puede modificarse* de manera subsiguiente; a menudo una colección, como un objeto `List`, `Map` o `Set`. La versión del método `collect` en la línea 21 recibe como argumento un objeto que implementa a la interfaz **`Collector`** (paquete `java.util.stream`), el cual especifica cómo realizar la reducción mutable. La clase **`Collectors`** (paquete `java.util.stream`) proporciona métodos `static` que devuelven implementaciones de `Collector` predefinidas. Por ejemplo, el método **`toList`** de `Collectors` (línea 21) transforma el `Stream<Integer>` en una colección `List<Integer>`. En las líneas 18 a 21, el `List<Integer>` resultante se muestra en seguida con una llamada *implícita* a su método `toString`.

En la sección 17.6 demostramos otra versión del método `collect`.

17.4.3 Filtrado de un Stream y almacenamiento de los resultados para su uso posterior

En las líneas 24 a 27 de la figura 17.6 se crea un `Stream<Integer>`, se hace una llamada al método `filter` de `Stream` (que recibe un `Predicate`) para localizar todos los valores mayores que 4 y se recolectan (`collect`) los resultados en un objeto `List<Integer>`. Al igual que `IntPredicate` (sección 17.3.4), la interfaz funcional `Predicate` tiene un método `test` que devuelve un `boolean` para indicar si el argumento satisface una condición, además de los métodos **`and`**, **`negate`** y **`or`**.

Asignamos el objeto `List<Integer>` resultante de la canalización de flujo a la variable `mayorQue4`, que se usa en la línea 28 para mostrar los valores mayores que 4 y se usa de nuevo en las líneas 40 a 42, para realizar operaciones adicionales sólo en los valores mayores que 4.

17.4.4 Filtrado y ordenamiento de un objeto Stream y recolección de los resultados

En las líneas 31 a 35 se muestran los valores mayores que 4 en orden. Primero, en la línea 32 se crea un `Stream<Integer>`. Después, en la línea 33 se filtran (`filter`) los elementos para localizar todos los valores mayores que 4. Luego, en la línea 34 indicamos que queremos los resultados ordenados (`sorted`). Por último, en la línea 35 se recolectan (`collect`) los resultados en un objeto `List<Integer>`, que después se muestra como `String`.

17.4.5 Ordenamiento de los resultados recolectados previamente

En las líneas 40 a 42 se usa la colección `mayorQue4` que se creó en las líneas 24 a 27 para mostrar el procesamiento adicional en una colección que contiene los resultados de una canalización de flujo anterior.

En este caso usamos flujos para ordenar los valores en `mayorQue4`, recolectamos (`collect`) los resultados en un nuevo objeto `List<Integers>` y mostramos los valores ordenados.

17.5 Manipulaciones de objetos `Stream<String>`

[En esta sección se demuestra cómo pueden usarse las lambdas y los flujos para simplificar las tareas de programación que aprendió en el capítulo 14, Cadenas, caracteres y expresiones regulares].
En la figura 17.7 se realizan algunas de las mismas operaciones con flujos que vimos en las secciones 17.3 y 17.4, sólo que con un `Stream<String>`. Además, demostramos el *ordenamiento sin sensibilidad al uso de mayúsculas y minúsculas*, así como el ordenamiento en forma *descendente*. En este ejemplo usamos el arreglo `String` llamado `cadenas` (líneas 11 y 12) que se inicializa con los nombres de los colores; algunos con una letra mayúscula inicial. En la línea 15 se muestra el contenido de `cadenas` *antes* de realizar cualquier procesamiento de flujos.

```
1  // Fig. 17.7: ArreglosYFlujos2.java
2  // Demostración de las lambdas y los flujos con un arreglo de objetos String.
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.stream.Collectors;
6
7  public class ArreglosYFlujos2
8  {
9     public static void main(String[] args)
10    {
11       String[] cadenas =
12          {"Rojo", "naranja", "Amarillo", "verde", "Azul", "indigo", "Violeta"};
13
14       // muestra las cadenas originales
15       System.out.printf("Cadenas originales: %s%n", Arrays.asList(cadenas));
16
17       // cadenas en mayúscula
18       System.out.printf("cadenas en mayuscula: %s%n",
19          Arrays.stream(cadenas)
20                .map(String::toUpperCase)
21                .collect(Collectors.toList()));
22
23       // cadenas mayores que "m" (sin susceptibilidad al uso de
         mayúsculas/minúsculas) en orden ascendente
24       System.out.printf("cadenas mayores que m en orden ascendente: %s%n",
25          Arrays.stream(cadenas)
26                .filter(s -> s.compareToIgnoreCase("m") > 0)
27                .sorted(String.CASE_INSENSITIVE_ORDER)
28                .collect(Collectors.toList()));
29
30       // cadenas mayores que "m" (sin susceptibilidad al uso de
         mayúsculas/minúsculas) en orden descendente
31       System.out.printf("cadenas mayores que m en orden descendente: %s%n",
32          Arrays.stream(cadenas)
33                .filter(s -> s.compareToIgnoreCase("m") > 0)
34                .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
35                .collect(Collectors.toList()));
36    }
37 } // fin de la clase ArreglosYFlujos2
```

Fig. 17.7 | Demostración de lambdas y flujos con un arreglo de objetos `String` (parte 1 de 2).

```
Cadenas originales: [Rojo, naranja, Amarillo, verde, Azul, indigo, Violeta]
cadenas en mayuscula: [ROJO, NARANJA, AMARILLO, VERDE, AZUL, INDIGO, VIOLETA]
cadenas mayores que m en orden ascendente: [naranja, Rojo, verde, Violeta]
cadenas mayores que m en orden descendente: [Violeta, verde, Rojo, naranja]
```

Fig. 17.7 | Demostración de lambdas y flujos con un arreglo de objetos `String` (parte 2 de 2).

17.5.1 Asociación de objetos `String` a mayúsculas mediante la referencia a un método

En las líneas 18 a 21 se muestran los objetos `String` en mayúsculas. Para ello, en la línea 19 se crea un `Stream<String>` a partir del arreglo `cadenas` y luego en la línea 20 se hace una llamada al método `map` de `Stream` para asignar cada `String` a su versión en mayúsculas mediante una llamada al método de instancia `toUpperCase` de `String`. `String::toUpperCase` se conoce como una **referencia a método** y es una notación abreviada para una expresión lambda; en este caso, para una expresión lambda como:

```
(String s) -> {return s.toUpperCase();}
```

o

```
s -> s.toUpperCase()
```

`String::toUpperCase` es una referencia a método para el método de instancia `toUpperCase` de `String`. En la figura 17.8 se muestran los cuatro tipos de referencias a métodos.

Lambda	Descripción
`String::toUpperCase`	Referencia a método para un método de instancia de una clase. Crea una lambda de un parámetro que invoca al método de instancia con el argumento de la lambda y devuelve el resultado del método. Se utiliza en la figura 17.7.
`System.out::println`	Referencia a método para un método de instancia que debe invocarse sobre un objeto específico. Crea una lambda de un parámetro que invoca al método de instancia sobre el objeto especificado (pasa el argumento de la lambda al método de instancia) y devuelve el resultado del método. Se usa en la figura 17.10.
`Math::sqrt`	Referencia a método para un método `static` de una clase. Crea una lambda de un parámetro en donde el argumento de la lambda se pasa al método `static` especificado y la lambda devuelve el resultado del método.
`TreeMap::new`	Referencia a un constructor. Crea una lambda que invoca el constructor sin argumentos de la clase especificada para crear e inicializar un nuevo objeto de esa clase. Se usa en la figura 17.17.

Fig. 17.8 | Tipos de referencias a métodos.

El método `map` de `Stream` recibe como argumento un objeto que implementa a la interfaz funcional `Function`; es decir, la referencia al método de instancia `String::toUpperCase` se trata como una lambda que implementa a la interfaz `Function`. El método `apply` de esta interfaz recibe un parámetro y devuelve un resultado; en este caso, el método `apply` recibe un `String` y devuelve la versión en mayúsculas del objeto `String`. En la línea 21 se *recolectan* los resultados en un objeto `List<String>` que mostramos en pantalla como objeto `String`.

17.5.2 Filtrado de objetos `String` y ordenamiento ascendente sin distinguir entre mayúsculas y minúsculas

En las líneas 24 a 28 se filtran y ordenan los objetos `String`. En la línea 25 se crea un `Stream<String>` a partir del arreglo `cadenas`, luego en la línea 26 se hace una llamada al método `filter` para localizar todos los objetos `String` que sean mayores que "`m`", utilizando una comparación *sin susceptibilidad al uso de mayúsculas y minúsculas* en la lambda `Predicate`. En la línea 27 se ordenan los resultados y en la línea 28 se recolectan en un objeto `List<String>` que mostramos en pantalla como objeto `String`. En este caso, en la línea 27 se invoca la versión del método `sorted` de `stream` que recibe un `Comparator` como argumento. Como vimos en la sección 16.7.1, un `Comparator` define a un método `compare` que devuelve un valor negativo si el primer valor que se va a comparar es mayor que el segundo. De manera predeterminada, el método `sorted` usa el *orden natural* para el tipo; para los objetos `String`, el orden natural es susceptible al uso de mayúsculas y minúsculas, lo que significa que "`Z`" es menor que "`a`". Al pasar el objeto `Comparator String.CASE_INSENSITIVE_ORDER` se realiza un ordenamiento *sin susceptibilidad al uso de minúsculas y mayúsculas*.

17.5.3 Filtrado de objetos `String` y ordenamiento descendente sin distinguir entre mayúsculas y minúsculas

En las líneas 31 a 35 se realizan las mismas tareas que en las líneas 24 a 28, pero se ordenan los objetos `String` en forma *descendente*. La interfaz funcional `Comparator` contiene el método `default` **`reversed`**, que invierte el orden de un `Comparator` existente. Cuando se aplica a `String.CASE_INSENSITIVE_ORDER`, los objetos `String` se ordenan en forma *descendente*.

17.6 Manipulaciones de objetos `Stream<Empleado>`

El ejemplo en las figuras 17.9 a 17.16 demuestra varias herramientas de lambdas y flujos que usan un `Stream<Employee>`. La clase `Empleado` (figura 17.9) representa a un empleado con un nombre de pila, apellido, salario y departamento; además proporciona métodos para manipular estos valores. Así mismo, la clase proporciona un método `obtenerNombre` (líneas 69 a 72) que devuelve la combinación de nombre y apellido como un objeto `String`, y un método `toString` (líneas 75 a 80) que devuelve un objeto `String` con formato que contiene el nombre, apellido, salario y departamento del empleado.

```
1  // Fig. 17.9: Empleado.java
2  // Clase Empleado.
3  public class Empleado
4  {
5     private String primerNombre;
6     private String apellidoPaterno;
7     private double salario;
8     private String departamento;
9
10    // constructor
11    public Empleado(String primerNombre, String apellidoPaterno,
12       double salario, String departamento)
13    {
14       this.primerNombre = primerNombre;
15       this.apellidoPaterno = apellidoPaterno;
```

Fig. 17.9 | Clase `Empleado` que se usará en las figuras 17.10 a 17.16 (parte 1 de 3).

```
16          this.salario = salario;
17          this.departamento = departamento;
18       }
19
20       // establece primerNombre
21       public void establecerPrimerNombre(String primerNombre)
22       {
23          this.primerNombre = primerNombre;
24       }
25
26       // obtiene primerNombre
27       public String obtenerPrimerNombre()
28       {
29          return primerNombre;
30       }
31
32       // establece apellidoPaterno
33       public void establecerApellidoPaterno(String apellidoPaterno)
34       {
35          this.apellidoPaterno = apellidoPaterno;
36       }
37
38       // obtiene apellidoPaterno
39       public String obtenerApellidoPaterno()
40       {
41          return apellidoPaterno;
42       }
43
44       // establece salario
45       public void establecerSalario(double salario)
46       {
47          this.salario = salario;
48       }
49
50       // obtiene salario
51       public double obtenerSalario()
52       {
53          return salario;
54       }
55
56       // establece departamento
57       public void establecerDepartamento(String departamento)
58       {
59          this.departamento = departamento;
60       }
61
62       // obtiene departamento
63       public String obtenerDepartamento()
64       {
65          return departamento;
66       }
67
```

Fig. 17.9 | Clase `Empleado` que se usará en las figuras 17.10 a 17.16 (parte 2 de 3).

```
68      // devuelve primer nombre y apellido combinados del Empleado
69      public String obtenerNombre()
70      {
71       return String.format("%s %s", obtenerPrimerNombre(), obtenerApellidoPaterno());
72      }
73
74      // devuelve un objeto String que contiene la información del Empleado
75      @Override
76      public String toString()
77      {
78         return String.format("%-8s %-8s %8.2f   %s",
79            obtenerPrimerNombre(), obtenerApellidoPaterno(), obtenerSalario(),
               obtenerDepartamento());
80      } // fin del método toString
81   } // fin de la clase Empleado
```

Fig. 17.9 | Clase `Empleado` que se usará en las figuras 17.10 a 17.16 (parte 3 de 3).

17.6.1 Creación e impresión en pantalla de un objeto `List<Empleado>`

La clase `ProcesarEmpleados` (figuras 17.10 a 17.16) se divide en varias figuras para poder mostrarle las operaciones de lambdas y flujos con sus correspondientes resultados. En la figura 17.10 se crea un arreglo de objetos `Empleado` (líneas 17 a 24) y se obtiene su vista `List` (línea 27).

```
1   // Fig. 17.10: ProcesarEmpleados.java
2   // Procesamiento de flujos de objetos Empleado.
3   import java.util.Arrays;
4   import java.util.Comparator;
5   import java.util.List;
6   import java.util.Map;
7   import java.util.TreeMap;
8   import java.util.function.Function;
9   import java.util.function.Predicate;
10  import java.util.stream.Collectors;
11
12  public class ProcesarEmpleados
13  {
14     public static void main(String[] args)
15     {
16        // inicializa arreglo de objetos Empleado
17        Empleado[] empleados = {
18           new Empleado("Jason", "Red", 5000, "TI"),
19           new Empleado("Ashley", "Green", 7600, "TI"),
20           new Empleado("Matthew", "Indigo", 3587.5, "Ventas"),
21           new Empleado("James", "Indigo", 4700.77, "Marketing"),
22           new Empleado("Luke", "Indigo", 6200, "TI"),
23           new Empleado("Jason", "Blue", 3200, "Ventas"),
24           new Empleado("Wendy", "Brown", 4236.4, "Marketing")};
25
26        // obtiene vista List de los objetos Empleado
27        List<Empleado> lista = Arrays.asList(empleados);
```

Fig. 17.10 | Crear un arreglo de objetos `Empleado`, convertirlo en un objeto `List` y mostrarlo en pantalla (parte 1 de 2).

```
28
29          // muestra todos los objetos Empleado
30          System.out.println("Lista completa de empleados:");
31          lista.stream().forEach(System.out::println);
32
```

```
Lista completa de empleados:
Jason    Red       5000.00   TI
Ashley   Green     7600.00   TI
Matthew  Indigo    3587.50   Ventas
James    Indigo    4700.77   Marketing
Luke     Indigo    6200.00   TI
Jason    Blue      3200.00   Ventas
Wendy    Brown     4236.40   Marketing
```

Fig. 17.10 | Crear un arreglo de objetos `Empleado`, convertirlo en un objeto `List` y mostrarlo en pantalla (parte 2 de 2).

En la línea 31 se crea un objeto `Stream<Empleado>` y luego se usa el método `forEach` de `String` para mostrar la representación `String` de cada `Empleado`. El compilador convierte la referencia al método de instancia `System.out::println` en un objeto que implementa a la interfaz funcional `Consumer`. El método **`accept`** de esta interfaz recibe un argumento y devuelve `void`. En este ejemplo, el método `accept` pasa cada `Empleado` al método de instancia `println` del objeto `System.out`, que invoca de manera implícita al método `toString` de `Empleado` para obtener la representación `String`. La salida al final de la figura 17.10 muestra los resultados de mostrar todos los objetos `Empleado`.

17.6.2 Filtrado de objetos `Empleado` con salarios en un rango especificado

En la figura 17.11 se demuestra cómo filtrar objetos `Empleado` con un objeto que implementa la interfaz funcional `Predicate<Empleado>`, que se define con una lambda en las líneas 34 y 35. Al definir las lambdas de esta manera puede reutilizarlas varias veces, como en las líneas 42 y 49. En las líneas 41 a 44 se muestran en pantalla los objetos `Empleado` con los salarios en el rango 4000 a 6000, ordenados por salario de la siguiente manera:

- En la línea 41 se crea un `Stream<Empleado>` a partir del objeto `List<Empleado>`.
- En la línea 42 se filtra el flujo usando el `Predicate` llamado `cuatroASeisMil`.
- En la línea 43 se ordenan por salario los objetos `Empleado` que quedan en el flujo. Para especificar un `Comparator` para los salarios, usamos el método `static comparing` de la interfaz `Comparator`. El compilador convierte la referencia al método `Empleado::obtenerSalario` que se pasa como argumento en un objeto que implementa a la interfaz `Function`. Esta interfaz se usa para extraer un valor de un objeto en el flujo para usarlo en las comparaciones. El método `comparing` devuelve un objeto `Comparator` que invoca a `obtenerSalario` en cada uno de dos objetos `Empleado`, después devuelve un valor negativo si el salario del primer `Empleado` es menor que el del segundo, 0 si son iguales y un valor positivo si el salario del primer `Empleado` es mayor que el del segundo.
- Por último, en la línea 44 se realiza la operación terminal `forEach` que procesa la canalización de flujo y muestra en pantalla los objetos `Empleado` ordenados por salario.

```
33        // Predicado que devuelve true para salarios en el rango $4000-$6000
34        Predicate<Empleado> cuatroASeisMil =
35           e -> (e.obtenerSalario() >= 4000 && e.obtenerSalario() <= 6000);
36
37        // Muestra los empleados con salarios en el rango $4000-$6000
38        // en orden ascendente por salario
39        System.out.printf(
40           "%nEmpleados que ganan $4000-$6000 mensuales ordenados por salario:%n");
41        lista.stream()
42           .filter(cuatroASeisMil)
43           .sorted(Comparator.comparing(Empleado::obtenerSalario))
44           .forEach(System.out::println);
45
46        // Muestra el primer empleado con salario en el rango $4000-$6000
47        System.out.printf("%nPrimer empleado que gana $4000-$6000:%n%s%n",
48           lista.stream()
49              .filter(cuatroASeisMil)
50              .findFirst()
51              .get());
52
```

```
Empleados que ganan $4000-$6000 mensuales ordenados por salario:
Wendy    Brown     4236.40   Marketing
James    Indigo    4700.77   Marketing
Jason    Red       5000.00   TI

Primer empleado que gana $4000-$6000:
Jason    Red       5000.00   TI
```

Fig. 17.11 | Filtrar objetos `Empleado` con salarios en el rango de $4,000 a $6,000.

Procesamiento de canalización de flujo de cortocircuito

En la sección 5.9 estudió la evaluación de cortocircuito con los operadores AND lógico (`&&`) y OR lógico (`||`). Una de las buenas características de rendimiento de la evaluación perezosa es la habilidad de realizar la *evaluación de cortocircuito*; es decir, detener el procesamiento de la canalización de flujo tan pronto como esté disponible el resultado deseado. En la línea 50 se demuestra el método **`findFirst`** de `Stream`: una *operación terminal de cortocircuito* que procesa la canalización de flujo y termina el procesamiento tan pronto como se encuentra el *primer* objeto de la canalización de flujo. Con base en la lista original de objetos `Empleado`, el procesamiento del flujo en las líneas 48 a 51 (que filtra los objetos `Empleado` con salarios en el rango de $4,000 a $6,000) se lleva a cabo así: el `Predicate cuatroASeisMil` se aplica al primer `Empleado` (Jason Red). Su salario ($5,000.00) está en el rango de $4,000 a $6,000, por lo que el `Predicate` devuelve `true` y el procesamiento del flujo termina *de inmediato*, habiendo procesado sólo uno de los ocho objetos en el flujo. Después el método `findFirst` devuelve un objeto `Optional` (en este caso un `Optional<Empleado>`) que contiene el objeto que se encontró, en caso de haberlo. La llamada al método `get` de `Optional` (línea 51) devuelve el objeto `Empleado` que coincide en este ejemplo. Incluso aunque el flujo contuviera millones de objetos `Empleado`, la operación `filter` se realizaría sólo hasta encontrar una coincidencia.

17.6.3 Ordenamiento de objetos `Empleado` según varios campos

En la figura 17.12 se muestra cómo ordenar objetos según *varios* campos. En este ejemplo, ordenamos los objetos `Empleado` por apellido y luego, en el caso de los objetos `Empleado` con el mismo apellido, también

los ordenamos por nombre. Para ello comenzamos a crear dos objetos `Function`, cada uno de los cuales recibe un `Empleado` y devuelve un objeto `String`:

- A `porPrimerNombre` (línea 54) se le asigna la referencia a un método para el método de instancia `obtenerPrimerNombre` de `Empleado`
- A `porApellidoPaterno` (línea 55) se le asigna la referencia a un método para el método de instancia `obtenerApellidoPaterno` de `Empleado`

A continuación usamos estos objetos `Function` para crear un `Comparator` (`apellidoLuegoNombre`; líneas 58 a 59) que compara primero dos objetos `Empleado` por apellido y luego por nombre. Usamos el método `comparing` de `Comparator` para crear un `Comparator` que invoca a `Function porApellidoPaterno` en un `Empleado` para obtener su apellido paterno. En el `Comparator` resultante, invocamos al método **`thenComparing`** de `Comparator` para crear un objeto `Comparator` que primero compara a los objetos `Empleado` por apellido y, *si los apellidos son iguales*, los compara por nombre. En las líneas 64 y 65 se usa este nuevo `Comparator` llamado `apellidoLuegoNombre` para ordenar a los objetos `Empleado` en orden *ascendente* y luego mostrar los resultados. Reutilizamos el `Comparator` en las líneas 71 a 73, pero invocamos a su método `reversed` para indicar que los objetos `Empleado` deben ordenarse en forma *descendente* por apellido y luego por nombre.

```
53        // Funciones para obtener primer nombre y apellido de un Empleado
54        Function<Empleado, String> porPrimerNombre = Empleado::obtenerPrimerNombre;
55        Function<Empleado, String> porApellidoPaterno = Empleado::obtenerApellidoPaterno;
56
57        // Comparator para comparar empleados por primer nombre y luego por
          apellido paterno
58        Comparator<Empleado> apellidoLuegoNombre =
59           Comparator.comparing(porApellidoPaterno).thenComparing(porPrimerNombre);
60
61        // ordena empleados por apellido paterno y luego por primer nombre
62        System.out.printf(
63           "%nEmpleados en orden ascendente por apellido y luego por nombre:%n");
64        lista.stream()
65           .sorted(apellidoLuegoNombre)
66           .forEach(System.out::println);
67
68        // ordena empleados en forma descendente por apellido, luego por nombre
69        System.out.printf(
70           "%nEmpleados en orden descendente por apellido y luego por nombre:%n");
71        lista.stream()
72           .sorted(apellidoLuegoNombre.reversed())
73           .forEach(System.out::println);
74
```

```
Empleados en orden ascendente por apellido y luego por nombre:
Jason    Blue      3200.00   Ventas
Wendy    Brown     4236.40   Marketing
Ashley   Green     7600.00   TI
James    Indigo    4700.77   Marketing
Luke     Indigo    6200.00   TI
Matthew  Indigo    3587.50   Ventas
Jason    Red       5000.00   TI
```

Fig. 17.12 | Ordenar objetos `Empleado` por apellido y luego por nombre (parte 1 de 2).

```
Empleados en orden descendente por apellido y luego por nombre:
Jason    Red       5000.00   TI
Matthew  Indigo    3587.50   Ventas
Luke     Indigo    6200.00   TI
James    Indigo    4700.77   Marketing
Ashley   Green     7600.00   TI
Wendy    Brown     4236.40   Marketing
Jason    Blue      3200.00   Ventas
```

Fig. 17.12 | Ordenar objetos `Empleado` por apellido y luego por nombre (parte 2 de 2).

17.6.4 Asociación de objetos `Empleado` a objetos `String` con apellidos únicos

Anteriormente utilizó operaciones `map` para realizar cálculos con valores `int` y convertir objetos `String` a letras mayúsculas. En ambos casos, los flujos resultantes contenían valores de los mismos tipos que los flujos originales. En la figura 17.13 se muestra cómo asignar objetos de un tipo (`Empleado`) a objetos de un tipo diferente (`String`). En las líneas 77 a 81 se realizan las siguientes tareas:

- En la línea 77 se crea un `Stream<Empleado>`.
- En la línea 78 se asignan los objetos `Empleado` a sus apellidos mediante la referencia al método de instancia `empleado::obtenerNombre` como el argumento `Function` del método `map`. El resultado es un `Stream<String>`.
- En la línea 79 se invoca el método **`distinct`** de `Stream` en el `Stream<String>` para eliminar los objetos `String` duplicados en un `Stream<String>`.
- En la línea 80 se ordenan los apellidos únicos.
- Por último, en la línea 81 se realiza una operación terminal `forEach` que procesa la canalización de flujo y muestra en pantalla los apellidos únicos en orden.

En las líneas 86 a 89 se ordenan los objetos `Empleado` por apellido y luego por nombre; después se asignan (`map`) los objetos `Empleado` a los objetos `String` con el método de instancia `obtenerNombre` de `Empleado` (línea 88) y se muestran los nombres ordenados en una operación terminal `forEach`.

```
75        // muestra apellidos de empleados únicos ordenados
76        System.out.printf("%nApellidos de empleados unicos:%n");
77        lista.stream()
78           .map(Empleado::obtenerApellidoPaterno)
79           .distinct()
80           .sorted()
81           .forEach(System.out::println);
82
83        // muestra sólo nombre y apellido
84        System.out.printf(
85           "%nNombres de empleados en orden por apellido y luego por nombre:%n");
86        lista.stream()
87           .sorted(apellidoLuegoNombre)
88           .map(Empleado::obtenerNombre)
89           .forEach(System.out::println);
90
```

Fig. 17.13 | Asignar objetos `Empleado` a apellidos y nombres completos (parte 1 de 2).

```
Apellidos de empleados unicos:
Blue
Brown
Green
Indigo
Red

Nombres de empleados en orden por apellido y luego por nombre:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red
```

Fig. 17.13 | Asignar objetos `Empleado` a apellidos y nombres completos (parte 2 de 2).

17.6.5 Agrupación de objetos `Empleado` por departamento

En la figura 17.14 se usa el método `collect` de `Stream` (línea 95) para agrupar los objetos `Empleado` por departamento. El argumento del método `collect` es un `Collector` que especifica cómo resumir los datos en un formato útil. En este caso usamos el `Collector` devuelto por el método `static` **`groupingBy`** de `Collectors`, el cual recibe un objeto `Function` que clasifica los objetos en el flujo; los valores devueltos por esta función se usan como las claves en un objeto `Map`. Los valores correspondientes de manera predeterminada son objetos `List` que contienen los elementos del flujo en una categoría dada. Cuando se usa el método `collect` con este `Collector`, el resultado es un `Map<String, List<Empleado>>` en donde cada clave `String` es un departamento y cada `List<Empleado>` contiene los objetos `Empleado` en ese departamento. Asignamos este objeto `Map` a la variable `agrupadoPorDepartamento` (que se usa en las líneas 96 a 103) para mostrar a los objetos `Empleado` agrupados por departamento. El método **`forEach`** de `Map` realiza una operación sobre cada uno de los pares clave-valor del objeto `Map`. El argumento del método es un objeto que implementa a la interfaz funcional **`BiConsumer`**. El método `accept` de esta interfaz tiene dos parámetros. Para los objetos `Map`, el primer parámetro representa la clave y el segundo representa el valor correspondiente.

```
91      // agrupa empleados por departamento
92      System.out.printf("%nEmpleados por departamento:%n");
93      Map<String, List<Empleado>> agrupadoPorDepartamento =
94        lista.stream()
95           .collect(Collectors.groupingBy(Empleado::obtenerDepartamento));
96      agrupadoPorDepartamento.forEach(
97         (departamento, empleadosEnDepartamento) ->
98         {
99           System.out.println(departamento);
100          empleadosEnDepartamento.forEach(
101             empleado -> System.out.printf("   %s%n", empleado));
102        }
103     );
104
```

Fig. 17.14 | Agrupación de objetos `Empleado` por departamento (parte 1 de 2).

```
Empleados por departamento:
Ventas
    Matthew  Indigo    3587.50   Ventas
    Jason    Blue      3200.00   Ventas
TI
    Jason    Red       5000.00   TI
    Ashley   Green     7600.00   TI
    Luke     Indigo    6200.00   TI
Marketing
    James    Indigo    4700.77   Marketing
    Wendy    Brown     4236.40   Marketing
```

Fig. 17.14 | Agrupación de objetos `Empleado` por departamento (parte 2 de 2).

17.6.6 Conteo del número de objetos `Empleado` en cada departamento

En la figura 17.15 se demuestra una vez más el método `collect` de `Stream` y el método `static groupingBy` de `Collectors`, pero en este caso contamos el número de objetos `Empleado` en cada departamento. En las líneas 107 a 110 se produce un `Map<String, Long>` en donde cada clave `String` es el nombre de un departamento y el valor `Long` correspondiente es el número de objetos `Empleado` en ese departamento. En este caso usamos una versión del método `static groupingBy` de `Collectors` que recibe dos argumentos: el primero es un objeto `Function` que clasifica los objetos en el flujo y el segundo es otro `Collector` (conocido como el **`Collector` de flujo descendente**). En este caso, usamos una llamada al método `static counting` de `Collectors` como el segundo argumento. Este método devuelve un `Collector` que cuenta el número de objetos en una clasificación dada, en vez de recolectarlos en un objeto `List`. En las líneas 111 a 113 se muestran en pantalla los pares clave-valor del objeto resultante `Map<String, Long>`.

```
105       // cuenta el número de empleados en cada departamento
106       System.out.printf("%nConteo de empleados por departamento:%n");
107       Map<String, Long> conteoEmpleadosPorDepartamento =
108          lista.stream()
109             .collect(Collectors.groupingBy(Empleado::obtenerDepartamento,
110                TreeMap::new, Collectors.counting()));
111       conteoEmpleadosPorDepartamento.forEach(
112          (departamento, conteo) -> System.out.printf(
113             "%s tiene %d empleado(s)%n", departamento, conteo));
114
```

```
Conteo de empleados por departamento:
Marketing tiene 2 empleado(s)
TI tiene 3 empleado(s)
Ventas tiene 2 empleado(s)
```

Fig. 17.15 | Contar el número de objetos `Empleado` en cada departamento.

17.6.7 Suma y promedio de salarios de objetos `Empleado`

En la figura 17.16 se demuestra el método **`mapToDouble`** de `Stream` (líneas 119, 126 y 132), que asigna objetos a valores `double` y devuelve un `DoubleStream`. En este caso asignamos objetos `Empleado` a sus

salarios, para poder calcular la *suma* y el *promedio*. El método `mapToDouble` recibe un objeto que implementa a la interfaz funcional **`ToDoubleFunction`** (paquete `java.util.function`). El método **`applyAsDouble`** de esta interfaz invoca a un método de instancia sobre un objeto y devuelve un valor `double`. En cada una de las líneas 119, 126 y 132 se pasa a `mapToDouble` la referencia al método de instancia `Empleado::obtenerSalario` de `Empleado`, que devuelve el salario del `Empleado` actual como un valor `double`. El compilador convierte esta referencia a método en un objeto que implementa a la interfaz funcional `ToDoubleFunction`.

```
115         // suma de salarios de empleados con el método sum de DoubleStream
116         System.out.printf(
117           "%nSuma de los salarios de los empleados (mediante el metodo sum): %.2f%n",
118             lista.stream()
119                 .mapToDouble(Empleado::obtenerSalario)
120                 .sum());
121
122         // calcula la suma de los salarios de los empleados con el método reduce
            de Stream
123         System.out.printf(
124           "Suma de los salarios de los empleados (mediante el metodo reduce): %.2f%n",
125             lista.stream()
126                 .mapToDouble(Empleado::obtenerSalario)
127                 .reduce(0, (valor1, valor2) -> valor1 + valor2));
128
129         // promedio de salarios de empleados con el método average de DoubleStream
130         System.out.printf("Promedio de salarios de los empleados: %.2f%n",
131             lista.stream()
132                 .mapToDouble(Empleado::obtenerSalario)
133                 .average()
134                 .getAsDouble());
135     } // fin de main
136 } // fin de la clase Procesar empleados
```

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34525.67
Average of Employees' salaries: 4932.10
```

Fig. 17.16 | Contar el número de objetos `Empleado` en cada departamento.

En las líneas 118 a 120 se crea un `Stream<Empleado>`, se asigna a un `DoubleStream` y luego se invoca al método `sum` de `DoubleStream` para calcular la suma de los salarios de los objetos `Empleado`. En las líneas 125 a 127 también se suman los salarios de los objetos `Empleado`, pero se utiliza el método `reduce` de `DoubleStream` en vez de `sum`; en la sección 17.3 presentamos el método `reduce` con flujos `IntStream`. Por último, en las líneas 131 a 134 se calcula el promedio de los salarios de los objetos `Empleado` mediante el método `average` de `DoubleStream`, que devuelve un `OptionalDouble` en caso de que el `DoubleStream` no contenga elementos. En este caso sabemos que el flujo tiene elementos, por lo que simplemente llamamos al método `getAsDouble` de `OptionalDouble` para obtener el resultado. Recuerde que también puede usar el método `orElse` para especificar un valor que debe usarse si el método `average` se invocó con un `DoubleStream` vacío, y por ende no se pudo calcular el promedio.

17.7 Creación de un objeto Stream<String> a partir de un archivo

En la figura 17.17 se usan lambdas y flujos para resumir el número de ocurrencias de cada palabra en un archivo y luego mostrar un resumen de las palabras en orden alfabético, agrupadas por letra inicial. A esto se le conoce comúnmente como una concordancia. Las concordancias se usan con frecuencia para analizar obras publicadas. Por ejemplo, las concordancias de las obras de William Shakespeare y Christopher Marlowe se han usado para cuestionar si son la misma persona. En la figura 17.18 se muestran los resultados del programa. En la línea 16 de la figura 17.17 se crea un objeto `Pattern` de expresión regular que usaremos para dividir las líneas de texto en sus palabras individuales. Este `Pattern` representa uno o más caracteres consecutivos de espacio en blanco (en la sección 14.7 presentamos las expresiones regulares).

```
1  // Fig. 17.17: FlujoDeLineas.java
2  // Contar las ocurrencias de palabras en un archivo de texto.
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.util.Map;
7  import java.util.TreeMap;
8  import java.util.regex.Pattern;
9  import java.util.stream.Collectors;
10
11 public class FlujoDeLineas
12 {
13    public static void main(String[] args) throws IOException
14    {
15       // Expresión regular que coincide con uno o más caracteres consecutivos de
            espacio en blanco
16       Pattern patron = Pattern.compile("\\s+");
17
18       // cuenta las ocurrencias de cada palabra en un Stream<String> ordenado
            por palabra
19       Map<String, Long> cuentasPalabras =
20          Files.lines(Paths.get("ParrafoCapitulo2.txt"))
21             .map(line -> line.replaceAll("(?!')\\p{P}", ""))
22             .flatMap(line -> patron.splitAsStream(line))
23             .collect(Collectors.groupingBy(String::toLowerCase,
24                TreeMap::new, Collectors.counting()));
25
26       // muestra las palabras agrupadas por letra inicial
27       cuentasPalabras.entrySet()
28          .stream()
29          .collect(
30             Collectors.groupingBy(entry -> entry.getKey().charAt(0),
31                TreeMap::new, Collectors.toList()))
32          .forEach((letra, wordList) ->
33             {
34                System.out.printf("%n%C%n", letra);
35                wordList.stream().forEach(word -> System.out.printf(
36                   "%13s: %d%n", word.getKey(), word.getValue()));
37             });
38    }
39 } // fin de la clase FlujoDeLineas
```

Fig. 17.17 | Contar las ocurrencias de palabras en un archivo de texto.

```
A                    I                    Q
             a: 2          imprimen: 1           que: 4
    aplicacion: 1           indican: 1
  aplicaciones: 1          instruir: 1    R
     aprendera: 1        introduce: 1          realice: 1
   aritmeticos: 1                              recibe: 1
                        J                    resultado: 2
C                             java: 1        resultados: 1
       calcula: 1              jdk: 1
      calculos: 1                          S
      capitulo: 2        L                           su: 1
      comandos: 1              la: 6               suma: 1
    comenzamos: 1             las: 1               sus: 1
          como: 1           lector: 1
       compara: 1            linea: 1        T
   comparacion: 1              los: 2              tomar: 1
      compilar: 1            luego: 1
   computadora: 1                          U
           con: 1        M                          ultimo: 1
                              mensajes: 2             un: 3
D                              muestra: 1            usara: 1
            de: 7             muestran: 1              uso: 1
    decisiones: 1                                   usuario: 1
           del: 1        N
     demuestra: 1               numeros: 2     Y
     despliega: 1                                       y: 3
       despues: 1          O
           dos: 2                obtiene: 1

E                        P
      ejecutar: 1              pantalla: 1
       ejemplo: 1                   para: 3
      ejemplos: 1             posterior: 1
            el: 3          presentamos: 1
            en: 2             programa: 1
         este: 2           programacion: 1
                             programas: 2
G
        guarde: 1

H
  herramientas: 1
```

Fig. 17.18 | Resultados del programa de la figura 17.17 dispuestos en tres columnas.

Sintetizar las ocurrencias de cada palabra en el archivo

En las líneas 19 a 24 se sintetiza el contenido del archivo de texto "`ParrafoCapitulo2.txt`" (que se encuentra en la carpeta con el ejemplo) en un `Map<String, Long>` en el que cada clave `String` es una palabra en el archivo y el correspondiente valor `Long` es el número de ocurrencias de esa palabra. La instrucción realiza las siguientes tareas:

- En la línea 20 se usa el método **`lines`** de `Files` para crear un `Stream<String>` y leer las líneas de texto de un archivo. La clase `Files` (paquete `java.nio.file`) es una de muchas clases de las API de Java que se mejoraron para soportar objetos `Stream`.
- En la línea 21 se usa el método `map` de `Stream` para eliminar toda la puntuación, excepto los apóstrofes, en las líneas de texto. El argumento lambda representa un objeto `Function` que invoca al método `replaceAll` de `String` en su argumento `String`. Este método recibe dos argumentos: el primero es el objeto `String` de expresión regular con el que se busca coincidir y el segundo es un `String` con el que se sustituye cada coincidencia. En la expresión regular, "`(?!')`"

indica que el resto de la expresión regular debe ignorar los apóstrofes (como en una contracción en inglés tal como "you'll") y "\\p{P}" coincide con cualquier carácter de puntuación. En cualquier coincidencia, la llamada a `replaceAll` elimina la puntuación sustituyéndola con un objeto `String` vacío. El resultado de la línea 21 es un objeto `Stream<String>` intermedio que contiene las líneas sin puntuación.

- En la línea 22 se usa el método **`flatMap`** de `Stream` para dividir cada línea de texto en sus palabras separadas. El método `flatMap` recibe un objeto `Function` que asigna un objeto a un flujo de elementos. En este caso, el objeto es un `String` que contiene palabras y el resultado es otro `Stream<String>` intermedio para las palabras individuales. La lambda en la línea 22 pasa el `String` que representa una línea de texto al método **`splitAsStream`** de `Pattern` (nuevo en Java SE 8), que usa la expresión regular especificada en el `Pattern` (línea 16) para dividir en tokens el objeto `String` y obtener sus palabras individuales.

- En las líneas 23 y 24 se usa el método `collect` de `Stream` para contar la frecuencia de cada palabra y colocar las palabras y sus conteos en el objeto `TreeMap<String, Long>`. Aquí usamos una versión del método `groupingBy` de `Collectors` que recibe tres argumentos: un clasificador, una fábrica `Map` y un `Collector` de flujo descendente. El clasificador es un objeto `Function` que devuelve objetos para usarlos como claves en el `Map` resultante; la referencia a método `String::toLowerCase` convierte cada palabra en el `Stream<String>` a minúsculas. La fábrica `Map` es un objeto que implementa a la interfaz `Supplier` y devuelve una nueva colección `Map`. La *referencia al constructor* `TreeMap::new` devuelve un `TreeMap` que mantiene sus claves ordenadas. `Collectors.counting()` es el `Collector` de flujo descendente que determina el número de ocurrencias de cada clave en el flujo.

Mostrar el resumen agrupado por letra inicial

A continuación, en las líneas 27 a 37 se agrupan los pares clave-valor en el objeto `Map cuentasPalabras` con base en la primera letra de las claves. Esto produce un nuevo `Map` en el que cada clave es un `Character` y el valor correspondiente es un objeto `List` de los pares clave-valor en `cuentasPalabras` en donde la clave comienza con el `Character`. La instrucción realiza las siguientes tareas:

- Primero necesitamos obtener un `Stream` para procesar los pares clave-valor en `cuentasPalabras`. La interfaz `Map` no contiene métodos que devuelvan objetos `Stream`. Por ende, en la línea 27 se invoca el método `entrySet` de `Map` sobre `cuentasPalabras` para obtener un objeto `Set` de objetos **`Map.Entry`**, cada uno de los cuales contiene un par clave-valor de `cuentasPalabras`. Esto produce un objeto de tipo `Set<Map.Entry<String, Long>>`.

- En la línea 28 se invoca el método `stream` de `Set` para obtener un `Stream<Map.Entry<String, Long>>`.

- En las líneas 29 a 31 se invoca el método `collect` de `Stream` con tres argumentos: un clasificador, una fábrica `Map` y un `Collector` de flujo descendente. El objeto `Function` clasificador en este caso obtiene la clave del `Map`. Después `Entry` usa el método `charAt` de `String` para obtener el primer carácter de la clave; esto se convierte en una clave `Character` en el `Map` resultante. Una vez más, usamos la referencia al constructor `TreeMap::new` como la fábrica `Map` para crear un `TreeMap` que mantenga sus claves ordenadas. El `Collector` de flujo descendente (`Collectors.toList()`) coloca los objetos `Map.Entry` en una colección `List`. El resultado de `collect` es un `Map<Character, List<Map.Entry<String, Long>>>`.

- Por último, para mostrar el resumen de las palabras y sus cuentas por letra (es decir, la concordancia), en las líneas 32 a 37 se pasa una lambda al método `forEach` de `Map`. La lambda (un `BiConsumer`) recibe dos parámetros: `letra` y `listaPalabras` que representan la clave `Character` y el valor `List` respectivamente para cada par clave-valor en el `Map` producido por la operación `collect` anterior. El cuerpo de esta lambda tiene dos instrucciones, por lo que *debe* ir encerrada entre llaves. La instrucción en la línea 34 muestra la clave `Character` en su propia línea. La instrucción en las líneas 35 y 36 obtiene un `Stream<Map.Entry<String, Long>>` de la `listaPalabras` y luego invoca al método `forEach` de `Stream` para mostrar la clave y el valor de cada objeto `Map.Entry`.

17.8 Generación de flujos de valores aleatorios

En la figura 6.7 demostramos cómo tirar un dado de seis lados 6,000,000 de veces y sintetizar las frecuencias de cada cara mediante la *iteración externa* (un ciclo `for`) y una instrucción `switch` que determinaba cuál contador incrementar. Después mostramos los resultados usando instrucciones separadas que realizaban una iteración externa. En la figura 7.7 reimplementamos la figura 6.7, sustituyendo toda la instrucción `switch` con una sola instrucción que incrementaba los contadores en un arreglo; esa versión de tirar el dado seguía usando la iteración externa para producir y sintetizar 6,000,000 de tiros aleatorios y mostrar los resultados finales. Las dos versiones anteriores de este ejemplo, usaban variables mutables para controlar la iteración externa y sintetizar los resultados. En la figura 17.19 se reimplementan esos programas con *una sola instrucción* que lo hace todo, mediante el uso de lambdas, flujos, iteración interna y sin variables mutable para tirar el dado 6,000,000 de veces, calcular las frecuencias y mostrar los resultados en pantalla.

```
1  // Fig. 17.19: IntStreamAleatorio.java
2  // Tirar un dado 6,000,000 de veces
3  import java.security.SecureRandom;
4  import java.util.Map;
5  import java.util.function.Function;
6  import java.util.stream.IntStream;
7  import java.util.stream.Collectors;
8
9  public class IntStreamAleatorio
10 {
11    public static void main(String[] args)
12    {
13       SecureRandom aleatorio = new SecureRandom();
14
15       // tira un dado 6,000,000 de veces y sintetiza los resultados
16       System.out.printf("%-6s%s%n", "Cara", "Frecuencia");
17       aleatorio.ints(6_000_000, 1, 7)
18             .boxed()
19             .collect(Collectors.groupingBy(Function.identity(),
20                Collectors.counting()))
21             .forEach((cara, frecuencia) ->
22                System.out.printf("%-6d%d%n", cara, frecuencia));
23    }
24 } // fin de la clase IntStreamAleatorio
```

Fig. 17.19 | Tirar un dado 6,000,000 de veces con flujos (parte 1 de 2).

```
Cara  Frecuencia
1     998791
2     1000798
3     1000247
4     1001559
5     997599
6     1001006
```

Fig. 17.19 | Tirar un dado 6,000,000 de veces con flujos (parte 2 de 2).

Creación de un `IntStream` de valores aleatorios

En Java SE 8, la clase `SecureRandom` sobrecargó los métodos **`ints`**, **`longs`** y **`doubles`**, que hereda de la clase `Random` (paquete `java.util`). Estos métodos devuelven objetos `IntStream`, `LongStream` y `DoubleStream`, respectivamente, que representan flujos de números aleatorios. Cada método tiene cuatro sobrecargas. Aquí describiremos las sobrecargas `ints`; los métodos `longs` y `doubles` realizan las mismas tareas para flujos de valores `long` y `double`, respectivamente:

- `ints()` crea un `IntStream` para un *flujo infinito* de valores `int` aleatorios. Un **flujo infinito** tiene un número *desconocido* de elementos: se usa una operación terminal de cortocircuito para completar el procesamiento en un flujo infinito. En el capítulo 23 (en inglés) usaremos un flujo infinito para buscar números primos con la Criba de Eratóstenes.
- `ints(long)` crea un `IntStream` con el número especificado de valores `int` aleatorios.
- `ints(int, int)` crea un `IntStream` para un *flujo infinito* de valores `int` aleatorios en el rango, comenzando con el primer argumento y hasta el segundo argumento, pero sin incluirlo.
- `ints(long, int, int)` crea un `IntStream` con el número especificado de valores `int` aleatorios en el rango, comenzando con el primer argumento y hasta el segundo argumento, pero sin incluirlo.

En la línea 17 se usa la última versión sobrecargada de `ints` para crear un `IntStream` de 6,000,000 de valores enteros aleatorios en el rango del 1 al 6.

Convertir un `IntStream` en un `Stream<Integer>`

En este ejemplo sintetizamos las frecuencias de los tiros recolectándolas en un `Map<Integer, Long>` en el que cada clave `Integer` es un lado del dado y cada valor `Long` es la frecuencia de ese lado. Por desgracia, Java no permite valores primitivos en las colecciones, por lo que para sintetizar los resultados en un `Map`, primero debemos convertir el `IntStream` en un `Stream<Integer>`. Para ello invocamos al método **`boxed`** de `IntStream`.

Sintetizar las frecuencias de los dados

En las líneas 19 a 20 se invoca el método `collect` de `Stream` para sintetizar los resultados en un `Map<Integer, Long>`. El primer argumento para el método `groupingBy` de `Collectors` (línea 19) invoca al método `static` **`identity`** de la interfaz `Function`, que crea un objeto `Function` que simplemente devuelve su argumento. Esto permite usar los valores aleatorios actuales como las claves del objeto `Map`. El segundo argumento para el método `groupingBy` cuenta el número de ocurrencias de cada clave.

Mostrar los resultados

En las líneas 21 y 22 se invoca el método `forEach` del `Map` resultante para mostrar el resumen de los resultados. Este método recibe un objeto que implementa la interfaz funcional `BiConsumer` como un

argumento. Recuerde que para los objetos `Map` el primer parámetro representa la clave y el segundo representa el valor correspondiente. La lambda en las líneas 21 y 22 usa el parámetro `cara` como la clave y `frecuencia` como el valor, y muestra tanto la cara como la frecuencia.

17.9 Manejadores de eventos de lambda

En la sección 12.11 aprendió a implementar un manejador de eventos mediante el uso de una clase interna anónima. Algunas interfaces de escucha de eventos (como `ActionListener` y `ItemListener`) son interfaces funcionales. Para dichas interfaces es posible implementar manejadores de eventos con lambdas. Por ejemplo, la siguiente instrucción de la figura 12.21:

```
imagesJComboBox.addItemListener(
   new ItemListener() // clase interna anónima
   {
      // maneja el evento JComboBox
      @Override
      public void itemStateChanged(ItemEvent evento)
      {
         // determina si se seleccionó el elemento
         if (evento.getStateChange() == ItemEvent.SELECTED)
            etiqueta.setIcon(iconos[
               imagenesJComboBox.getSelectedIndex()]);
      }
   } // fin de la clase interna anónima
); // fin de la llamada a addItemListener
```

que registra un manejador de evento para un objeto `JComboBox` puede implementarse de manera más concisa como:

```
imagenesJComboBox.addItemListener(evento -> {
   if (evento.getStateChange() == ItemEvent.SELECTED)
      etiqueta.setIcon(iconos[imagenesJComboBox.getSelectedIndex()]);
});
```

Para un manejador de eventos simple como éste, una lambda reduce de manera considerable la cantidad de código que necesita escribir.

17.10 Comentarios adicionales sobre las interfaces de Java SE 8

Las interfaces de Java SE 8 permiten la herencia de implementaciones de métodos

Las interfaces funcionales *deben* contener sólo un método `abstract`, pero también pueden contener métodos `default` y métodos `static` que se implementen por completo en las declaraciones de la interfaz. Por ejemplo, la interfaz `function` (que se usa de manera extensa en la programación funcional) tiene los métodos `apply` (`abstract`), `compose` (`default`), `andThen` (`default`) y `identity` (`static`).

Cuando una clase implementa a una interfaz con métodos `default` y *no* los sobrescribe, la clase hereda las implementaciones de los métodos `default`. El diseñador de una interfaz puede ahora evolucionar una interfaz agregando nuevos métodos `default` y `static` sin descomponer el código existente que implementa a la interfaz. Por ejemplo, la interfaz `Comparator` (sección 16.7.1) ahora contiene muchos métodos `default` y `static`, pero las clases anteriores que implementan esta interfaz aún deben poder compilarse y funcionar correctamente en Java SE 8.

Si una clase hereda el mismo método `default` de dos interfaces no relacionadas, la clase *debe* sobrescribir ese método; de lo contrario, el compilador no sabrá qué método usar y generará un error de compilación.

Java SE 8: Anotación `@FunctionalInterface`

Puede crear sus propias interfaces funcionales asegurando que cada una contenga sólo un método `abstract` y cero o más métodos `default` o `static`. Aunque no se requiere, puede declarar que una interfaz es funcional si le antepone la **anotación `@FunctionalInterface`**. Así el compilador se asegurará de que la interfaz contenga sólo un método `abstract`; de lo contrario generará un error de compilación.

17.11 Java SE 8 y los recursos de programación funcional

Visite la página Web de los autores

```
http://www.deitel.com/books/jhtp10
```

en donde encontrará los vínculos a los Centros de recursos Deitel (en inglés) que generamos mientras escribíamos *Cómo programar en Java, 10/e*.

17.12 Conclusión

En este capítulo aprendió sobre las nuevas herramientas de programación funcional de Java SE 8. Presentamos muchos ejemplos, a menudo mostrando maneras más simples de implementar tareas que programó en capítulos anteriores.

Vimos las generalidades sobre las tecnologías de programación funcional claves: interfaces funcionales, lambdas y flujos. Aprendió a procesar elementos en un `IntStream`: un flujo de valores `int`. Creó un `IntStream` para un arreglo de valores `int` y luego usó las operaciones de flujos intermedias y terminales para crear y procesar una canalización de flujo que produjo un resultado. Usó lambdas para crear métodos anónimos que implementaron interfaces funcionales.

Le mostramos cómo usar una operación terminal `forEach` para realizar una operación en cada elemento de flujo. Usamos operaciones de reducción para contar el número de elementos de un flujo, determinar los valores mínimo y máximo, y obtener la suma y promedio de los valores. También aprendió a usar el método `reduce` para crear sus propias operaciones de reducción.

Usó operaciones intermedias para filtrar elementos que coincidieran con un predicado y asignar elementos a nuevos valores; en cada caso, estas operaciones produjeron flujos intermedios en los que podía realizar un procesamiento adicional. También aprendió a ordenar elementos en forma ascendente y descendente, y cómo ordenar objetos según varios campos.

Demostramos cómo almacenar los resultados de una canalización de flujo en una colección para uso posterior. Para ello, aprovechó las diversas implementaciones de `Collector` predefinidas que proporciona la clase `Collectors`. También aprendió a usar un `Collector` para agrupar elementos en categorías.

Aprendió que las diversas clases de Java SE 8 se mejoraron para soportar la programación funcional. Luego usó el método `lines` de `Files` para obtener un `Stream<String>` que leyó líneas de texto de un archivo y usó el método `ints` de `SecureRandom` para obtener un `IntStream` de valores aleatorios. También aprendió a convertir un `IntStream` en un `Stream<Integer>` (por medio del método `boxed`) para poder usar el método `collect` de `Stream` para sintetizar las frecuencias de los valores `Integer` y almacenar los resultados en un `Map`.

Después aprendió a implementar una interfaz funcional para manejo de eventos mediante el uso de una lambda. Por último, le presentamos información adicional sobre las interfaces y flujos de Java SE 8. En el siguiente capítulo hablaremos sobre la programación recursiva, en donde los métodos se invocan a sí mismos, ya sea de manera directa o indirecta.

Resumen

Sección 17.1 Introducción

- Antes de Java SE 8, Java soportaba tres paradigmas de programación: programación por procedimientos, programación orientada a objetos y programación genérica. Java SE 8 agrega la programación funcional.
- El nuevo lenguaje y las herramientas de la biblioteca que soportan la programación funcional se agregaron a Java como parte del proyecto Lambda.

Sección 17.2 Generalidades acerca de las tecnologías de programación funcional

- Antes de la programación funcional, por lo general se determinaba lo que se quería lograr y luego se especificaban los pasos precisos para realizar esa tarea.
- Usar un ciclo para iterar sobre una colección de elementos se conoce como iteración externa (pág. 731) y requiere del acceso secuencial a los elementos. Dicha iteración también requiere variables mutables.
- En la programación funcional (pág. 732), el programador especifica lo que desea realizar en una tarea, pero no cómo lograrlo.
- Dejar que la biblioteca determine cómo iterar sobre una colección de elementos se conoce como iteración interna (pág. 732). La iteración interna es fácil de paralelizar.
- La programación funcional se enfoca en la inmutabilidad (pág. 732); es decir, no modificar el origen de datos que se está procesando o cualquier otro estado del programa.

Sección 17.2.1 Interfaces funcionales

- Las interfaces funcionales se conocen también como interfaces de un solo método abstracto (SAM).
- El paquete `java.util.function` contiene las seis interfaces funcionales básicas `BinaryOperator`, `Consumer`, `Function`, `Predicate`, `Supplier` y `UnaryOperator`.
- Hay muchas versiones especializadas de las seis interfaces funcionales básicas para usar con valores primitivos `int`, `long` y `double`. También hay personalizaciones genéricas de `Consumer`, `Function` y `Predicate` para operaciones binarias; es decir, métodos que reciben dos argumentos.

Sección 17.2.2 Expresiones lambda

- Una expresión lambda (pág. 733) representa a un método anónimo; una notación abreviada para implementar una interfaz funcional.
- El tipo de lambda es el tipo de interfaz funcional que la lambda implementa.
- Las expresiones lambda pueden usarse en donde se esperan interfaces funcionales.
- Una lambda consiste en una lista de parámetros seguida por el token de flecha (->, pág. 733) y un cuerpo, como en:

  (*listaParámetros*) -> {*instrucciones*}

  Por ejemplo, la siguiente lambda recibe dos valores `int` y devuelve su suma:

  ```
  (int x, int y) -> {return x + y;}
  ```

  El cuerpo de esta lambda es un bloque de instrucciones que puede contener una o más instrucciones encerradas entre llaves.
- Los tipos de los parámetros de una lambda pueden omitirse, como en:

  ```
  (x, y) -> {return x + y;}
  ```

  en cuyo caso, los tipos de los parámetros y del valor de retorno los determina el contexto de la lambda.
- Una lambda con un cuerpo de una sola expresión puede escribirse así:

  ```
  (x, y) -> x + y
  ```

  En este caso, el valor de la expresión se devuelve de manera implícita.

- Cuando la lista de parámetros contiene sólo un parámetro pueden omitirse los paréntesis, como en:

```
valor -> System.out.printf("%d ", valor)
```

- Una lambda con una lista de parámetros vacía se define con `()` a la izquierda del token de flecha (->), como en:

```
() -> System.out.println("Bienvenido a las lambdas!")
```

- También hay formas abreviadas especializadas de lambdas que se conocen como referencias a métodos.

Sección 17.2.3 Flujos

- Los flujos (pág. 734) son objetos que implementan a la interfaz `Stream` (del paquete `java.util.stream`) y permiten al programador realizar tareas de programación funcional. También hay interfaces de flujos especializadas para procesar valores `int`, `long` o `double`.
- Los flujos mueven elementos a través de una secuencia de pasos de procesamiento (lo que se conoce como una canalización de flujo) que comienza con un origen de datos, realiza varias operaciones intermedias sobre los elementos del origen de datos y finaliza con una operación terminal. Una canalización de flujo se forma encadenando llamadas a métodos.
- A diferencia de las colecciones, los flujos no tienen su propio almacenamiento; una vez que se procesa un flujo, no puede reutilizarse ya que no mantiene una copia de la fuente de datos original.
- Una operación intermedia (pág. 734) especifica las tareas a realizar sobre los elementos del flujo y siempre produce un nuevo flujo.
- Las operaciones intermedias son perezosas (pág. 734); no se realizan sino hasta que se invoque una operación terminal. Esto permite a los desarrolladores de bibliotecas optimizar el rendimiento del procesamiento de flujos.
- Una operación terminal (pág. 734) inicia el procesamiento de las operaciones intermedias de una canalización de flujo y produce un resultado. Las operaciones terminales son ansiosas (pág. 734); es decir, realizan la operación solicitada cuando se invocan.

Sección 17.3 Operaciones con `IntStream`

- Un `IntStream` (paquete `java.util.stream`) es un flujo especializado para manipular valores `int`.

Sección 17.3.1 Creación de un `IntStream` e impresión en pantalla de sus valores con la operación terminal `forEach`

- El método `static of` de `IntStream` (pág. 739) recibe un arreglo `int` como argumento y devuelve un `IntStream` para procesar los valores del arreglo.
- El método `forEach` de `IntStream` (una operación terminal; pág. 738) recibe como argumento un objeto que implementa a la interfaz funcional `IntConsumer` (paquete `java.util.function`). El método `accept` de esta interfaz recibe un valor `int` y realiza una tarea con él.
- El compilador de Java puede inferir los tipos de los parámetros de una lambda y el tipo de valor de retorno de una lambda a partir del contexto en el que ésta se utilice. Esto se determina mediante el tipo de destino de la lambda (pág. 738): el tipo de la interfaz funcional que se espera en donde la lambda aparece en el código.
- Las lambdas pueden usar variables locales `final` o variables locales efectivamente `final` (pág. 738).
- Una lambda que hace referencia a una variable local en el alcance léxico circundante se conoce como lambda de captura.
- Una lambda puede usar la referencia `this` de la clase externa sin necesidad de calificarla con el nombre de la clase exterior.
- Los nombres de los parámetros y de las variables que utilice en las lambdas no pueden ser los mismos que los de otras variables locales en el alcance léxico de la lambda; de lo contrario, se producirá un error de compilación.

Sección 17.3.2 Operaciones terminales `count`, `min`, `max`, `sum` y `average`

- La clase `IntStream` proporciona operaciones terminales para reducciones de flujos comunes: `count` (pág. 739) devuelve el número de elementos, `min` devuelve el `int` más pequeño, `max` devuelve el `int` más grande, `sum` devuelve la suma de todos los valores `int` y `average` devuelve un `OptionalDouble` (paquete `java.util`) que contiene el promedio de los valores `int` como el valor de tipo `double`.

- El método `getAsDouble` de la clase `OptionalDouble` devuelve el `double` en el objeto o lanza una excepción `NoSuchElementException`. Para evitar esta excepción puede llamar al método `orElse`, que devuelve el valor del `OptionalDouble` si hay uno, o el valor que pasa a `orElse` en cualquier otro caso.
- El método `summaryStatistics` de `IntStream` realiza las operaciones `count`, `min`, `max`, `sum` y `average` en una pasada de los elementos de un `IntStream` y devuelve los resultados como un objeto `IntSummaryStatistics` (paquete `java.util`).

Sección 17.3.3 Operación terminal `reduce`

- Puede definir sus propias reducciones para un `IntStream` invocando a su método `reduce`. El primer argumento es un valor que le ayuda a comenzar la operación de reducción y el segundo argumento es un objeto que implementa la interfaz funcional `IntBinaryOperator` (pág. 740).
- El primer argumento del método `reduce` se conoce formalmente como un valor de identidad (pág. 740); es decir, un valor que si se combina con un elemento de flujo mediante el uso de `IntBinaryOperator`, produce el valor original de ese elemento.

Sección 17.3.4 Operaciones intermedias: filtrado y ordenamiento de valores `IntStream`

- Hay que filtrar elementos para producir un flujo de resultados inmediatos que coinciden con un predicado. El método `filter` de `IntStream` (pág. 741) recibe un objeto que implementa la interfaz funcional `IntPredicate` (paquete `java.util.function`).
- El método `sorted` de `IntStream` (una operación perezosa) ordena los elementos del flujo en forma ascendente (de manera predeterminada). Todas las operaciones intermedias anteriores en la canalización de flujo deben completarse, de modo que el método `sorted` sepa qué elementos ordenar.
- El método `filter` es una operación intermedia sin estado; no requiere información sobre los otros elementos en el flujo para probar si el elemento actual satisface el predicado.
- El método `sorted` es una operación intermedia con estado que requiere información sobre todos los demás elementos en el flujo para poder ordenarlos.
- El método `default and` de la interfaz `IntPredicate` (pág. 741) realiza una operación AND lógica con evaluación de cortocircuito entre el `IntPredicate` sobre el cual se invoca y su argumento `IntPredicate`.
- El método `default negate` de `IntPredicate` (pág. 741) invierte el valor `boolean` del `IntPredicate` sobre el cual se invoca.
- El método `default or` de la interfaz `IntPredicate` (pág. 741) realiza una operación OR lógica con evaluación de cortocircuito entre el `IntPredicate` sobre la cual se invoca y su argumento `IntPredicate`.
- Puede usar los métodos `default` de la interfaz `IntPredicate` para elaborar condiciones más complejas.

Sección 17.3.5 Operación intermedia: asignación

- La asignación es una operación intermedia que transforma los elementos de un flujo en nuevos valores y produce un flujo que contiene los elementos resultantes (posiblemente de un tipo diferente).
- El método `map` de `IntStream` (una operación intermedia sin estado; pág. 742) recibe un objeto que implementa la interfaz funcional `IntUnaryOperator` (paquete `java.util.function`).

Sección 17.3.6 Creación de flujos de valores `int` *con los métodos* `range` *y* `rangeClosed` *de* `IntStream`

- Los métodos `range` (pág. 743) y `rangeClosed` de `IntStream` pueden producir cada uno una secuencia de valores `int`. Ambos métodos reciben dos argumentos `int` que representan el rango de valores. El método `range` produce una secuencia de valores desde su primer argumento hasta su segundo argumento, pero sin incluirlo. El método `rangeClosed` produce una secuencia de valores, incluyendo ambos argumentos.

Sección 17.4 Manipulaciones de objetos `Stream<Integer>`

- El método `stream` de la clase `Array` se usa para crear un `Stream` a partir de un arreglo de objetos.

Sección 17.4.1 Creación de un `Stream<Integer>`

- La interfaz `Stream` (paquete `java.util.stream`; pág. 744) es una interfaz genérica para realizar operaciones de flujos en objetos. Los tipos de objetos que se procesan los determina el origen del `Stream`.
- La clase `Arrays` proporciona los métodos `stream` sobrecargados para crear objetos `IntStream`, `LongStream` y `DoubleStream` a partir de arreglos `int`, `long` y `double`, o a partir de rangos de elementos en los arreglos.

Sección 17.4.2 Ordenamiento de un objeto `Stream` y recolección de los resultados

- De manera predeterminada, el método `sorted` de `Stream` (pág. 745) ordena los elementos de un flujo en forma ascendente.
- Para crear una colección que contenga los resultados de una canalización de flujo, puede usar el método `collect` de `Stream` (una operación terminal). A medida que se procesa la canalización de flujo, el método `collect` realiza una operación de reducción mutable que coloca los resultados en un objeto como `List`, `Map` o `Set`.
- El método `collect` con un argumento recibe un objeto que implementa la interfaz `Collector` (paquete `java.util.stream`), que especifica cómo realizar la reducción mutable.
- La clase `Collectors` (paquete `java.util.stream`) proporciona métodos `static` que devuelven implementaciones predefinidas de `Collector`.
- El método `toList` de `Collectors` transforma un `Stream<T>` en una colección `List<T>`.

Sección 17.4.3 Filtrado de un `Stream` y almacenamiento de los resultados para su uso posterior

- El método `filter` de `Stream` (pág. 745) recibe un `Predicate` y produce un flujo de objetos que coinciden con el `Predicate`. El método `test` de `Predicate` devuelve un `boolean` que indica si el argumento satisface una condición. La interfaz `Predicate` también tiene los métodos `and`, `negate` y `or`.

Sección 17.4.5 Ordenamiento de los resultados recolectados previamente

- Una vez que coloque los resultados de una canalización de flujo en una colección, podrá crear un nuevo flujo a partir de la colección para realizar operaciones de flujo adicionales con los resultados anteriores.

Sección 17.5.1 Asociación de objetos `String` a mayúsculas mediante la referencia a un método

- El método `map` de `Stream` (pág. 747) asigna cada elemento a un nuevo valor y produce un nuevo flujo con el mismo número de elementos que el flujo original.
- Una referencia a método (pág. 747) es una notación abreviada para una expresión lambda.
- *NombreClase*::*nombreMétodoInstancia* representa la referencia a un método para un método de instancia de una clase. Crea una lambda de un parámetro que invoca al método de instancia con el argumento de la lambda y devuelve el resultado del método.
- *nombreObjeto*::*nombreMétodoInstancia* representa la referencia a método para un método de instancia que debe invocarse sobre un objeto específico. Crea una lambda de un parámetro que invoca al método de instancia sobre el objeto especificado (pasando el argumento de la lambda al método de instancia) y devuelve el resultado del método.
- *NombreClase*::*nombreMétodoEstático* representa una referencia a método para un método `static` de una clase. Crea una lambda de un parámetro en donde el argumento de la lambda se pasa al método `static` especificado y la lambda devuelve el resultado del método.
- *NombreClase*::*new* representa la referencia a un constructor. Crea una lambda que invoca al constructor sin argumentos de la clase especificada para crear e inicializar un nuevo objeto de esa clase.

17.5.2 Filtrado de objetos `String` y ordenamiento ascendente sin distinguir entre mayúsculas y minúsculas

- El método `sorted` de `Stream` (pág. 748) puede recibir un `Comparator` como argumento para especificar cómo comparar elementos de flujo para ordenarlos.
- De manera predeterminada, el método `sorted` usa el orden natural para el tipo de elemento del flujo.

- Para los objetos `String`, el orden natural es susceptible al uso de mayúsculas y minúsculas, lo que significa que "Z" es menor que "a". Al pasar el `Comparator String.CASE_INSENSITIVE_ORDER` predefinido se realiza un ordenamiento sin susceptibilidad al uso de mayúsculas minúsculas.

Sección 17.5.3 Filtrado de objetos `String` y ordenamiento descendente sin distinguir entre mayúsculas y minúsculas

- El método `default reversed` de la interfaz funcional `Comparator` (pág. 748) invierte el ordenamiento de un `Comparator` existente.

Sección 17.6.1 Creación e impresión en pantalla de un objeto `List<Empleado>`

- Cuando la referencia al método de instancia `System.out::println` se pasa al método `forEach` de `Stream`, el compilador la convierte en un objeto que implementa a la interfaz funcional `Consumer`. El método `accept` de esta interfaz recibe un argumento y devuelve `void`. En este caso, el método `accept` pasa el argumento al método de instancia `println` del objeto `System.out`.

Sección 17.6.2 Filtrado de objetos `Empleado` con salarios en un rango especificado

- Para reutilizar una lambda, puede asignarla a una variable del tipo de interfaz funcional apropiado.
- El método `static comparing` de la interfaz `Comparator` (pág. 753) recibe un objeto `Function` que se usa para extraer un valor de un objeto en el flujo y usarlo en comparaciones; luego devuelve un objeto `Comparator`.
- Una agradable característica de rendimiento de la evaluación perezosa es la habilidad de realizar una evaluación de cortocircuito; es decir, detener el procesamiento de la canalización de flujo tan pronto como esté disponible el resultado deseado.
- El método `findFirst` de `Stream` es una operación terminal de cortocircuito que procesa la canalización de flujo y termina el procesamiento tan pronto como se encuentra el primer objeto de la canalización de flujo. El método devuelve un `Optional` que contiene el objeto encontrado, si lo hay.

Sección 17.6.3 Ordenamiento de objetos `Empleado` según varios campos

- Para ordenar objetos con base en dos campos, hay que crear un `Comparator` que use dos objetos `Function`. Primero se invoca al método `comparing` de `Comparator` para crear un `Comparator` con el primer objeto `Function`. En el `Comparator` resultante, puede invocar al método `thenComparing` con el segundo objeto `Function`. El `Comparator` resultante compara los objetos usando el primer objeto `Function` y luego, para los objetos que sean iguales, los compara mediante el segundo objeto `Function`.

Sección 17.6.4 Asociación de objetos `Empleado` a objetos `String` con apellidos únicos

- Puede asignar objetos en un flujo a tipos diferentes para producir otro flujo con el mismo número de elementos que el flujo original.
- El método `distinct` de `Stream` (pág. 754) elimina los objetos duplicados en un flujo.

Sección 17.6.5 Agrupación de objetos `Empleado` por departamento

- El método `static groupingBy` de `Collectors` (pág. 755) con un argumento recibe un objeto `Function` que clasifica objetos en el flujo; los valores devueltos por esta función se usan como las claves en un objeto `Map`. De manera predeterminada, los valores correspondientes son objetos `List` que contienen los elementos del flujo en una categoría dada.
- El método `forEach` de `Map` realiza una operación sobre cada par clave-valor. El método recibe un objeto que implementa a la interfaz funcional `BiConsumer`. El método `accept` de esta interfaz tiene dos parámetros. En los objetos `Map` el primero representa la clave y el segundo el valor correspondiente.

Sección 17.6.6 Conteo del número de objetos `Empleado` en cada departamento

- El método `static groupingBy` de `Collectors` con dos argumentos recibe un objeto `Function` que clasifica los objetos en el flujo y otro `Collector` (conocido como el `Collector` de flujo descendente; pág. 756).
- El método `static counting` de `Collectors` devuelve un `Collector` que cuenta el número de objetos en una clasificación dada, en vez de recolectarlos en un objeto `List`.

Sección 17.6.7 Suma y promedio de salarios de objetos `Empleado`

- El método `mapToDouble` de `Stream` (pág. 756) asigna objetos a valores `double` y devuelve un `DoubleStream`. El método recibe un objeto que implementa a la interfaz funcional `ToDoubleFunction` (paquete `java.util.function`). El método `applyAsDouble` de esta interfaz invoca a un método de instancia sobre un objeto y devuelve un valor `double`.

Sección 17.7 Creación de un objeto `Stream<String>` a partir de un archivo

- El método `lines` de `Files` crea un `Stream<String>` para leer las líneas de texto de un archivo.
- El método `flatMap` de `Stream` (pág. 760) recibe un objeto `Function` que asigna un objeto a un flujo; por ejemplo, una línea de texto a palabras.
- El método `splitAsStream de Pattern` (pág. 760) usa una expresión regular para dividir un objeto `String` en tokens.
- El método `groupingBy` de `Collectors` con tres argumentos recibe un clasificador, una fábrica `Map` y un `Collector` de flujo descendente. El clasificador es un objeto `Function` que devuelve objetos que se usan como claves en el `Map` resultante. La fábrica `Map` es un objeto que implementa a la interfaz `Supplier` y devuelve una nueva colección `Map`. El `Collector` de flujo descendente determina cómo recolectar los elementos de cada grupo.
- El método `entrySet` de `Map` devuelve un objeto `Set` de objetos `Map.Entry` que contienen los pares clave-valor del `Map`.
- El método `stream` de `Set` devuelve un flujo para procesar los elementos del objeto `Set`.

Sección 17.8 Generación de flujos de valores aleatorios

- Los métodos `ints` (pág. 762) `longs` y `doubles` de la clase `SecureRandom` (heredados de la clase `Random`) devuelven objetos `IntStream`, `LongStream` y `DoubleStream` respectivamente, para flujos de números aleatorios.
- El método `ints` sin argumentos crea un `IntStream` para un flujo infinito (pág. 762) de valores `int` aleatorios. Un flujo infinito es un flujo con un número desconocido de elementos; se usa una operación terminal de cortocircuito para completar el procesamiento en un flujo infinito.
- El método `ints` con un argumento `long` crea un `IntStream` con el número especificado de valores `int` aleatorios.
- El método `ints` con dos argumentos `int` crea un `IntStream` para un flujo infinito de valores `int` aleatorios en el rango a partir del primer argumento y hasta el segundo, pero sin incluirlo.
- El método `ints` con un argumento `long` y dos argumentos `int` crea un `IntStream` con el número especificado de valores `int` aleatorios en el rango, comenzando con el primer argumento y hasta el segundo, pero sin incluirlo.
- Para convertir un `IntStream` en un `Stream<Integer>` hay que invocar al método `boxed` de `IntStream`.
- El método `static identity` de `Function` crea un objeto `Function` que simplemente devuelve su argumento.

Sección 17.9 Manejadores de eventos de lambda

- Algunas interfaces de escucha de eventos son interfaces funcionales (pág. 763). Para dichas interfaces se puede implementar manejadores de eventos con lambdas. Para un manejador de eventos simple, una lambda reduce de manera considerable la cantidad de código que necesita escribir.

17.10 Comentarios adicionales sobre las interfaces de Java SE 8

- Las interfaces funcionales deben contener sólo un método `abstract`, pero también pueden contener métodos `default` y métodos `static` que se implementen por completo en las declaraciones de la interfaz.
- Cuando una clase implementa a una interfaz con métodos `default` y no los sobrescribe, la clase hereda las implementaciones de los métodos `default`. El diseñador de una interfaz puede ahora evolucionar una interfaz al agregar nuevos métodos `default` y `static` sin quebrantar el código existente que implementa a la interfaz.

- Si una clase hereda el mismo método `default` de dos interfaces, la clase debe sobrescribir ese método; de lo contrario, el compilador generará un error de compilación.
- Puede crear sus propias interfaces funcionales asegurando que cada una contenga sólo un método `abstract` y cero o más métodos `default` o `static`.
- Puede declarar que una interfaz es funcional si le antepone la anotación `@FunctionalInterface`. Así el compilador se asegurará de que la interfaz contenga sólo un método `abstract`; de lo contrario generará un error de compilación.

Ejercicios de autoevaluación

17.1 Complete los espacios en blanco de cada uno de los siguientes enunciados:

a) Las expresiones lambda implementan _____.
b) Los programas funcionales son más fáciles de _____ (es decir, realizar varias operaciones al mismo tiempo) de modo que sus programas puedan aprovechar las arquitecturas multinúcleo para mejorar el rendimiento.
c) Con la iteración _____ la biblioteca determina cómo acceder a todos los elementos en una colección para realizar una tarea.
d) La interfaz funcional _____ contiene el método `apply` que recibe dos argumentos `T`, realiza una operación sobre ellos (como un cálculo) y devuelve un valor de tipo `T`.
e) La interfaz funcional _____ contiene el método `test` que recibe un argumento `T` y devuelve un `boolean`; además evalúa si el argumento `T` satisface una condición.
f) Una _____ representa a un método anónimo: una notación abreviada para implementar una interfaz funcional.
g) Las operaciones de flujo intermedias son _____, ya que no se realizan sino hasta que se invoca a una operación terminal.
h) La operación de flujo terminal _____ realiza el procesamiento sobre cada elemento en un flujo.
i) Las lambdas _____ usan variables locales del alcance léxico circundante.
j) Una característica de rendimiento de la evaluación perezosa es la habilidad de realizar la evaluación _____; es decir, detener el procesamiento de la canalización de flujo tan pronto como esté disponible el resultado deseado.
k) Para los objetos `Map`, el primer parámetro de un objeto `BiConsumer` representa la _____ y su segundo argumento representa el _____ correspondiente.

17.2 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Las expresiones lamba pueden usarse en cualquier parte en donde se esperen interfaces funcionales.
b) Las operaciones terminales son perezosas, ya que realizan la operación solicitada cuando se invocan.
c) El primer argumento del método `reduce` se conoce formalmente como valor de identidad: un valor que cuando se combina con un elemento de flujo usando el `IntBinaryOperator` produce el valor original del elemento de flujo. Por ejemplo, al sumar los elementos, el valor de identidad es 1 y al obtener el producto de los elementos el valor de identidad es 0.
d) El método `findFirst` de `Stream` es una operación terminal de cortocircuito que procesa la canalización de flujo pero termina el procesamiento tan pronto como se encuentra un objeto.
e) El método `flatMap` de `Stream` recibe un objeto `Function` que asigna un flujo a un objeto. Por ejemplo, el objeto podría ser un `String` que contenga palabras y el resultado podría ser otro `Stream<String>` intermedio para las palabras individuales.
f) Cuando una clase implementa a una interfaz con métodos `default` y los sobrescribe, la clase hereda las implementaciones de los métodos `default`. Ahora un diseñador de una interfaz puede evolucionarla agregando nuevos métodos `default` y `static` sin descomponer el código existente que implementa a la interfaz.

17.3 Escriba una lambda o referencia a método para cada una de las siguientes tareas:

a) Escriba una lambda que pueda usarse en vez de la siguiente clase interna anónima:

```
new IntConsumer()
{
  public void aceptar(int valor)
  {
    System.out.printf("%d ", valor);
  }
}
```

b) Escriba una referencia a método que pueda usarse en lugar de la siguiente lambda:

```
(String s) -> {return s.toUpperCase();}
```

c) Escriba una lambda sin argumentos que devuelva de manera implícita el objeto `String` "`Bienvenido a las lambdas!`".

d) Escriba una referencia a método para el método `sqrt` de `Math`.

e) Cree una lambda sin parámetros que devuelva el cubo de su argumento.

Respuestas a los ejercicios de autoevaluación

17.1 a) interfaces funcionales. b) paralelizar. c) interna. d) `BinaryOperator<T>`. e) `Predicate<T>`. f) expresión lambda. g) perezosas. h) `forEach`. i) de captura. j) de cortocircuito. k) clave, valor.

17.2 a) Verdadero. b) Falso. Las operaciones terminales son *ansiosas*: realizan la operación solicitada cuando se invocan. c) Falso. Al sumar los elementos, el valor de identidad es 0 y al obtener el producto de los elementos, el valor de identidad es 1. d) Verdadero. e) Falso. El método `flatMap` de `Stream` recibe un objeto `Function` que asigna un objeto a un flujo. f) Falso. Debe decir: "… no los sobrescribe, …" en vez de "los sobrescribe".

17.3 a) `valor -> System.out.printf("%d ", valor)`

b) `String::toUpperCase`

c) `() -> "Bienvenido a las lambdas!"`

d) `Math::sqrt`

e) `valor -> valor * valor * valor`

Ejercicios

17.4 Complete los espacios en blanco en cada uno de los siguientes enunciados:

a) Las _____ de flujos se forman a partir de orígenes de flujos, operaciones intermedias y operaciones terminales.

b) El siguiente código usa la técnica de la iteración _____:

```
int suma = 0;
for (int contador = 0; contador < valores.length; contador++)
  suma += valores[contador];
```

c) Las herramientas de programación funcional se enfocan en _____: no modificar los orígenes de datos que se están procesando, o cualquier otro estado del programa.

d) La interfaz funcional _____ contiene el método `accept` que recibe un argumento `T` y devuelve `void`; `accept` realiza una tarea con su argumento `T`, como mostrar en pantalla el objeto, invocar a un método del objeto, etc.

e) La interfaz funcional _____ contiene el método `get` que no recibe argumentos y produce un valor de tipo `T`; a menudo esto se usa para crear un objeto colección en donde se colocan los resultados de la operación del flujo.

f) Los flujos son objetos que implementan a la interfaz `Stream` y le permiten realizar tareas de programación funcional con _____ de elementos.
g) La operación de flujo intermedia _____ da como resultado un flujo que contiene sólo los elementos que satisfacen una condición.
h) _____ colocan los resultados de procesar una canalización de flujo en una colección tal como un objeto `List`, `Set` o `Map`.
i) Las llamadas a `filter` y otros flujos intermedios son perezosas, ya que no se evalúan sino hasta que se realice una operación _____ ansiosa.
j) El método _____ de `Pattern` (nuevo en Java SE 8) usa una expresión regular para dividir un objeto `String`.
k) Las interfaces funcionales *deben* contener sólo un método _____, pero también pueden contener métodos _____ y métodos `static` que se implementen por completo en las declaraciones de la interfaz.

17.5 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
a) Una operación intermedia especifica las tareas a realizar sobre los elementos del flujo; esto es eficiente debido a que evita la creación de un nuevo flujo.
b) Las operaciones de reducción toman todos los valores en el flujo y los convierten en un nuevo flujo.
c) Si necesita una secuencia ordenada de valores `int`, puede crear un `IntStream`, que contiene dichos valores con los métodos `range` y `rangeClosed` de `IntStream`. Ambos métodos reciben dos argumentos `int` que representan el rango de valores. El método `rangeClosed` produce una secuencia de valores desde su primer argumento hasta su segundo argumento, pero sin incluirlo. El método `range` produce una secuencia de valores, incluyendo sus dos argumentos.
d) La clase `Files` (paquete `java.nio.file`) es una de muchas clases de las API de Java que se mejoraron para soportar los objetos `Stream`.
e) La interfaz `Map` no contiene métodos que devuelven objetos `Stream`.
f) La interfaz funcional `Function` (que se utiliza mucho en la programación funcional) tiene los método `apply` (`abstract`), `compose` (`abstract`), `andThen` (`default`) y `identity` (`static`).
g) Si una clase hereda el mismo método `default` de dos interfaces, la clase *debe* sobrescribir ese método; de lo contrario el compilador no sabe qué método usar y genera un error de compilación.

17.6 Escriba una lambda o una referencia a método para cada una de las siguientes tareas:
a) Escriba una expresión lambda que reciba dos parámetros `double` llamados `a` y `b`, y que devuelva su producto. Use la forma de lambda que liste de manera explícita el tipo de cada parámetro.
b) Vuelva a escribir la expresión lambda del inciso (a) usando la forma de lambda que no lista el tipo de cada parámetro.
c) Vuelva a escribir la expresión lambda del inciso (b) usando la forma de lambda que devuelve de manera implícita el valor de la expresión del cuerpo de la lambda.
d) Escriba una lambda sin argumentos que devuelva de manera implícita la cadena "`Bienvenido a las lambdas!`".
e) Escriba una referencia a constructor para la clase `ArrayList`.
f) Vuelva a implementar la siguiente instrucción usando una lambda como el manejador de eventos.

```
boton.addActionListener(
  new ActionListener()
  {
    public void actionPerformed(ActionEvent evento)
    {
      JoptionPane.showMessageDialog(ParentFrame.this,
        "JButton event handler");
    }
  }
);
```

17.7 Suponiendo que `lista` es un `List<Integer>`, explique en detalle la canalización de flujo:

```
lista.stream()
  .filter(valor -> valor % 2 != 0)
  .sum()
```

17.8 Suponiendo que `aleatorio` sea un objeto `SecureRandom`, explique en detalle la canalización de flujo:

```
random.ints(1000000, 1, 3)
    .boxed()
    .collect(Collectors, groupingBy(Function.identity(),
      Collectors.counting()))
    .forEach((lado, frecuencia) ->
      System.out.printf("%-6d%d%n", lado, frecuencia));
```

17.9 ***(Sintetizar los caracteres en un archivo)*** Modifique el programa de la figura 17.17 para sintetizar el número de ocurrencias de cada carácter en el archivo.

17.10 ***(Sintetizar los tipos de archivos en un directorio)*** En la sección 15.3 se demostró cómo obtener información sobre los archivos y directorios en el disco. Además, usó un `DirectoryStream` para mostrar el contenido de un directorio. La interfaz `DirectoryStream` ahora contiene el método `default entries`, que devuelve un `Stream`. Use las técnicas de la sección 15.3, el método `entries` de `DirectoryStream`, lambdas y flujos para sintetizar los tipos de archivos en un directorio especificado.

17.11 ***(Eliminación de palabras duplicadas)*** Escriba un programa que reciba un enunciado del usuario (asuma que no hay signos de puntuación), luego determine y muestre las palabras únicas en orden alfabético. Trate igual las letras mayúsculas y minúsculas.

17.12 ***(Ordenar letras y eliminar duplicados)*** Escriba un programa que inserte 30 letras aleatorias en un objeto `List<Character>`. Realice las siguientes operaciones y muestre sus resultados:

a) Ordene el objeto `List` en forma ascendente.
b) Ordene el objeto `List` en forma descendente.
c) Muestre el objeto `List` en forma ascendente con los duplicados eliminados.

17.13 ***(Asignar y luego reducir un `IntStream` para paralelización)*** La lambda que pase al método `reduce` de un flujo debe ser *asociativa*; es decir, sin importar el orden en el que se evalúen sus subexpresiones, el resultado debe ser el mismo. La expresión lambda en las líneas 34 a 36 de la figura 17.5 *no* es asociativa. Si fuera a usar flujos paralelos (capítulo 23, Concurrency, en inglés) con esa lambda, podría obtener resultados incorrectos para la suma de los cuadrados, dependiendo del orden en el que se evalúen las subexpresiones. La manera apropiada de implementar las líneas 34 a 36 sería *primero* asignar cada valor `int` al cuadrado de ese valor y *luego* reducir el flujo a la suma de los cuadrados. Modifique la figura 17.5 para implementar las líneas 34 a 36 de esta forma.