Java 8, 9, 10 y 11: ¿Qué novedades hay?

Desde el lanzamiento del Kit de Desarrollo de Java (JDK 1.0) en 1996, Java ha ganado una gran popularidad entre estudiantes, gestores de proyectos y programadores que lo utilizan activamente. Es un lenguaje expresivo que se sigue utilizando en proyectos de todos los tamaños. Su evolución (mediante la incorporación de nuevas funcionalidades) desde Java 1.1 (1997) hasta Java 7 (2011) se ha gestionado de forma eficaz. Java 8 se lanzó en marzo de 2014, Java 9 en septiembre de 2017, Java 10 en marzo de 2018 y Java 11 está previsto para septiembre de 2018. La pregunta es: ¿Por qué deberían importarte estos cambios?

1. ¿Qué novedades hay?

1.1. ¿Cuál es la principal novedad?

Argumentamos que los cambios en Java 8 fueron, en muchos sentidos, más profundos que cualquier otro cambio en la historia de Java (Java 9 añade cambios importantes, pero menos profundos, en la productividad, como verá más adelante en este capítulo, mientras que Java 10 realiza ajustes mucho menores a la inferencia de tipos). La buena noticia es que los cambios permiten escribir programas con mayor facilidad. Por ejemplo, en lugar de escribir código extenso (para ordenar una lista de manzanas en el inventario según su peso), como:

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

En Java 8, puedes escribir código más conciso que se ajusta mucho mejor al enunciado del problema, como el siguiente:

```
inventory.sort(comparing(Apple::getWeight));
```

¡El primer código Java 8 del libro!

Se lee como "ordenar inventario comparando el peso de las manzanas". No te preocupes por este código por ahora. Este libro explicará qué hace y cómo puedes escribir código similar.

También hay una influencia del hardware: las CPU comerciales se han vuelto multinúcleo; el procesador de tu portátil o computadora de escritorio probablemente tenga cuatro o más núcleos. Pero la gran mayoría de los programas Java existentes usan solo uno de estos núcleos y dejan los otros tres inactivos (o dedican una pequeña fracción de su potencia de procesamiento a ejecutar parte del sistema operativo o un antivirus).

Antes de Java 8, los expertos podrían decirte que tenías que usar hilos para usar estos núcleos. El problema es que trabajar con hilos es difícil y propenso a errores. Java ha seguido una trayectoria evolutiva en la que se ha intentado continuamente hacer que la concurrencia sea más fácil y menos propensa a errores. Java 1.0 incluía hilos, bloqueos e incluso un modelo de memoria —la mejor práctica en aquel momento—, pero estas primitivas resultaron demasiado difíciles de usar de forma fiable en equipos de proyecto no especializados. Java 5 añadió componentes robustos como grupos de hilos y colecciones concurrentes. Java 7 incorporó el marco de trabajo `fork/join`, lo que hizo que el paralelismo fuera más práctico, aunque seguía siendo complejo. Java 8 nos ofreció una nueva forma más sencilla de concebir el paralelismo. Sin embargo, aún es necesario seguir ciertas reglas, que aprenderá en este libro.

Como verá más adelante, Java 9 añade un método de estructuración adicional para la concurrencia: la programación reactiva. Si bien su uso es más especializado, estandariza una forma de aprovechar las herramientas de flujos reactivos RxJava y Akka, que se están popularizando en sistemas altamente concurrentes.

De los dos objetivos anteriores (código más conciso y uso más sencillo de procesadores multinúcleo) surge toda la estructura coherente que ofrece Java 8. Empezamos ofreciéndoles una breve introducción a estas ideas (esperamos que les resulte interesante, pero lo suficientemente concisa como para resumirlas):

- La API Streams
- Técnicas para pasar código a métodos
- Métodos predeterminados en interfaces

Java 8 proporciona una nueva API (llamada Streams) que admite múltiples operaciones paralelas para procesar datos y se asemeja a la forma en que se piensa en los lenguajes de consulta de bases de datos: se expresa lo que se desea de forma general, y la implementación (en este caso, la biblioteca Streams) elige el mejor mecanismo de ejecución de bajo nivel. Como resultado, se evita la necesidad de escribir código que utilice operaciones *sincronizadas*, que no solo son muy propensas a errores, sino también más costosas de lo que se podría pensar en las CPU multinúcleo.[1].

[1] Las CPU multinúcleo tienen cachés separadas (memoria rápida) conectadas a cada núcleo del procesador. El bloqueo requiere que estos elementos estén sincronizados, lo que exige una comunicación entre núcleos relativamente lenta mediante el protocolo de coherencia de caché

Desde una perspectiva ligeramente revisionista, la incorporación de `Streams` en Java 8 puede considerarse una causa directa de las otras dos incorporaciones a Java 8: *técnicas concisas para pasar código a métodos* (referencias a métodos, expresiones lambda) y *métodos predeterminados* en interfaces.

Pero considerar el paso de código a métodos como una mera consecuencia de Streams subestima su versatilidad en Java 8. Streams ofrece una nueva forma concisa de expresar la *parametrización del comportamiento*. Supongamos que queremos escribir dos métodos que se diferencian en solo unas pocas líneas de código. Ahora podemos pasar simplemente el código de las partes que difieren como argumento (esta técnica de programación es más corta, clara y menos propensa a errores que la tendencia común a usar copiar y pegar). Los expertos señalarán que, antes de Java 8, la parametrización del comportamiento podía codificarse mediante clases anónimas; pero dejaremos que el ejemplo del inicio de este capítulo, que muestra una mayor concisión del código con Java 8, hable por sí solo en términos de claridad.

La característica de Java 8 de pasar código a métodos (y poder devolverlo e incorporarlo a estructuras de datos) también proporciona acceso a una serie de técnicas adicionales que se conocen comúnmente como *programación funcional*. En resumen, este tipo de código, denominado *funciones* en la comunidad de programación funcional, se puede pasar y combinar para generar potentes patrones de programación que verá en Java a lo largo de este libro.

La parte central de este capítulo comienza con una explicación general de por qué evolucionan los lenguajes, continúa con secciones sobre las características principales de Java 8 y, a continuación, introduce las ideas de la programación funcional, cuyo uso se simplifica gracias a las nuevas características y que favorecen las nuevas arquitecturas informáticas. En esencia, la sección 1.2 analiza el proceso de evolución y los conceptos, de los que Java carecía anteriormente, para aprovechar el paralelismo multinúcleo de forma sencilla. La sección 1.3 explica por qué pasar código a métodos en Java 8 es un nuevo y potente patrón de programación, y la sección 1.4 hace lo mismo con Streams, la nueva forma de Java 8 de representar datos secuenciados e indicar si se pueden procesar en paralelo. La sección 1.5 explica cómo la nueva característica de Java 8, los métodos predeterminados, permite que las interfaces y sus bibliotecas evolucionen con menos complicaciones y menos recompilación. También explica la adición de *módulos* a Java 9, que permite especificar los componentes de grandes sistemas Java de forma más clara que con *un simple archivo JAR de paquetes*. Finalmente, la sección 1.6 analiza las ideas de la programación funcional en Java y otros lenguajes que comparten la JVM. En resumen, este capítulo introduce ideas que se desarrollan sucesivamente en el resto del libro. ¡Disfruta del viaje!

1.2. ¿Por qué Java sigue cambiando?

En la década de 1960 surgió la búsqueda del lenguaje de programación perfecto. Peter Landin, un famoso científico informático de su época, señaló en 1966 en un artículo fundamental[2], afirmó que ya existían 700 lenguajes de programación y especuló sobre cómo serían los siguientes 700, incluyendo argumentos a favor de la programación funcional, similar a la de Java 8. Tras miles de lenguajes de programación, los académicos han concluido que estos se comportan como ecosistemas: aparecen nuevos lenguajes y los antiguos son reemplazados a menos que evolucionen. Todos anhelamos un lenguaje universal perfecto, pero en realidad, ciertos lenguajes se adaptan mejor a nichos específicos. Por ejemplo, C y C++ siguen siendo populares para la creación de sistemas operativos y otros sistemas embebidos debido a su reducido tamaño y a pesar de su falta de seguridad en la programación. Esta falta de seguridad puede provocar fallos impredecibles en los programas y exponer vulnerabilidades a virus y otros programas maliciosos; de hecho, lenguajes con tipado seguro como Java y C# han reemplazado a C y C++ en diversas aplicaciones cuando el mayor tamaño de ejecución resulta aceptable. La ocupación previa de un nicho de mercado tiende a desalentar a la competencia. Cambiar a un nuevo lenguaje y conjunto de herramientas suele ser demasiado complicado para una sola funcionalidad, pero los nuevos lenguajes acabarán desplazando a los existentes, a menos que evolucionen con la suficiente rapidez para mantenerse al día. (Los lectores más veteranos suelen poder mencionar varios lenguajes en los que programaron anteriormente, pero cuya popularidad ha disminuido: Ada, Algol, COBOL, Pascal, Delphi y SNOBOL, por nombrar solo algunos). Eres programador de Java, y Java ha logrado colonizar (y desplazar a lenguajes de la competencia en) un amplio nicho de mercado de tareas de programación durante casi 20 años. Analicemos algunas de las razones.

1.2.1. El lugar de Java en el ecosistema de lenguajes de programación

Java tuvo un buen comienzo. Desde el principio, fue un lenguaje orientado a objetos bien diseñado con muchas bibliotecas útiles. Además, desde el primer día, admitió la concurrencia a pequeña escala gracias a su soporte integrado para hilos y bloqueos (y con su premonitorio reconocimiento, en forma de un modelo de memoria independiente del hardware, de que los hilos concurrentes en procesadores multinúcleo pueden tener comportamientos inesperados, además de los que ocurren en procesadores de un solo núcleo). Asimismo, la decisión de compilar Java a bytecode de la JVM (un código de máquina virtual que pronto fue compatible con todos los navegadores) significó que se convirtiera en el lenguaje preferido para los applets de internet (¿recuerdan los applets?).

De hecho, existe el riesgo de que la Máquina Virtual de Java (JVM) y su bytecode se consideren más importantes que el propio lenguaje Java y que,

[2] P. J. Landin, en *Los próximos 700 lenguajes de programación*, CACM 9(3):157–65, marzo de 1966

para ciertas aplicaciones, Java pueda ser reemplazado por alguno de sus lenguajes competidores, como Scala, Groovy o Kotlin, que también se ejecutan en la JVM. Varias actualizaciones recientes de la JVM (por ejemplo, el nuevo código de bytes invokedynamic en JDK7) buscan facilitar la ejecución fluida de lenguajes de la competencia en la JVM y la interoperabilidad con Java. Java también ha logrado integrarse con éxito en diversos ámbitos de la informática embebida (desde tarjetas inteligentes, tostadoras y decodificadores hasta sistemas de frenado de automóviles).

¿Cómo se consolidó Java como lenguaje de programación general?

La programación orientada a objetos se puso de moda en la década de 1990 por dos razones: su disciplina de encapsulación generó menos problemas de ingeniería de software que C; y, como modelo mental, se adaptó fácilmente al modelo de programación WIMP de Windows 95 y versiones posteriores. Esto se puede resumir de la siguiente manera: todo es un objeto; y un clic del ratón envía un mensaje de evento a un controlador (invoca el método `clicked` en un objeto `Mouse`). El modelo de Java de escribir una vez, ejecutar en cualquier lugar, y la capacidad de los primeros navegadores para ejecutar (de forma segura) applets de código Java le otorgaron un lugar destacado en las universidades, cuyos graduados posteriormente se incorporaron a la industria. Inicialmente hubo resistencia al mayor coste de ejecución de Java en comparación con C/C++, pero las máquinas se volvieron más rápidas y el tiempo del programador cobró cada vez más importancia. C# de Microsoft validó aún más el modelo orientado a objetos al estilo Java.

Sin embargo, el panorama está cambiando para el ecosistema de lenguajes de programación; los programadores trabajan cada vez más con los llamados macrodatos (conjuntos de datos de terabytes o más) y desean aprovechar eficazmente los ordenadores multinúcleo o los clústeres de computación para procesarlos. Esto implica el uso del procesamiento paralelo, algo que Java no facilitaba anteriormente. Es posible que hayas encontrado ideas en otros nichos de programación (por ejemplo, MapReduce de Google o la relativa facilidad de manipulación de datos mediante lenguajes de consulta de bases de datos como SQL) que te ayudan a trabajar con grandes volúmenes de datos y CPU multinúcleo. La Figura 1.1 resume gráficamente el ecosistema de lenguajes: imagina el paisaje como el espacio de problemas de programación y la vegetación dominante en un área específica como el lenguaje preferido para ese programa. El cambio climático se basa en la idea de que el nuevo hardware o las nuevas tendencias de programación (por ejemplo, *¿Por qué no puedo programar con un estilo similar a SQL?*) hacen que otros lenguajes se conviertan en la opción preferida para nuevos proyectos, del mismo modo que el aumento de las temperaturas regionales permite que las uvas crezcan mejor en latitudes más altas. Sin embargo, existe cierta histéresis: muchos agricultores tradicionales seguirán cultivando sus cosechas habituales. En resumen, están surgiendo nuevos lenguajes que se popularizan cada vez más porque se han adaptado rápidamente al cambio climático.

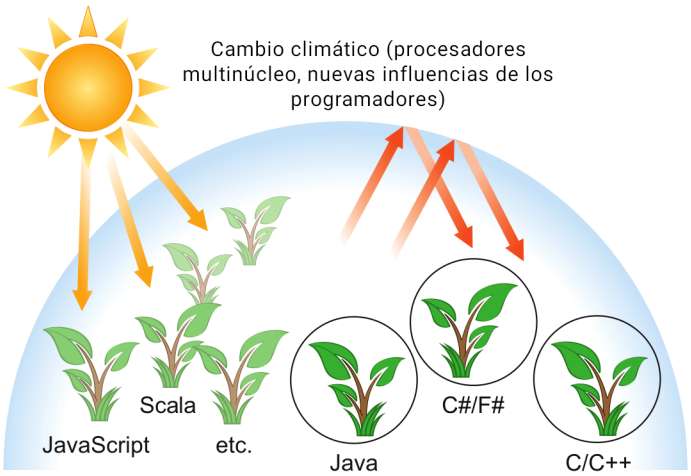


Figura 1.1 Ecosistema de lenguajes de programación y cambio climático

La principal ventaja de las novedades de Java 8 para un programador es que proporcionan más herramientas y conceptos de programación para resolver problemas de programación, tanto nuevos como existentes, con mayor rapidez o, lo que es más importante, de una manera más concisa y fácil de mantener. Aunque estos conceptos son nuevos en Java, han demostrado su eficacia en lenguajes especializados como los de investigación. En las siguientes secciones, destacaremos y desarrollaremos las ideas detrás de tres de estos conceptos de programación que han impulsado el desarrollo de las características de Java 8 para aprovechar el paralelismo y escribir código más conciso en general. Los presentaremos en un orden ligeramente diferente al del resto del libro para permitir una analogía con Unix y exponer las dependencias del tipo *esto es necesario por aquello* en el nuevo paralelismo multinúcleo de Java 8.

Otro factor de cambio climático para Java

Un factor de cambio climático tiene que ver con la forma en que se diseñan los sistemas grandes. Hoy en día, es común que un sistema grande incorpore subsistemas de componentes de otras fuentes, y es posible que estos se construyan sobre otros componentes de otros proveedores. Peor aún, estos componentes y sus interfaces también tienden a evolucionar. Java 8 y Java 9 abordaron estos aspectos al proporcionar métodos y módulos predeterminados para facilitar este estilo de diseño.

Las siguientes tres secciones examinan los tres conceptos de programación que impulsaron el diseño de Java 8.

1.2.2. Procesamiento de flujos / *Stream processing*

El primer concepto de programación es el *procesamiento de flujos*. A modo de introducción, un flujo/*stream* es una secuencia de elementos de datos que se generan conceptualmente uno a uno. Un programa puede leer elementos de un flujo de entrada uno por uno y, de forma similar, escribir elementos en

un flujo de salida. El flujo de salida de un programa bien podría ser el flujo de entrada de otro.

Un ejemplo práctico se encuentra en Unix o Linux, donde muchos programas operan leyendo datos de la entrada estándar (`stdin` en Unix y C, `System.in` en Java), procesándolos y luego escribiendo sus resultados en la salida estándar (`stdout` en Unix y C, `System.out` en Java). Primero, un poco de contexto: `cat` de Unix crea un flujo concatenando dos archivos, `tr` traduce los caracteres de un flujo, `sort` ordena las líneas de un flujo y `tail -3` devuelve las últimas tres líneas de un flujo. La línea de comandos de Unix permite enlazar programas mediante tuberías (`|`), dando ejemplos como:

```
cat file1 file2  |  tr "[A-Z]"  "[a-z]"  |  sort  |  tail -3
```

(suponiendo que `file1` y `file2` contienen una sola palabra por línea) imprime las tres palabras de los archivos que aparecen más tarde en el orden del diccionario, tras convertirlas a minúsculas. Decimos que `sort` toma como entrada una secuencia de líneas[3] y produce otra secuencia de líneas como salida (esta última ordenada), como se ilustra en la figura 1.2. Cabe destacar que en Unix estos comandos (`cat`, `tr`, `sort` y `tail`) se ejecutan simultáneamente, por lo que `sort` puede estar procesando las primeras líneas antes de que `cat` o `tr` hayan terminado. Una analogía más mecánica es una cadena de montaje de automóviles donde una serie de coches se coloca en cola entre estaciones de procesamiento. Cada estación toma un coche, lo modifica y lo pasa a la siguiente para su posterior procesamiento; el procesamiento en estaciones separadas suele ser concurrente, aunque la cadena de montaje sea físicamente una secuencia.

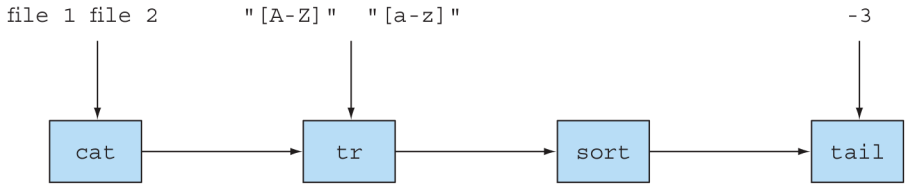


Figura 1.2 Comandos de Unix que operan sobre flujos de datos

Java 8 incorpora la API Streams (nótese la S mayúscula) en `java.util.stream`, basada en esta idea; `Stream<T>` es una secuencia de elementos de tipo `T`. Por ahora, se puede considerar como un iterador avanzado. La API Streams cuenta con numerosos métodos que se pueden encadenar para formar una canalización compleja, al igual que se encadenaban los comandos de Unix en el ejemplo anterior.

La principal ventaja es que ahora se puede programar en Java 8 con un mayor nivel de abstracción, estructurando la idea de convertir una secuencia de un tipo en otra (de forma similar a como se piensa al escribir consultas de base de datos), en lugar de procesar un elemento a la vez. Otra ventaja es que

[3] Los puristas dirán «*secuencia de caracteres*», pero conceptualmente es más sencillo pensar que `sort` reordena las *líneas*

Java 8 puede ejecutar de forma transparente la canalización de operaciones `Stream` en varios núcleos de CPU, procesando partes disjuntas de la entrada. Esto proporciona paralelismo *prácticamente gratuito*, en lugar del esfuerzo que supone usar *hilos* / `Threads`. Analizamos la API Streams de Java 8 en detalle en los capítulos 4 a 7.

1.2.3. Pasar código a métodos con parametrización de comportamiento

El segundo concepto de programación añadido a Java 8 es la capacidad de pasar un fragmento de código a una API. Esto puede sonar bastante abstracto. En el ejemplo de Unix, podríamos querer indicarle al comando `sort` que utilice un orden personalizado. Si bien el comando `sort` admite parámetros de línea de comandos para realizar varios tipos de ordenación predefinidos, como el orden inverso, estos son limitados.

Por ejemplo, supongamos que tenemos una colección de identificadores de factura con un formato similar a 2013UK0001, 2014US0002, etc. Los primeros cuatro dígitos representan el año, las dos letras siguientes un código de país y los últimos cuatro dígitos el ID de un cliente. Podríamos querer ordenar estos identificadores de factura por año, o quizás utilizando el ID del cliente o incluso el código de país.

Lo que necesitamos es la capacidad de indicarle al comando `sort` que acepte como argumento un orden definido por el usuario: un fragmento de código independiente que se pasa al comando `sort`.

Ahora, como ejemplo directo en Java, queremos indicarle a un método `sort` que compare utilizando un orden personalizado. Podríamos escribir un método `compareUsingCustomerId` para comparar dos ID de factura, pero, antes de Java 8, ¡no se podía pasar este método a otro! Podríamos crear un objeto `Comparator` para pasarlo al método `sort`, como mostramos al inicio de este capítulo, pero esto es engorroso y dificulta la idea de simplemente reutilizar un comportamiento existente. Java 8 añade la capacidad de pasar métodos (nuestro código) como argumentos a otros métodos. La figura 1.3, basada en la figura 1.2, ilustra esta idea. También nos referimos a esto conceptualmente como *parametrización de comportamiento*. ¿Por qué es importante? La API de Streams se basa en la idea de pasar código para parametrizar el comportamiento de sus operaciones, al igual que pasamos `compareUsingCustomerId` para parametrizar el comportamiento de la ordenación.

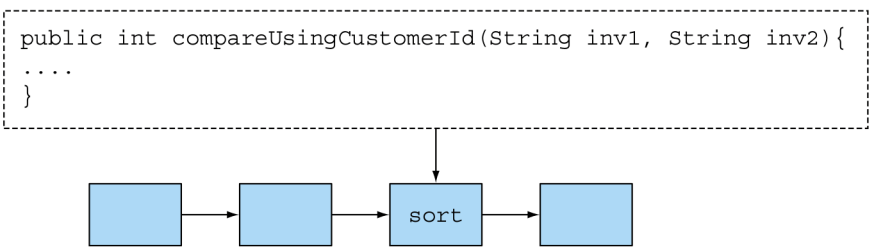


Figura 1.3 Pasar el método `compareUsingCustomerId` como argumento para `sort`

Resumimos su funcionamiento en la sección 1.3 de este capítulo, pero los detalles completos se abordan en los capítulos 2 y 3. Los capítulos 18 y 19 exploran funciones más avanzadas que se pueden realizar con esta característica, utilizando técnicas de la comunidad de *programación funcional*.

1.2.4. Paralelismo y datos mutables compartidos

El tercer concepto de programación es bastante más implícito y surge de la frase *paralelismo casi gratuito* de nuestra discusión anterior sobre el procesamiento de flujos. ¿Qué hay que sacrificar? Es posible que deba realizar algunos pequeños cambios en la forma en que codifica el comportamiento que se pasa a los métodos de flujo. Al principio, estos cambios pueden resultar un poco incómodos, pero una vez que se acostumbre a ellos, le encantarán. Debe proporcionar un comportamiento que sea *seguro para ejecutarse* concurrentemente en diferentes partes de la entrada. Por lo general, esto significa escribir código que no acceda a datos mutables compartidos para realizar su función. A veces se las denomina funciones puras, funciones sin efectos secundarios o funciones sin estado, y las analizaremos en detalle en los capítulos 18 y 19. El paralelismo anterior surge solo al asumir que múltiples copias de su fragmento de código pueden funcionar de forma independiente. Si hay una variable u objeto compartido al que se le escribe, entonces las cosas ya no funcionan. ¿Qué sucede si dos procesos quieren modificar la variable compartida al mismo tiempo? (La sección 1.4 ofrece una explicación más detallada con un diagrama). Encontrará más información sobre este estilo a lo largo del libro.

Los flujos de Java 8 aprovechan el paralelismo con mayor facilidad que la API Threads de Java, por lo que, si bien es *posible* usar `synchronized` para romper la regla de no compartir datos mutables, esto contradice el sistema al abusar de una abstracción optimizada para dicha regla. Usar `synchronized` en múltiples núcleos de procesamiento suele ser mucho más costoso de lo esperado, ya que la sincronización obliga a que el código se ejecute secuencialmente, lo cual va en contra del objetivo del paralelismo.

Dos de estos puntos (la ausencia de datos mutables compartidos y la capacidad de pasar métodos y funciones —código— a otros métodos) son los pilares de lo que generalmente se describe como el paradigma de la *programación funcional*, que se verá en detalle en los capítulos 18 y 19. En contraste, en el paradigma de la *programación imperativa*, un programa se describe normalmente en términos de una secuencia de instrucciones que modifican el estado. El requisito de no compartir datos mutables implica que un método se describe perfectamente solo por la forma en que transforma los argumentos en resultados; en otras palabras, se comporta como una función matemática y no tiene efectos secundarios (visibles).

1.2.5. Java necesita evolucionar

Ya has visto la evolución de Java. Por ejemplo, la introducción de los genéricos y el uso de `List<String>` en lugar de solo `List` puede haber resultado molesto al principio. Pero ahora estás familiarizado con este estilo y las ventajas que aporta (detección de más errores en tiempo de compilación y código más legible, ya que ahora sabes de qué es una lista).

Otros cambios han simplificado la expresión de operaciones comunes (por ejemplo, el uso de un bucle `for-each` en lugar de exponer el uso repetitivo de un `Iterator`). Los principales cambios en Java 8 reflejan un alejamiento de la programación orientada a objetos clásica, que a menudo se centra en la modificación de valores existentes, y una aproximación al espectro de la programación funcional, donde lo *que* se desea hacer en términos generales (por ejemplo, *crear un valor* que represente todas las rutas de transporte de A a B por menos de un precio determinado) se considera primordial y se separa de *cómo* lograrlo (por ejemplo, recorrer/*scan* una estructura de datos *modificando* ciertos componentes). Cabe destacar que la programación orientada a objetos clásica y la programación funcional, como extremos, podrían parecer contradictorias. Sin embargo, la idea es aprovechar lo mejor de ambos paradigmas de programación para tener mayores probabilidades de contar con la herramienta adecuada para cada tarea. Analizamos esto en detalle en las secciones 1.3 y 1.4.

Una conclusión clave podría ser la siguiente: los lenguajes deben evolucionar para adaptarse a los cambios en el hardware o a las expectativas de los programadores (si necesita pruebas, considere que COBOL fue en su momento uno de los lenguajes más importantes comercialmente). Para perdurar, Java debe evolucionar incorporando nuevas funcionalidades. Esta evolución será inútil si no se utilizan las nuevas características, así que al usar Java 8, proteges tu estilo de vida como programador Java. Además, tenemos la sensación de que te encantarán las nuevas características de Java 8. ¡Pregúntale a cualquiera que haya usado Java 8 si estaría dispuesto a volver atrás! Asimismo, las nuevas características de Java 8 podrían, siguiendo la analogía del ecosistema, permitir que Java conquiste el territorio de tareas de programación actualmente ocupado por otros lenguajes, por lo que los programadores de Java 8 tendrán aún más demanda.

Ahora presentamos los nuevos conceptos de Java 8, uno por uno, señalando los capítulos que los abordan con mayor detalle.

1.3. Funciones en Java

En los lenguajes de programación, la palabra *función* se usa comúnmente como sinónimo de *método*, especialmente de método estático; además, también se utiliza para referirse a *funciones matemáticas*, es decir, funciones sin efectos secundarios. Afortunadamente, como veremos, cuando Java 8 se

refiere a funciones, estos usos coinciden casi por completo.

Java 8 añade funciones como nuevas formas de representar valores. Estas facilitan el uso de flujos, que se tratan en la sección 1.4, y que Java 8 proporciona para aprovechar la programación paralela en procesadores multinúcleo. Comenzaremos mostrando que las funciones como valores son útiles en sí mismas.

Pensemos en los posibles valores que pueden manipular los programas Java. En primer lugar, existen valores primitivos como 42 (de tipo `int`) y 3.14 (de tipo `double`). En segundo lugar, los valores pueden ser objetos (más precisamente, referencias a objetos). La única forma de obtener uno de estos es mediante el método `new`, quizás a través de un método de fábrica o una función de biblioteca; las referencias a objetos apuntan a instancias de una clase. Algunos ejemplos son *abc* (de tipo `String`), `new Integer(1111)` (de tipo `Integer`) y el resultado `new HashMap<Integer, String>(100)` al llamar explícitamente a un constructor para `HashMap`. Incluso los arrays son objetos. ¿Cuál es el problema?

Para ayudar a responder a esto, cabe señalar que el objetivo principal de un lenguaje de programación es manipular valores, los cuales, siguiendo la tradición histórica de los lenguajes de programación, se denominan valores de primera clase (o ciudadanos, en la terminología tomada del movimiento por los derechos civiles de la década de 1960 en Estados Unidos). Otras estructuras en nuestros lenguajes de programación, que quizás nos ayudan a expresar la estructura de los valores pero que no se pueden pasar durante la ejecución del programa, son ciudadanos de segunda clase. Los valores, como se mencionó anteriormente, son ciudadanos de primera clase en Java, pero otros conceptos de Java, como los métodos y las clases, son ejemplos de ciudadanos de segunda clase. Los métodos son adecuados cuando se utilizan para definir clases, que a su vez pueden instanciarse para producir valores, pero no son valores en sí mismos. ¿Importa esto? Sí, resulta que poder pasar métodos en tiempo de ejecución, y por lo tanto convertirlos en elementos de primera clase, es útil en la programación, por lo que los diseñadores de Java 8 añadieron la capacidad de expresar esto directamente en Java. De paso, cabe preguntarse si convertir otros elementos de segunda clase, como las clases, en valores de primera clase también sería una buena idea. Varios lenguajes, como Smalltalk y JavaScript, han explorado esta posibilidad.

1.3.1. Métodos y expresiones lambda como elementos de primera clase

Experimentos en otros lenguajes, como Scala y Groovy, han demostrado que permitir que conceptos como los métodos se utilicen como valores de primera clase facilita la programación al ampliar el conjunto de herramientas disponibles para los programadores. Y una vez que los programadores se familiarizan con una característica potente, ¡se resisten a usar lenguajes que no la incluyan! Los diseñadores de Java 8 decidieron permitir que los métodos fue-

ran valores para facilitar la programación. Además, la característica de Java 8 de que los métodos sean valores constituye la base de otras características de Java 8 (como Streams).

La primera novedad de Java 8 que presentamos son las *referencias a métodos*. Supongamos que queremos filtrar todos los archivos ocultos de un directorio. Necesitamos escribir un método que, dado un objeto `File`, nos indique si está oculto. Afortunadamente, la clase `File` incluye un método llamado `isHidden`. Este método se puede considerar una función que recibe un objeto `File` y devuelve un valor `boolean`. Sin embargo, para usarlo en el filtrado, debemos encapsularlo en un objeto `FileFilter` que luego pasamos al método `File.listFiles`, como se muestra a continuación:

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
    public boolean accept(File file) {
        return file.isHidden();          <-- Filtrando archivos ocultos!
    }
});
```

¡Qué horror! Aunque solo son tres líneas importantes, son tres líneas opacas; todos recordamos haber dicho *¿De verdad tengo que hacerlo así?* la primera vez que lo vimos. Ya tienes el método `isHidden` que podrías usar. ¿Por qué tienes que encapsularlo en una clase *FileFilter* tan larga y luego instanciarla? Porque así era como se hacía antes de Java 8.

Ahora, puedes reescribir ese código de la siguiente manera:

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

¡Guau! ¿No es genial? Ya tienes la función `isHidden` disponible, así que la pasas al método `listFiles` usando la sintaxis de *referencia de método* `::` de Java 8 (que significa *usa este método como un valor*); fíjate que también hemos empezado a usar la palabra *función* para referirnos a los métodos. Explicaremos más adelante cómo funciona. Una ventaja es que tu código ahora se parece más al enunciado del problema.

Aquí tienes un adelanto de lo que viene: los métodos ya no son valores de segunda clase. De forma análoga a como se usa una *referencia a un objeto* al pasarlo como argumento (y las referencias a objetos se crean con `new`), en Java 8, al escribir `File::isHidden`, se crea una *referencia a un método*, que también se puede pasar como argumento. Este concepto se analiza en detalle en el capítulo 3. Dado que los métodos contienen código (el cuerpo ejecutable de un método), el uso de referencias a métodos permite pasar código como se muestra en la figura 1.3. La figura 1.4 ilustra el concepto. También se verá un ejemplo concreto (seleccionar manzanas de un inventario) en la siguiente sección.

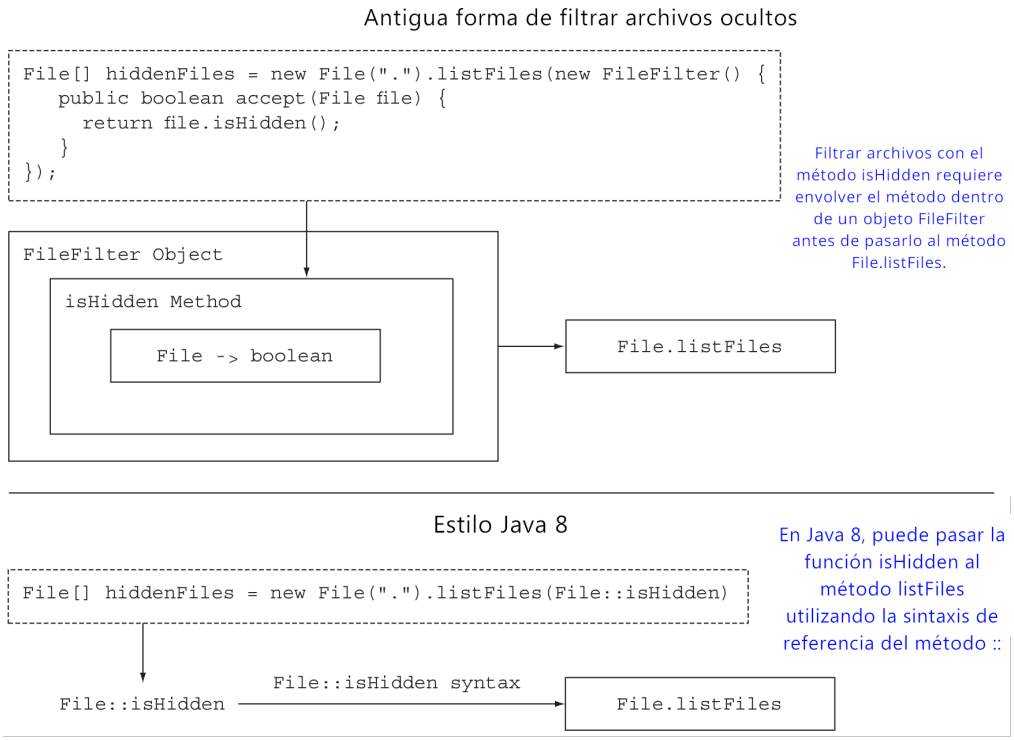


Figura 1.4 Pasar la referencia del método `File::isHidden` al método `listFiles`

LAMBDAS: FUNCIONES ANÓNIMAS

Además de permitir que los métodos (con nombre) sean valores de primera clase, Java 8 permite una concepción más completa de las *funciones como valores*, incluyendo las expresiones *lambda*[4] (o funciones anónimas). Por ejemplo, ahora puedes escribir `(int x) ->x + 1` para indicar «*la función que, al ser llamada con el argumento x, devuelve el valor x + 1*». Quizás te preguntes por qué es necesario, ya que podrías definir un método *add1* dentro de una clase `MyMathsUtils` y luego escribir `MyMathsUtils::add1`. Sí, podrías, pero la nueva sintaxis lambda es más concisa para los casos en los que no dispones de un método y una clase convenientes. El capítulo 3 explora las expresiones lambda en detalle. Se dice que los programas que utilizan estos conceptos están escritos en estilo de programación funcional; esta expresión significa «*escribir programas que pasan funciones como valores de primera clase*».

1.3.2. Paso de código: un ejemplo

Veamos un ejemplo de cómo esto te ayuda a escribir programas (se analiza con más detalle en el capítulo 2). Todo el código de los ejemplos está disponible en un repositorio de GitHub y se puede descargar desde el sitio web del libro. Ambos enlaces se pueden encontrar en www.manning.com/books/modern-java-in-action. Supongamos que tiene una clase `Apple` con un método `getColor` y una variable `inventory` que contiene una lista de `Apples`; entonces podría querer seleccionar todas las manzanas verdes (aquí usando

[4] Originalmente llamado así por la letra griega λ (*lambda*). Aunque el símbolo no se utiliza en Java, su nombre sigue vivo.

un tipo enumerado `Color` que incluye los valores `GREEN` y `RED`) y devolverlas en una lista. La palabra *filter* se usa comúnmente para expresar este concepto. Antes de Java 8, podría escribir un método `filterGreenApples`:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (GREEN.equals(apple.getColor())) {
            result.add(apple);
        }
    }
    return result;
}
```

La lista de resultados acumula el resultado; comienza vacío y luego se agregan manzanas verdes una por una.

El texto resaltado selecciona sólo manzanas verdes.

Pero a continuación, alguien querría la lista de manzanas pesadas (digamos, de más de 150 g), así que, con pesar, escribirías el siguiente método para conseguirlo (quizás incluso usando copiar y pegar):

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

Aquí el texto resaltado selecciona sólo manzanas pesadas.

Todos conocemos los peligros de copiar y pegar en el desarrollo de software (actualizaciones y correcciones de errores en una variante pero no en la otra), y resulta que estos dos métodos solo difieren en una línea: la condición resaltada dentro de la estructura `if`. Si la diferencia entre las dos llamadas al método en el código resaltado hubiera sido el rango de peso aceptable, podrías haber pasado los pesos mínimo y máximo aceptables como argumentos al `filter`; por ejemplo, (150, 1000) para seleccionar manzanas pesadas (de más de 150 g) o (`0, 80`) para seleccionar manzanas ligeras (de menos de 80 g).

Pero como ya mencionamos, Java 8 permite pasar el código de la condición como argumento, evitando así la duplicación del código del método `filter`. Ahora puedes escribir esto:

```
public static boolean isGreenApple(Apple apple) {
    return GREEN.equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
    return apple.getWeight() > 150;
}
public interface Predicate<T>{
    boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

Incluido para mayor claridad (normalmente importado de java.util.function)

Se pasa un método como un parámetro de predicado llamado p (consulte la barra lateral "¿Qué es un predicado?")

¿La manzana coincide con la condición representada por p?

Para usarlo, llama a:

```
filterApples(inventory, Apple::isGreenApple);
```

o

```
filterApples(inventory, Apple::isHeavyApple);
```

Explicaremos su funcionamiento en detalle en los próximos dos capítulos. La idea clave por ahora es que puedes pasar un método como argumento en Java 8.

¿Qué es un predicado?

El código anterior pasó el método `Apple::isGreenApple` (que recibe una manzana como argumento y devuelve un `boolean`) a *filterApples*, que esperaba un parámetro de tipo `Predicate<Apple>`. El término *predicado* se usa a menudo en matemáticas para referirse a una función que recibe un valor como argumento y devuelve `true` o `false`. Como verás más adelante, Java 8 también permite escribir `Function<Apple, Boolean>` —más familiar para quienes aprendieron sobre funciones pero no sobre predicados—, pero usar `Predicate<Apple>` es más estándar (y ligeramente más eficiente, ya que evita convertir un `boolean` en otro `Boolean`).

1.3.3. De pasar métodos a expresiones lambda

Pasar métodos como valores es claramente útil, pero resulta molesto tener que escribir una definición para métodos cortos como `isHeavyApple` e `isGreenApple` cuando se usan quizás solo una o dos veces. Pero Java 8 también ha solucionado esto. Introduce una nueva notación (funciones anónimas o expresiones lambda) que permite escribir simplemente:

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()) );
```

o

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

o incluso

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
                                     RED.equals(a.getColor()) );
```

Ni siquiera necesitas escribir la definición de un método que se use solo una vez; el código es más conciso y claro porque no necesitas buscar el código que estás pasando.

Pero si dicha expresión lambda ocupa más de unas pocas líneas (de modo que su comportamiento no sea inmediatamente claro), deberías usar una referencia a un método con un nombre descriptivo en lugar de una expresión lambda anónima. La claridad del código debe ser tu guía.

Los diseñadores de Java 8 casi podrían haberse detenido aquí, y quizás lo habrían hecho antes de las CPU multinúcleo. La programación funcional, tal como se ha presentado hasta ahora, resulta ser potente, como verás. Java podría haberse completado añadiendo `filter` y algunos métodos similares como métodos de biblioteca genéricos, como:

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
```

Ni siquiera tendrías que escribir métodos como `filterApples` porque, por ejemplo, la llamada anterior:

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

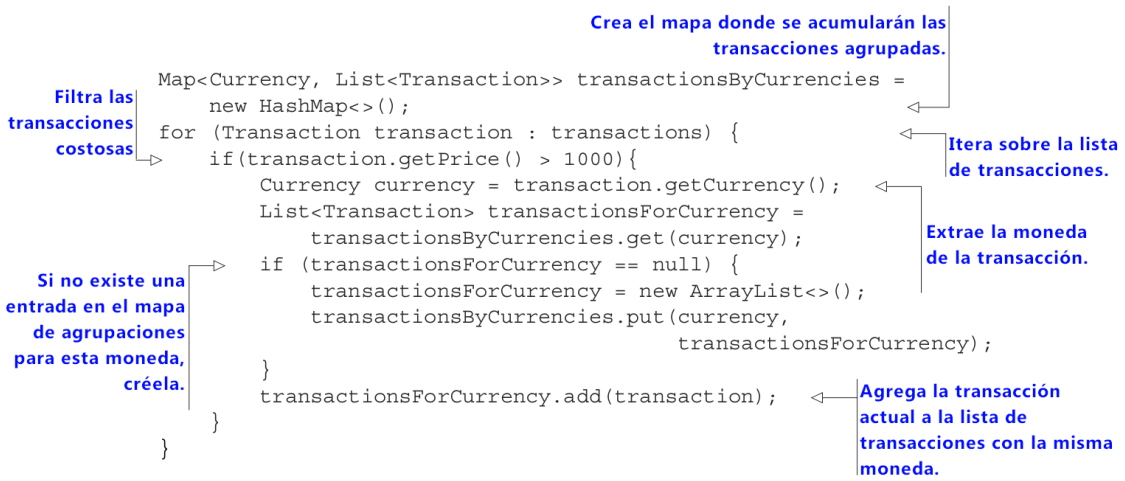
podría escribirse como una llamada al método de biblioteca *filter*:

```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

Pero, por razones centradas en aprovechar mejor el paralelismo, los diseñadores no lo hicieron. Java 8 incluye una nueva API similar a las colecciones llamada **Stream**, que contiene un conjunto completo de operaciones parecidas a la operación de `filter` con la que los programadores funcionales pueden estar familiarizados (por ejemplo, `map` y `reduce`), junto con métodos para convertir entre `colecciones` y `Streams`, que analizaremos a continuación.

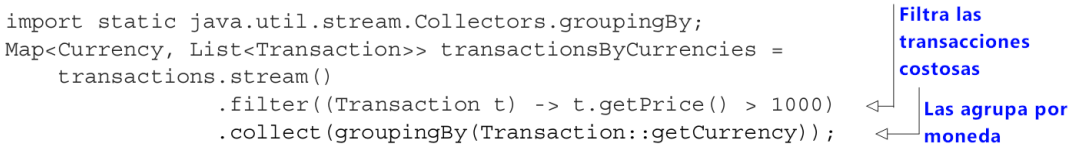
1.4. Streams

Casi todas las aplicaciones Java *crean/makes* y *procesan/process* colecciones. Sin embargo, trabajar con colecciones no siempre es lo ideal. Por ejemplo, supongamos que necesita filtrar las transacciones costosas de una lista y luego agruparlas por moneda. Tendría que escribir mucho código repetitivo para implementar esta consulta de procesamiento de datos, como se muestra aquí:



Además, es difícil comprender a simple vista qué hace el código debido a las múltiples sentencias de control de flujo anidadas.

Utilizando la API Streams, puede resolver este problema de la siguiente manera:



No te preocupes por este código por ahora, ya que puede parecer magia. Los capítulos 4 a 7 se dedican a explicar cómo entender la API de Streams. Por ahora, vale la pena destacar que la API de Streams ofrece una forma diferente de procesar datos en comparación con la API de Collections. Al usar una colección, gestionas tú mismo el proceso de iteración. Necesitas iterar sobre los elementos uno por uno usando un bucle `for-each`, procesándolos secuencialmente. A esta forma de iterar sobre los datos la llamamos *iteración externa*. En cambio, con la API de Streams, no necesitas pensar en términos de bucles. El procesamiento de datos se realiza internamente dentro de la biblioteca. A esta idea la llamamos *iteración interna*. Retomaremos estas ideas en el capítulo 4.

Como segundo inconveniente al trabajar con colecciones, piensa por un momento en cómo procesarías la lista de transacciones si tuvieras una gran cantidad de ellas; ¿cómo podrías procesar una lista tan enorme? Una sola CPU no podría procesar tal cantidad de datos, pero probablemente tengas una computadora multinúcleo en tu escritorio. Lo ideal sería distribuir la carga de trabajo entre los distintos núcleos de la CPU disponibles en tu equipo para reducir el tiempo de procesamiento. En teoría, si tienes ocho núcleos, deberían poder procesar tus datos ocho veces más rápido que con un solo núcleo, ya que trabajan en paralelo.[5]

[5]Esta nomenclatura es desafortunada en cierto modo. Cada núcleo de un chip multinúcleo es una CPU completa. Sin embargo, la expresión CPU multinúcleo se ha popularizado, por lo que el término *«núcleo»* se usa para referirse a las CPU individuales.

Ordenadores multinúcleo

Todos los ordenadores de sobremesa y portátiles modernos son multinúcleo. En lugar de una sola CPU, tienen cuatro, ocho o más CPU (normalmente llamadas núcleos 5). El problema es que un programa Java clásico utiliza solo uno de estos núcleos, y la potencia de los demás se desperdicia. Del mismo modo, muchas empresas utilizan *clústeres de computación* (ordenadores conectados entre sí mediante redes de alta velocidad) para procesar grandes cantidades de datos de forma eficiente. Java 8 facilita nuevos estilos de programación para aprovechar mejor estos ordenadores.

El motor de búsqueda de Google es un ejemplo de un código demasiado grande para ejecutarse en una sola computadora. Lee todas las páginas de internet y crea un índice, mapeando cada palabra que aparece en cualquier página web con cada URL que contiene esa palabra. Luego, cuando realizas una búsqueda en Google con varias palabras, el software puede usar rápidamente este índice para proporcionarte un conjunto de páginas web que contienen esas palabras. Intenta imaginar cómo podrías programar este algoritmo en Java (incluso para un índice más pequeño que el de Google, necesitarías aprovechar todos los núcleos de tu computadora).

1.4.1. La programación multihilo es difícil

El problema es que aprovechar el paralelismo escribiendo código *multihilo* (usando la API Threads de versiones anteriores de Java) es difícil. Hay que pensar de manera diferente: los hilos pueden acceder y actualizar variables compartidas al mismo tiempo. Como resultado, los datos podrían cambiar inesperadamente si no se coordinan[6] apropiadamente. Este modelo es más difícil de entender que un modelo secuencial paso a paso[7]. Por ejemplo, la figura 1.5 muestra un posible problema con dos hilos que intentan sumar un número a una variable compartida llamada `sum` si no están sincronizados correctamente.

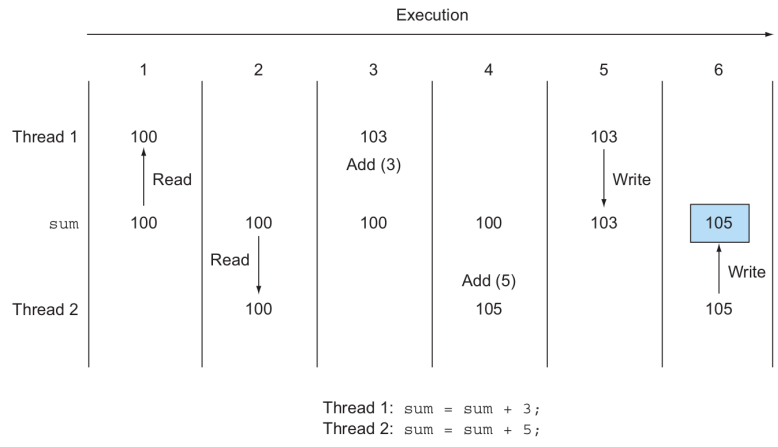


Figura 1.5. Un posible problema con dos hilos que intentan sumar a una variable de suma compartida. El resultado es 105 en lugar del resultado esperado de 108.

[6] Tradicionalmente se utilizaba la palabra clave `synchronized`, pero su uso incorrecto generaba numerosos errores sutiles. El paralelismo basado en `Streams` de Java 8 fomenta un estilo de programación funcional donde `synchronized` se usa con poca frecuencia; se centra en la partición de los datos en lugar de la coordinación del acceso a ellos.

[7] ¡Ajá! ¡Una fuente de presión para que el idioma evolucione!

Java 8 también aborda ambos problemas (código repetitivo y complejidad en el procesamiento de colecciones, así como la dificultad para aprovechar los procesadores multinúcleo) con la API Streams (`java.util.stream`). La primera razón de este diseño es que existen muchos patrones de procesamiento de datos (similares a `filterApples` en la sección anterior o a operaciones comunes en lenguajes de consulta de bases de datos como SQL) que se repiten constantemente y que se beneficiarían de formar parte de una biblioteca: filtrar/*filtering* datos según un criterio (por ejemplo, manzanas pesadas), *extraer* datos (por ejemplo, extraer el campo de peso de cada manzana en una lista) o *agrupar* datos (por ejemplo, agrupar una lista de números en listas separadas de números pares e impares), etc. La segunda razón es que estas operaciones a menudo se pueden paralelizar. Por ejemplo, como se ilustra en la figura 1.6, filtrar una lista en dos CPU podría hacerse asignando a una CPU el procesamiento de la primera mitad de la lista y a la segunda, el de la otra mitad. Esto se denomina *paso de bifurcación/forking step* (1). Las CPU filtran entonces sus respectivas sublistas (2). Finalmente (3), una CPU combina los dos resultados. (Esto guarda estrecha relación con la rapidez con la que funcionan las búsquedas de Google, utilizando muchos más de dos procesadores).

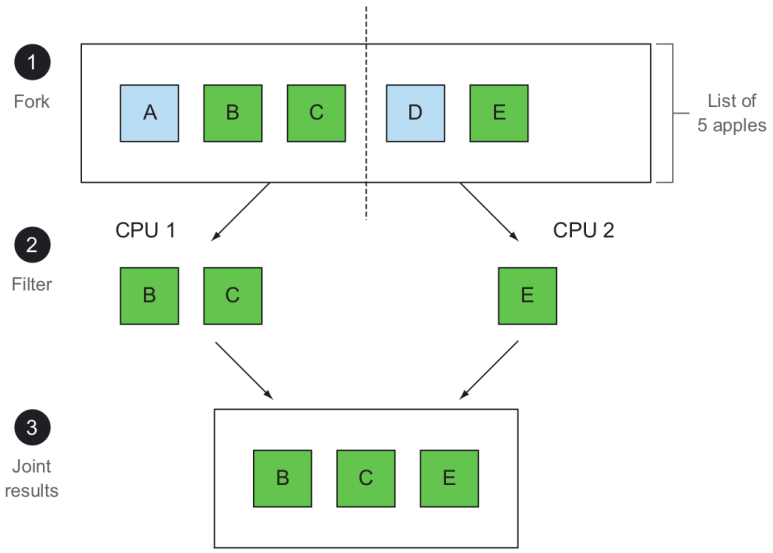


Figura 1.6 Bifurcación del filtro en dos CPU y unión del resultado

Por ahora, diremos que la nueva API Streams se comporta de forma similar a la API Collections de Java: ambas proporcionan acceso a secuencias de elementos de datos. Sin embargo, es útil recordar que Collections se centra principalmente en almacenar y acceder a datos, mientras que Streams se centra en describir cálculos sobre ellos. La clave reside en que la API Streams permite y fomenta el procesamiento en paralelo de los elementos de un flujo. Aunque pueda parecer extraño al principio, a menudo la forma más rápida de filtrar una colección (por ejemplo, para usar `filterApples` en la sección anterior sobre una lista) es convertirla en un flujo, procesarla en paralelo y luego volver a convertirla en una lista. Reiteramos que esto permite *un paralelismo casi gra-*

tuito y mostraremos cómo filtrar elementos de una lista de forma secuencial o en paralelo utilizando flujos y una expresión lambda.

Aquí tienes un ejemplo de procesamiento secuencial:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
```

Y aquí se muestra utilizando procesamiento paralelo:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

Paralelismo en Java y ausencia de estado mutable compartido

> Siempre se ha dicho que el paralelismo en Java es difícil y que todo esto de la `sincronización` es propenso a errores. ¿Dónde está la solución mágica en Java 8?
>
> Hay dos soluciones mágicas. Primero, la biblioteca se encarga de la partición, dividiendo un flujo grande en varios flujos más pequeños para procesarlos en paralelo. Segundo, este paralelismo, casi sin costo adicional para los flujos, funciona solo si los métodos pasados a los métodos de la biblioteca, como *filter*, no interactúan entre sí (por ejemplo, al tener objetos compartidos mutables). Pero resulta que esta restricción es natural para un programador (véase, por ejemplo, nuestro ejemplo `Apple::isGreenApple`). Si bien el significado principal de *funcional* en programación funcional es *usar funciones como valores de primera clase*, a menudo tiene un matiz secundario de *sin interacción durante la ejecución entre componentes*.

El capítulo 7 explora el procesamiento paralelo de datos en Java 8 y su rendimiento con más detalle. Uno de los problemas prácticos que encontraron los desarrolladores de Java 8 al evolucionar Java con todas estas novedades fue la evolución de las interfaces existentes. Por ejemplo, el método `Collections.sort` pertenece a la interfaz `List`, pero nunca se incluyó. Lo ideal sería usar `list.sort(comparator)` en vez de `Collections.sort(list, comparator)`. Esto puede parecer trivial, pero antes de Java 8, solo se podía actualizar una interfaz si se actualizaban todas las clases que la implementaban, ¡una pesadilla logística! Este problema se resuelve en Java 8 mediante *métodos predeterminados*.

1.5. Métodos predeterminados y módulos Java

Como mencionamos anteriormente, los sistemas modernos tienden a construirse a partir de componentes, quizás adquiridos de otras fuentes. Históricamente, Java ofrecía poco soporte para esto, aparte de un archivo JAR que contenía un conjunto de paquetes Java sin una estructura particular. Además, evolucionar las interfaces de dichos paquetes era difícil: cambiar una interfaz Java implicaba cambiar todas las clases que la implementaban. Java 8 y 9 han comenzado a abordar este problema.

En primer lugar, Java 9 proporciona un sistema de *módulos* que ofrece una sintaxis para definir módulos que contienen colecciones de paquetes, y permite un mayor control sobre la visibilidad y los espacios de nombres. Los módulos enriquecen un componente simple tipo JAR con estructura, tanto para la documentación del usuario como para la verificación automática; los explicamos en detalle en el capítulo 14. En segundo lugar, Java 8 añadió métodos predeterminados para admitir interfaces *evolutivas*. Los analizamos en detalle en el capítulo 13. Son importantes porque los encontrará cada vez con más frecuencia en las interfaces, pero dado que relativamente pocos programadores necesitarán escribir métodos predeterminados y que facilitan la evolución del programa en lugar de ayudar a escribir un programa en particular, mantenemos la explicación aquí breve y basada en ejemplos.

En la sección 1.4, proporcionamos el siguiente ejemplo de código Java 8:

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

Pero aquí hay un problema: una `List<T>` anterior a Java 8 no tiene métodos `stream` ni `parallelStream` —tampoco la interfaz `Collection<T>` que implementa— porque estos métodos no se habían concebido. Y sin estos métodos, este código no compilará. La solución más sencilla, que podrías emplear para tus propias interfaces, habría sido que los diseñadores de Java 8 añadieran el método `stream` a la interfaz `Collection` e incluyeran la implementación en la clase `ArrayList`.

Pero hacer esto habría sido una pesadilla para los usuarios. Muchos frameworks de colecciones alternativos implementan interfaces de la API Collections. Añadir un nuevo método a una interfaz implica que todas las clases concretas deben proporcionar una implementación para él. Los diseñadores del lenguaje no tienen control sobre las implementaciones existentes de `Collection`, por lo que surge un dilema: ¿Cómo se pueden desarrollar interfaces publicadas sin alterar las implementaciones existentes?

La solución de Java 8 consiste en romper el último vínculo: una interfaz ahora puede contener firmas de métodos para las que una clase implementadora no proporciona una implementación. Entonces, ¿quién las implementa? Los cuerpos de los métodos que faltan se proporcionan como parte de la interfaz (y, por lo tanto, como implementaciones predeterminadas) en lugar de en la clase que la implementa.

Esto permite al diseñador de interfaces ampliar una interfaz más allá de los métodos originalmente planificados, sin modificar el código existente. Java 8 permite usar la palabra clave `default` en las especificaciones de interfaz para lograr esto.

Por ejemplo, en Java 8, se puede llamar al método `sort` directamente en una lista. Esto es posible gracias al siguiente método predeterminado en la in-

terfaz `List` de Java 8, que llama al método estático `Collections.sort`:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Esto significa que las clases concretas de `List` no tienen que implementar explícitamente el método `sort`, mientras que en versiones anteriores de Java, dichas clases no se recompilaban a menos que proporcionaran una implementación para `sort`.

Pero un momento. Una sola clase puede implementar múltiples interfaces, ¿verdad? Si tienes múltiples implementaciones predeterminadas en varias interfaces, ¿significa eso que tienes una forma de herencia múltiple en Java? Sí, hasta cierto punto. En el capítulo 13 mostramos que existen algunas reglas que previenen problemas como el infame *problema de la herencia diamante* en C++.

1.6. Otras buenas ideas de la programación funcional

Las secciones anteriores presentaron dos ideas fundamentales de la programación funcional que ahora forman parte de Java: el uso de métodos y expresiones lambda como valores de primera clase, y la idea de que las llamadas a funciones o métodos pueden ejecutarse de forma eficiente y segura en paralelo en ausencia de un estado compartido mutable. Ambas ideas se aprovechan en la nueva API Streams que describimos anteriormente.

Los lenguajes funcionales comunes (SML, OCaml, Haskell) también proporcionan construcciones adicionales para ayudar a los programadores. Una de ellas es evitar los valores `null` mediante el uso explícito de tipos de datos más descriptivos. Tony Hoare, uno de los grandes de la informática, dijo lo siguiente en una presentación en QCon London 2009:

> *Lo llamo mi error de mil millones de dólares. Fue la invención de la referencia nula en 1965... No pude resistir la tentación de incluir una referencia nula, simplemente porque era muy fácil de implementar.*

Java 8 introdujo la clase `Optional<T>` que, si se usa de forma consistente, puede ayudar a evitar excepciones de puntero nulo. Es un objeto contenedor que puede o no contener un valor. `Optional<T>` incluye métodos para manejar explícitamente el caso en que un valor esté ausente, lo que permite evitar excepciones de puntero nulo. Utiliza el sistema de tipos para indicar cuándo se prevé que una variable pueda tener un valor faltante. Analizamos `Optional<T>` en detalle en el capítulo 11.

Una segunda idea es la de la *coincidencia de patrones* (*estructurales*).[8]

[8] Esta expresión tiene dos usos. Aquí nos referimos al uso común en matemáticas y programación funcional, donde una función se define mediante casos, en lugar de usar condicionales (`if-then-else`). El otro significado se refiere a expresiones como «*encontrar todos los archivos con el formato `IMG*.JPG` en un directorio determinado*», asociadas a las llamadas expresiones regulares.

Esto se utiliza en matemáticas. Por ejemplo:

```
f(0) = 1
f(n) = n*f(n-1) otherwise
```

En Java, se escribiría una sentencia `if-then-else` o `switch`. Otros lenguajes han demostrado que, para tipos de datos más complejos, la coincidencia de patrones puede expresar ideas de programación de forma más concisa que el uso de `if-then-else`. Para estos tipos de datos, también se podría usar el polimorfismo y la sobreescritura de métodos como alternativa a if-then-else, pero existe un debate continuo sobre el diseño del lenguaje respecto a cuál es más apropiado.[9] Consideramos que ambas son herramientas útiles y que conviene tenerlas a mano. Lamentablemente, Java 8 no ofrece soporte completo para la coincidencia de patrones, aunque mostramos cómo se puede expresar en el capítulo 19. También se está debatiendo una propuesta de mejora de Java para dar soporte a la coincidencia de patrones en una futura versión de Java (véase http://openjdk.java.net/jeps/305). Mientras tanto, ilustremos esto con un ejemplo expresado en el lenguaje de programación Scala (otro lenguaje similar a Java que utiliza la JVM y que ha inspirado algunos aspectos de la evolución de Java; véase el capítulo 20). Supongamos que queremos escribir un programa que realice simplificaciones básicas en un árbol que representa una expresión aritmética. Dado un tipo de dato `Expr` que representa dichas expresiones, en Scala podemos escribir el siguiente código para descomponer un Expr en sus partes y luego devolver otro `Expr`:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e
    case BinOp("-", e, Number(0)) => e
    case BinOp("*", e, Number(1)) => e
    case BinOp("/", e, Number(1)) => e
    case _ => expr
}
```

Adds 0

Subtracts 0

Multiplies by 1

Divides by 1

No se puede simplificar con estos casos, así que déjelos en paz.

Aquí, la sintaxis `expr match` de Scala se corresponde con la instrucción `switch (expr)` de Java. No te preocupes por este código por ahora; leerás más sobre la coincidencia de patrones en el capítulo 19. Por el momento, puedes considerar la coincidencia de patrones como una forma extendida de `switch` que puede descomponer un tipo de dato en sus componentes simultáneamente.

¿Por qué la instrucción `switch` en Java debería limitarse a valores primitivos y cadenas? Los lenguajes funcionales suelen permitir el uso de `switch` con muchos más tipos de datos, incluyendo la coincidencia de patrones (en el código Scala, esto se logra mediante una operación `match`). En el diseño orientado a objetos, el patrón `visitor` es un patrón común que se utiliza para recorrer una familia de clases (como los diferentes componentes de un coche: rueda, motor, chasis, etc.) y aplicar una operación a cada objeto visitado. Una ventaja de la coincidencia de patrones es que un compilador puede informar de errores comunes como: «*La clase `Brakes` forma parte de la familia de clases que*

[9] El artículo de **Wikipedia** sobre el "problema de la expresión" (término acuñado por Phil Wadler) sirve como punto de partida para este debate.

representan los componentes de la clase `Car`*. Olvidaste gestionarla explícitamente*».

Los capítulos 18 y 19 ofrecen una introducción completa a la programación funcional y a cómo escribir programas de estilo funcional en Java 8, incluyendo el conjunto de funciones que proporciona su biblioteca. El capítulo 20 analiza las diferencias entre las características de Java 8 y las de Scala, un lenguaje que, al igual que Java, se implementa sobre la JVM y que ha evolucionado rápidamente, amenazando algunos aspectos del nicho de Java en el ecosistema de lenguajes de programación. Este material se ubica hacia el final del libro para brindar información adicional sobre por qué se agregaron las nuevas características de Java 8 y Java 9.

Características de Java 8, 9, 10 y 11: ¿Por dónde empezar?

Java 8 y Java 9 aportaron actualizaciones significativas a Java. Sin embargo, como programador Java, es probable que las novedades de Java 8 sean las que más te afecten en tu trabajo diario de programación a pequeña escala: la idea de pasar un método o una expresión lambda se está convirtiendo rápidamente en un conocimiento esencial de Java. En cambio, las mejoras de Java 9 enriquecen nuestra capacidad para definir y utilizar componentes a mayor escala, ya sea estructurando un sistema mediante módulos o importando un kit de herramientas de programación reactiva. Finalmente, Java 10 representa un incremento mucho menor en comparación con las actualizaciones anteriores y consiste en permitir la inferencia de tipos para variables locales, tema que se aborda brevemente en el capítulo 21, donde también se menciona la sintaxis más completa para los argumentos de las expresiones lambda, que se introducirá en Java 11. En el momento de escribir este texto, el lanzamiento de Java 11 está previsto para septiembre de 2018. Java 11 también incluye una nueva biblioteca cliente HTTP asíncrona (http://openjdk.java.net/jeps/321) que aprovecha los avances de Java 8 y Java 9 (detalles en los capítulos 15, 16 y 17) en CompletableFuture y programación reactiva.

Resumen

- Tenga en cuenta el concepto de ecosistema de lenguajes y la consiguiente presión de evolución que sufren. Si bien Java goza de una excelente salud actualmente, podemos recordar otros lenguajes, como COBOL, que no lograron evolucionar.
- Las principales novedades de Java 8 ofrecen conceptos y funcionalidades novedosas que facilitan la escritura de programas eficaces y concisos.
- Los procesadores multinúcleo no se benefician plenamente de las prácticas de programación anteriores a Java 8.
- Las funciones son valores de primera clase; recuerde cómo se pueden pasar métodos como valores funcionales y cómo se escriben las funciones anónimas (lambdas).

- El concepto de flujos de Java 8 generaliza muchos aspectos de las colecciones, pero a menudo permite un código más legible y el procesamiento paralelo de elementos de un flujo.
- La programación basada en componentes a gran escala y la evolución de las interfaces de un sistema no se adaptaban bien históricamente a Java. Ahora, en Java 9, se pueden especificar módulos para estructurar sistemas y usar métodos predeterminados para mejorar una interfaz sin modificar todas las clases que la implementan.
- Otras ideas interesantes de la programación funcional incluyen el manejo de valores nulos y el uso de la coincidencia de patrones.