Funciones en Lambda

3.8 Métodos útiles para componer expresiones lambda

Varias interfaces funcionales de la API de Java 8 contienen métodos de conveniencia. En concreto, muchas interfaces funcionales, como `Comparator`, `Function` y `Predicate`, que se utilizan para pasar expresiones lambda, proporcionan métodos que permiten la composición. ¿Qué significa esto? En la práctica, significa que puedes combinar varias expresiones lambda sencillas para construir otras más complejas. Por ejemplo, puedes combinar dos predicados en un predicado más grande que realice una operación OR entre ellos. Además, también puedes componer funciones de forma que el resultado de una se convierta en la entrada de otra. Quizás te preguntes cómo es posible que existan métodos adicionales en una interfaz funcional. (¡Al fin y al cabo, esto contradice la definición de una interfaz funcional!). El truco está en que los métodos que presentaremos se denominan *métodos predeterminados*/*métodos por defecto* (no son métodos abstractos). Por ahora, confía en nosotros y lee el capítulo 13 más adelante, cuando quieras saber más sobre los métodos predeterminados y sus posibilidades.

3.8.1 Composición de Comparadores

Ya has visto que puedes usar el método estático `Comparator.comparing` para devolver un Comparator basado en una *Function* que extrae una clave para la comparación, como se muestra a continuación:

```
Comparator<Apple> c = Comparator.comparing(Apple::getWeight);
```

ORDEN INVERTIDO

¿Qué ocurre si quieres ordenar las manzanas por peso descendente? No es necesario crear una instancia diferente de `Comparator`. La interfaz incluye un método predeterminado, reversed, que invierte el orden de un comparador dado. Puedes modificar el ejemplo anterior para ordenar las manzanas por peso descendente reutilizando el `Comparator` inicial:

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

Ordena por peso decreciente

ENCADENAMIENTO DE COMPARADORES

Todo esto está muy bien, pero ¿qué ocurre si encuentras dos manzanas con el mismo peso? ¿Qué manzana debería tener prioridad en la lista ordenada? Quizás quieras proporcionar un segundo `Comparator` para refinar aún más la comparación. Por ejemplo, después de comparar dos manzanas según su peso, podrías ordenarlas por país de origen. El método `thenComparing` te permite hacerlo. Recibe una función como parámetro (como el método de `comparing`) y proporciona un segundo `Comparador` si dos objetos se consideran iguales según el `Comparador` inicial. Puedes resolver el problema de forma elegante de la siguiente manera:

```
inventory.sort(comparing(Apple::getWeight)
          .reversed()
          .thenComparing(Apple::getCountry));
```

Ordena por peso decreciente

Se clasifica aún más por país cuando dos manzanas tienen el mismo peso.

3.8.2 Composición de predicados

La interfaz `Predicate` incluye tres métodos que permiten reutilizar un predicado existente para crear otros más complejos: `negate`, `and` y `or`. Por ejemplo, se puede usar el método negate para devolver la negación de `Predicate`, como una manzana que no es roja:

```
Predicate<Apple> notRedApple = redApple.negate();     <-- Produce la negación del
                                                          predicado existente objeto
                                                          redApple
```

También se pueden combinar dos lambdas para indicar que una manzana es roja y pesada con el método `and`:

```
Predicate<Apple> redAndHeavyApple =
    redApple.and(apple -> apple.getWeight() > 150);   <-- Encadena dos predicados
                                                          para producir otro objeto
                                                          Predicado.
```

El predicado resultante se puede combinar aún más para expresar manzanas rojas y pesadas (de más de 150 g) o solo manzanas verdes:

```
Predicate<Apple> redAndHeavyAppleOrGreen =
    redApple.and(apple -> apple.getWeight() > 150)
            .or(apple -> GREEN.equals(a.getColor()));  <-- Encadena tres predicados
                                                           para construir un objeto
                                                           Predicate más complejo.
```

¿Por qué es esto útil? A partir de expresiones lambda más simples, se pueden representar expresiones lambda más complejas que se leen como el enunciado del problema. Tenga en cuenta que la precedencia de los métodos and y or en la cadena es de izquierda a derecha; no existe un equivalente a los paréntesis. Por lo tanto, `a.or(b).and(c)` debe leerse como `(a || b) && c`. De manera similar, `a.and(b).or(c)` debe leerse como `(a && b) || c`.

3.8.3 Composición de funciones

Finalmente, también puede componer expresiones lambda representadas por la interfaz `Function`. Esta interfaz incluye dos métodos predeterminados para ello: `andThen` y `compose`, que devuelven una instancia de `Function`.

El método `andThen` devuelve una función que primero aplica una función dada a una entrada y luego aplica otra función al resultado de dicha aplicación. Por ejemplo, dada una función `f` que incrementa un número `(x ->x + 1)` y otra función `g` que multiplica un número por 2, puede combinarlas para crear una función `h` que primero incrementa un número y luego multiplica el resultado por 2:

```
Function<Integer, Integer> f = x -> x + 1;         En matemáticas se
Function<Integer, Integer> g = x -> x * 2;         escribiría g(f(x)) o
Function<Integer, Integer> h = f.andThen(g);   <-- (g o f)(x).
int result = h.apply(1);                       <-- Esto retorna 4
```

También puede usar el método `compose` de forma similar para aplicar primero la función dada como argumento y luego aplicarla al resultado. Por ejemplo, en el ejemplo anterior usando `compose`, esto significaría `f(g(x))` en lugar de `g(f(x))` usando `andThen`:

```
Function<Integer, Integer> f = x -> x + 1;         En matemáticas se
Function<Integer, Integer> g = x -> x * 2;         escribiría f(g(x)) o
Function<Integer, Integer> h = f.compose(g);   <-- (f o g)(x)
int result = h.apply(1);                       <-- Esto retorna 3
```

La Figura 3.6 ilustra la diferencia entre `andThen` y *compose*.

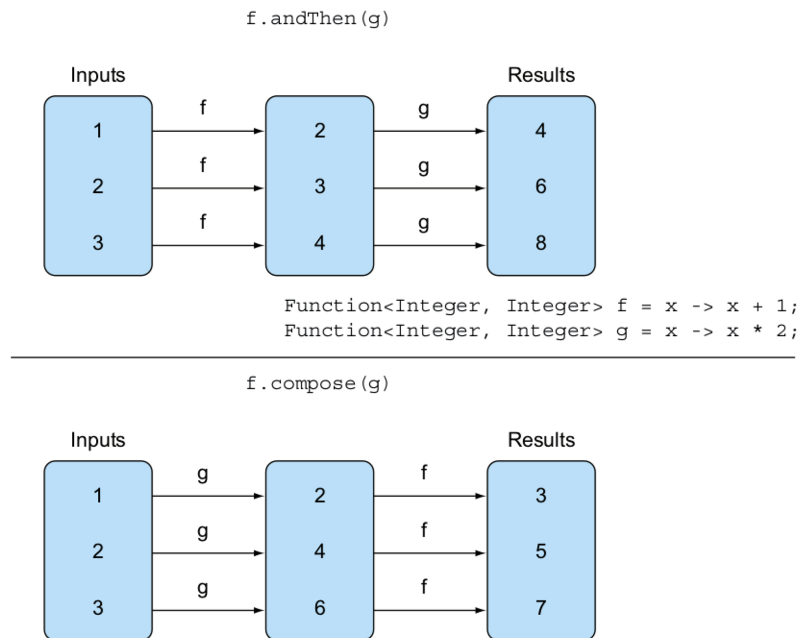

Figura 3.6 Usando `andThen` versus `compose`

Todo esto suena un poco abstracto. ¿Cómo se puede usar esto en la práctica? Supongamos que tienes varios métodos de utilidad que transforman texto en una letra representada como una `String`:

```
public class Letter{
    public static String addHeader(String text) {
        return "From Raoul, Mario and Alan: " + text;
    }
      public static String addFooter(String text) {
          return text + " Kind regards";
      }
      public static String checkSpelling(String text) {
          return text.replaceAll("labda", "lambda");
      }
  }
```

Ahora puedes crear varias secuencias de transformación combinando estos métodos. Por ejemplo, puedes crear una secuencia que primero agregue un encabezado, luego revise la ortografía y finalmente agregue un pie de página, como se muestra a continuación (y como se ilustra en la figura 3.7):

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::checkSpelling)
             .andThen(Letter::addFooter);
```

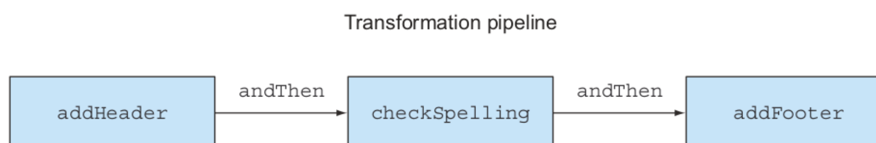

Figura 3.7 Una secuencia de transformación usando `andThen`

Una segunda secuencia podría consistir simplemente en agregar un encabezado y un pie de página sin revisar la ortografía:

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::addFooter);
```

3.9 Ideas similares de las matemáticas

Si te sientes cómodo con las matemáticas de bachillerato, esta sección ofrece otra perspectiva sobre el concepto de expresiones lambda y el paso de funciones. Puedes saltártela; el resto del libro no depende de ella. Sin embargo, puede que te interese conocer otra perspectiva.

3.9.1 Integración

Supongamos que tenemos una función f (matemática, no de Java), definida, por ejemplo, por:

$f(x) = x + 10$

Entonces, una pregunta frecuente (tanto en el colegio como en las carreras de ciencias e ingeniería) es cómo calcular el área bajo la curva de la función al representarla gráficamente (considerando el eje `x` como la línea cero). Por ejemplo, se escribe:

$\int_3^7 f(x)\,dx \quad \text{o} \quad \int_3^7 (x+10)\,dx$

para el área mostrada en la figura 3.8

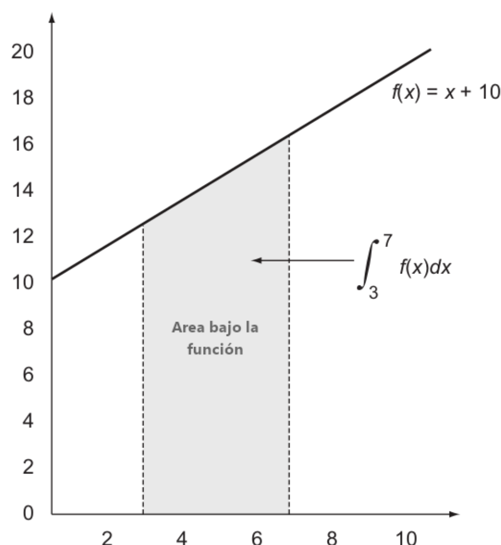

Figura 3.8 Área bajo la función $f(x) = x + 10$ para x entre 3 y 7

En este ejemplo, la función f es una línea recta, por lo que se puede calcular fácilmente esta área mediante el método del trapecio (dibujando triángulos y rectángulos) para encontrar la solución:

```
1/2 × ((3 + 10) + (7 + 10)) × (7 − 3) = 60
```

Ahora bien, ¿cómo se expresaría esto en Java? El primer problema es conciliar la notación extraña, como el símbolo de integración o dy/dx, con la notación familiar del lenguaje de programación.

De hecho, desde los principios básicos, se necesita un método, quizás llamado `integrate`, que tome tres argumentos: uno es `f` y los otros dos son los límites (3.0 y 7.0 en este caso). Por lo tanto, en Java se debe escribir algo como esto, donde la función f se pasa como argumento:

```
integrate(f, 3, 7)
```

Tenga en cuenta que no se puede escribir algo tan simple como

```
integrate(x + 10, 3, 7)
```

por dos razones. Primero, el alcance de `x` no está claro, y segundo, esto pasaría el valor `x+10` a la función de integración en lugar de la función `f`.

De hecho, la función oculta de `dx` en matemáticas es indicar *la función que toma como argumento x y cuyo resultado es* `x + 10`.

3.9.2 Conexión con expresiones lambda en Java 8

Como mencionamos anteriormente, Java 8 utiliza la notación `(double x) ->x + 10` (una expresión lambda) precisamente para este propósito; por lo tanto, puede escribir `integrate((double x) ->x + 10, 3, 7)` o `integrate((double x) ->f(x), 3, 7)` o, utilizando una referencia a método como se mencionó anteriormente, `integrate(C::f, 3, 7)`.

Si `C` es una clase que contiene `f` como método estático, la idea es que se le pasa el código de `f` al método `integrate`.

Ahora bien, ¿cómo se escribiría el método `integrate`? Supongamos que `f` es una función lineal (una línea recta). Probablemente te gustaría escribirlo de forma similar a las matemáticas:

```
public double integrate((double -> double) f, double a, double b) {
      return (f(a) + f(b)) * (b - a) / 2.0
}
```

¡Código Java incorrecto! (No se pueden escribir funciones como en matemáticas).

Pero como las expresiones lambda solo se pueden usar en un contexto que espera una interfaz funcional (en este caso, `DoubleFunction`[1], tienes que escribirlo de la siguiente manera:

```
public double integrate(DoubleFunction<Double> f, double a, double b) {
      return (f.apply(a) + f.apply(b)) * (b - a) / 2.0;
}
```

o usando `DoubleUnaryOperator`, que también evita el empaquetado del resultado:

```
public double integrate(DoubleUnaryOperator f, double a, double b) {
      return (f.applyAsDouble(a) + f.applyAsDouble(b)) * (b - a) / 2.0;
}
```

Como comentario aparte, es una lástima tener que escribir `f.apply(a)` en lugar de simplemente `f(a)` como en matemáticas, ¡pero Java no puede desprenderse de la idea de que todo es un objeto en lugar de la idea de que una función sea verdaderamente independiente!

Resumen

- Una *expresión lambda* puede entenderse como una especie de función anónima: no tiene nombre, pero tiene una lista de parámetros, un cuerpo, un tipo de retorno y, posiblemente, una lista de excepciones que se pueden lanzar.
- Las expresiones lambda permiten pasar código de forma concisa.
- Una *interfaz funcional* es una interfaz que declara un único método abstracto.
- Las expresiones lambda solo se pueden usar donde se espera una interfaz funcional.
- Las expresiones lambda permiten proporcionar la implementación del método abstracto de una interfaz funcional directamente en línea y *tratar toda la expresión como una instancia de dicha interfaz*.
- Java 8 incluye una lista de interfaces funcionales comunes en el paquete `java.util.function`, que comprende `BinaryOperator<T>`, `Predicate<T>`, `Function<T, R>`, `Supplier<T>` y `Consumer<T>`, descritas en la tabla 3.2.

[1] usar `DoubleFunction<Double>` es más eficiente que usar `Function<Double,Double>`, ya que evita el empaquetado del resultado)

- Se pueden usar especializaciones primitivas de interfaces funcionales genéricas comunes, como `Predicate<T>` y `Function<T, R>`, para evitar operaciones de empaquetado: `IntPredicate`, `IntToLongFunction`, etc.
- El patrón de ejecución iterativa (para cuando se necesita ejecutar un comportamiento específico en medio de código repetitivo necesario en un método, por ejemplo, asignación y limpieza de recursos) se puede usar con expresiones lambda para obtener mayor flexibilidad y reutilización.
- El tipo esperado para una expresión lambda se denomina tipo de *destino*.
- Las referencias a métodos permiten reutilizar una implementación de método existente y pasarla directamente.
- Las interfaces funcionales como `Comparator`, `Predicate` y `Function` tienen varios métodos predeterminados que se pueden usar para combinar expresiones lambda.

Referencias

- **Urma Raoul-Gabriel**, **Fusco Mario**, **Mycroft Alan** - “*Modern Java in Action*” Ed. Mannig Shelter Island, 2019