Expresiones Lambda

David J. Eck - Hobart and William Smith Colleges

En un programa en ejecución, una subrutina es simplemente una secuencia de números binarios (que representan instrucciones) almacenada en la memoria del ordenador. Considerada como una larga cadena de ceros y unos, una subrutina no parece muy diferente de un valor de datos como, por ejemplo, un entero, una cadena o un array, que también se representa como una cadena de ceros y unos en la memoria. Solemos pensar en las subrutinas y los datos como cosas muy distintas, pero internamente, una subrutina es simplemente otro tipo de datos. Algunos lenguajes de programación permiten trabajar con una subrutina como si fuera un valor de datos. En Java 8, esta capacidad se añadió a Java mediante las **expresiones lambda**.

Las expresiones lambda son cada vez más comunes en los programas Java. Son especialmente útiles para trabajar con el kit de herramientas GUI de JavaFX, y será útil conocerlas antes de abordar la programación GUI.

4.5.1 Funciones de primera clase

Lambda es una letra del alfabeto griego que el matemático Alonzo Church utilizó en su estudio de las funciones computables. Su notación lambda permite definir una función sin darle un nombre. Por ejemplo, podrías pensar que la notación x^2 es una forma perfectamente válida de representar una función que eleva un número al cuadrado, pero en realidad, es una expresión que representa el resultado de elevar *x* al cuadrado, lo que deja abierta la cuestión de qué representa *x*. Podemos definir una función con *x* como parámetro ficticio:

```
static double square( double x ) {
    return x*x;
}
```

pero para ello, tuvimos que nombrar la función `square`, y esa función se convierte en una parte permanente del programa, lo cual es excesivo si solo queremos usarla una vez. Alonzo Church introdujo la notación *lambda*(*x*).x^2 para representar «*la función de x dada por* x^2» (utilizando la letra griega en lugar de la palabra «*lambda*»). Esta notación es un tipo de literal de función que representa un valor de tipo «*función*», de la misma manera que 42 es un literal entero que representa un valor de tipo `int`.

Tener literales de función es el punto de partida para considerar una función como un tipo más de valor de datos. Una vez que lo hacemos, deberíamos poder realizar con las funciones las mismas acciones que con otros valores, como asignar una función a una variable, pasar una función como parámetro a una subrutina, devolver una función como valor de una subrutina o incluso crear un array de funciones. Un lenguaje de programación que permite realizar todas estas acciones con funciones se denomina lenguaje con «*funciones de primera clase*» o «*funciones como objetos de primera clase*».

De hecho, se pueden realizar todas estas acciones con expresiones lambda en Java. La notación de Java es diferente de la utilizada por Alonzo Church, y a pesar del nombre "*expresión lambda*", ni siquiera utiliza la palabra *lambda*. En Java, la expresión

lambda para una función de elevación al cuadrado como la anterior se puede escribir

```
x  ->  x*x
```

El operador **->** es lo que convierte esto en una expresión *lambda*. El parámetro ficticio para la función se encuentra a la izquierda del operador, y la expresión que calcula el valor de la función está a la derecha. Es posible que veas una expresión como esta pasándose como parámetro a una subrutina, asignándose a una variable o devolviendo una función.

Entonces, ¿son las funciones ahora de primera clase en Java? No estoy del todo seguro. Hay algunas cosas interesantes que se pueden hacer en otros lenguajes pero no en Java. Por ejemplo, en Java podemos asignar la expresión anterior a una variable llamada, digamos, `sqr`, pero luego no podemos usar `sqr` como si fuera una función. Por ejemplo, no podemos escribir `sqr(42)`. El problema, en realidad, es que Java es un lenguaje fuertemente tipado; para tener una variable llamada `sqr`, debemos declararla y asignarle un tipo. Pero, ¿qué tipo sería apropiado para un valor que es una función? La respuesta en Java es algo llamado **interfaz funcional**, que veremos a continuación.

Pero antes, una aclaración: las expresiones lambda en Java pueden representar subrutinas arbitrarias, no solo funciones. Sin embargo, el término "función" suele asociarse a ellas, en lugar de "subrutina" o "método".

4.5.2 Interfaces funcionales

Para saber cómo se puede usar legalmente una subrutina, es necesario conocer su nombre, cuántos parámetros requiere, sus tipos y el tipo de retorno. Una interfaz funcional especifica esta información sobre una subrutina. Una interfaz funcional es similar a una clase y se puede definir en un archivo `.java`, al igual que una clase. Sin embargo, su contenido es solo una especificación para una única subrutina. He aquí un ejemplo:

```
public interface FunctionR2R {
    double valueAt( double x );
}
```

Este código estaría en un archivo llamado `FunctionR2R.java`. Especifica una función llamada `valueAt` con un parámetro de tipo `double` y un tipo de retorno `double`. (El nombre del parámetro, *x*, no forma parte de la especificación, y resulta un poco molesto que tenga que estar ahí). Aquí hay otro ejemplo:

```
public interface ArrayProcessor {
    void process( String[] array, int count );
}
```

Java incluye muchas interfaces funcionales estándar. Una de las más importantes es una muy simple llamada **Runnable**, que ya está definida en Java como:

```
public interface Runnable {
    public void run();
}
```

En esta sección, usaré estas tres interfaces funcionales como ejemplos.

Las *interfaces* en Java pueden ser mucho más complejas que las interfaces funcionales. Aprenderá más sobre ellas en la Sección 5.7. Sin embargo, solo las interfaces funcionales son relevantes para las expresiones lambda: una interfaz funcional proporciona una plantilla para una subrutina que puede representarse mediante una expresión lambda. El nombre de una interfaz funcional es un tipo, al igual que `String` y `double`. Es decir, se puede usar para declarar variables y parámetros, y para especificar el tipo de retorno de una función. Cuando un tipo es una interfaz funcional, se puede proporcionar un valor para ese tipo como una expresión lambda.

4.5.3 Expresiones Lambda

Una expresión lambda representa una subrutina anónima, es decir, sin nombre. Sin embargo, sí tiene una lista de parámetros formal y una definición. La sintaxis completa es:

(⟨*parameter-list*⟩) -> { ⟨*statements*⟩ }

Al igual que en una subrutina normal, la *lista de parámetros* puede estar vacía o ser una lista de declaraciones de parámetros separadas por comas, donde cada declaración consta de un tipo seguido de un nombre de parámetro. Sin embargo, la sintaxis a menudo se puede simplificar. En primer lugar, se pueden omitir los tipos de parámetros, siempre que se puedan deducir del contexto. Por ejemplo, si se sabe que la expresión lambda es de tipo `FunctionR2R`, entonces el tipo de parámetro debe ser **double**, por lo que no es necesario especificar el tipo de parámetro en la expresión lambda. A continuación, si hay exactamente un parámetro y no se especifica su tipo, se pueden omitir los paréntesis alrededor de la lista de parámetros. En el lado derecho de **->**, si lo único que hay entre las llaves **{** y **}** es una única instrucción de llamada a subrutina, se pueden omitir las llaves. Y si el lado derecho tiene la forma `{ return` *expression;* **}**, entonces puedes omitir todo excepto la expresión.

Por ejemplo, supongamos que queremos una expresión lambda para representar una función que calcule el cuadrado de un valor **double**. El tipo de dicha función puede ser la interfaz *FunctionR2R* mencionada anteriormente. Si `sqr` es una variable de tipo *FunctionR2R*, entonces el valor de la función puede ser una expresión lambda, que se puede escribir de cualquiera de las siguientes formas:

```
sqr = (double x) -> { return x*x; };
sqr = (x) -> { return x*x; };
sqr = x -> { return x*x; };
sqr = x -> x*x;
sqr = (double fred) -> fred*fred;
sqr = (z) -> z*z;
```

Las dos últimas afirmaciones se incluyen para enfatizar que los nombres de los parámetros en una expresión lambda son parámetros ficticios; sus nombres son irrelevantes. Las seis expresiones lambda en estas afirmaciones definen exactamente la misma función. Ten en cuenta que el tipo de parámetro `double` se puede omitir porque el compilador sabe que `sqr` es de tipo *FunctionR2R*, y una *FunctionR2R* requiere un parámetro de tipo `double`. Una expresión lambda solo se puede usar en un contexto donde el compilador pueda deducir su tipo, y el tipo de parámetro solo debe incluirse en un caso en el que omitirlo haría que el tipo de la expresión lambda fuera ambiguo.

En Java, la variable `sqr`, tal como se define aquí, no es propiamente una función. Es un valor de tipo `FunctionR2R`, lo que significa que contiene una función llamada `valueAt`, tal como se especifica en la definición de la interfaz `FunctionR2R`. El nombre completo de esa función es `sqr.valueAt`, y debemos usar ese nombre para llamarla. Por ejemplo: `sqr.valueAt(42)` o `sqr.valueAt(x) + sqr.valueAt(y)`.

Cuando una expresión lambda tiene dos parámetros, los paréntesis son obligatorios. Aquí se muestra un ejemplo del uso de la interfaz `ArrayProcessor`, que también ilustra una expresión lambda con una definición multilínea:

```
ArrayProcessor concat;
concat = (A,n) -> { // parentheses around (A,n) are required!
    String str;
    str = "";
    for (int i = 0; i < n; i++)
        str += A[i];
    System.out.println(str);
};  // The semicolon marks the end of the assignment statement;
    //      it is not part of the lambda expression.

String[] nums;
nums = new String[4];
nums[0] = "One";
nums[1] = "Two";
nums[2] = "Three";
nums[3] = "Four";
for (int i = 1; i < nums.length; i++) {
  concat.process( nums, i );
}
```

Esto imprimirá:

```
One
OneTwo
OneTwoThree
OneTwoThreeFour
```

La cosa se pone más interesante cuando una expresión lambda se usa como parámetro, que es el uso más común en la práctica. Por ejemplo, supongamos que se define la siguiente función:

```
/**
 *  For a function f, compute f(start) + f(start+1) + ... + f(end).
 *  The value of end should be >= the value of start.
 */
static double sum( FunctionR2R f, int start, int end ) {
    double total = 0;
    for (int n = start; n <= end; n++) {
        total = total + f.valueAt( n );
    }
    return total;
}
```

Tenga en cuenta que, dado que *f* es un valor de tipo *FunctionR2R*, el valor de *f* en *n* se escribe como `f.valueAt(n)`. Cuando se llama a la función `sum`, el primer

parámetro puede especificarse como una expresión lambda. Por ejemplo:

```
System.out.print("The sum of n squared for n from 1 to 100 is ");
System.out.println( sum( x -> x*x, 1, 100 ) );
System.out.print("The sum of 2 raised to the power n, for n from 1 to 10 is ");
System.out.println( sum( num -> Math.pow(2,num), 1, 10 ) );
```

Como otro ejemplo, supongamos que tenemos una subrutina que realiza una tarea determinada varias veces. La tarea puede especificarse como un valor de tipo `Runnable`:

```
static void doSeveralTimes( Runnable task, int repCount ) {
    for (int i = 0; i < repCount; i++) {
        task.run();  // Perform the task!
    }
}
```

Podríamos entonces decir `Hola Mundo` diez veces llamando a

```
doSeveralTimes( () -> System.out.println("Hello World"), 10 );
```

Tenga en cuenta que, para una expresión lambda de tipo *Runnable*, la lista de parámetros se proporciona como un par de paréntesis vacíos. Aquí hay un ejemplo en el que la sintaxis se vuelve bastante compleja:

```
doSeveralTimes( () -> {
        // count from 1 up to some random number between 5 and 25
    int count = 5 + (int)(21*Math.random());
    for (int i = 1; i <= count; i++) {
        System.out.print(i + " ");
    }
    System.out.println();
}, 100);
```

Esta es una llamada a una subrutina en la que el primer parámetro es una expresión lambda que se extiende a lo largo de varias líneas. El segundo parámetro es 100, y el punto y coma en la última línea finaliza la llamada a la subrutina.

Hemos visto ejemplos de cómo asignar una expresión lambda a una variable y cómo usarla como parámetro. Aquí tenemos un ejemplo en el que una expresión lambda es el valor de retorno de una función:

```
static FunctionR2R makePowerFunction( int n ) {
   return x -> Math.pow(x,n);
}
```

Entonces, `makePowerFunction(2)` devuelve una `FunctionR2R` que calcula el cuadrado de su parámetro, mientras que `makePowerFunction(10)` devuelve una *FunctionR2R* que calcula la décima potencia de su parámetro. Este ejemplo también ilustra que una expresión lambda puede usar otras variables además de su parámetro, como `n` en este caso (aunque existen algunas restricciones sobre cuándo se puede hacer).

4.5.4 Referencias a métodos

Supongamos que queremos que una expresión lambda represente la raíz cuadrada como un valor de tipo `FunctionR2R`. Podríamos escribirlo como `x ->Math.sqrt(x)`. Sin embargo, esta expresión lambda es simplemente una envoltura para una función Math.sqrt que ya existe. En lugar de escribir la expresión lambda, esa función se puede escribir como una referencia a un método, que tiene la forma `Math::sqrt`. (Recordemos que en Java, **método** es sinónimo de **subrutina**"). Esta referencia a un método es simplemente una abreviatura de la expresión lambda y se puede usar dondequiera que se pueda usar esa expresión lambda, como en la función *sum* definida anteriormente:

```
System.out.print("The sum of the square root of n for n from 1 to 100 is ");
System.out.println( sum( Math::sqrt, 1, 100 ) );
```

Sería ideal poder usar simplemente el nombre `Math.sqrt` aquí en lugar de introducir una nueva notación con **::**, pero la notación `Math.sqrt` ya se definió para referirse a una **variable** llamada `sqrt` en la clase `Math`.

En general, una expresión lambda que simplemente llama a un método estático existente se puede escribir como una referencia a un método de la forma:

⟨*classname*⟩ :: ⟨*method-name*⟩

Además, esta notación se extiende a métodos que se encuentran en objetos en lugar de clases. Por ejemplo, si `str` es un `String`, entonces `` `str` `` contiene el método `str.length()`. La referencia al método `str::length` podría usarse como una expresión lambda de tipo `SupplyInt`, donde `SupplyInt` es la interfaz funcional.

```
public interface SupplyInt {
    int get( );
}
```