Expresiones Lambda y Referencias a Métodos

Java introdujo las expresiones *lambda* en Java 8, aportando capacidades de programación funcional al lenguaje. Esta incorporación ha mejorado significativamente la expresividad y la concisión del código Java. Junto con las expresiones *lambda*, las referencias a métodos también desempeñan un papel crucial en la simplificación del código. Aquí exploraremos en detalle las expresiones *lambda*, las interfaces funcionales y las referencias a métodos.

1. ¿Qué son las expresiones *lambda*?

En Java, las expresiones *lambda* permiten crear funciones anónimas que se pueden pasar como si fueran objetos. Proporcionan una forma clara y concisa de representar la interfaz de un método mediante una expresión.

Parámetros de las expresiones *lambda*

1. Parámetro cero:

```
() -> System.out.println("Lambda Function tutorial");
```

2. Parámetro único:

```
(x) -> System.out.println("Object: " + x);
```

3. Múltiples parámetros:

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

1.1. Dónde usar expresiones *lambda*

- **En interfaces funcionales**: Una interfaz funcional es una interfaz con un *único método abstracto* (SAM). Las expresiones lambda se utilizan a menudo para implementar estos métodos.
- **Cuando la lógica es compleja**: Las expresiones *lambda* son ideales cuando se necesita implementar una pequeña parte de la lógica en una sola línea o bloque, sin tener que escribir un método completo. Si la lógica es demasiado compleja, es mejor evitar las referencias a métodos.

1.2. Ejemplos de uso de expresiones Lambda

1. **Uso de una expresión *Lambda* en una operación de colección o flujo**: Por ejemplo, puede usar expresiones *Lambda* para definir el comportamiento en bucles `forEach`, `map`, `filter`, etc., cuando trabaje con colecciones o flujos.

```
List<String> names = Arrays.asList("Alicia", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));
```

```
import java.util.Arrays;
import java.util.List;

public class Ejemplo01 {
    public static void main (String[] args) {
        List<String> names = Arrays.asList("Alicia", "Bob", "Charlie");
        names.forEach(name -> System.out.println(name));
    }
}
```

Aquí, `name ->System.out.println(name)` es una expresión *lambda*, donde `name` es el parámetro y el cuerpo es `System.out.println(name)`.

2. **En el manejo de eventos (aplicaciones GUI)**: En Java, las expresiones *lambda* se usan comúnmente para simplificar las implementaciones de oyentes de eventos, donde una interfaz funcional (por ejemplo, `ActionListener`) requiere la implementación de un único método.

```
button.addActionListener(e -> System.out.println("Button clicked!"));
```

3. **Al realizar operaciones como ordenar o transformar datos**: puede pasar una expresión *lambda* al método *sort()* u otras transformaciones de datos.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.sort((s1, s2) -> s1.compareTo(s2)); // Sorting using lambda
```

2. Referencias a métodos

Una **referencia a un método** es una notación abreviada para llamar a un método directamente. Se refiere a un método existente en tu código y lo hace más conciso y legible.

Sintaxis

```
ClassName::methodName o instance::methodName
```

2.1. Cuándo usar referencias a métodos

- **Cuando la expresión lambda simplemente llama a un método existente**: Si una expresión *lambda* simplemente llama a un método existente, las referencias a métodos pueden usarse como una alternativa más concisa. Son ideales cuando la expresión *lambda* simplemente delega en un método.
- **Cuando se busca mayor claridad y legibilidad**: Si la lógica de la expresión *lambda* simplemente llama a un método existente, se prefieren las referencias a métodos porque brindan mayor claridad y son más legibles.

2.2. Tipos de referencias a métodos

1. **Referencia a un método estático**: Se puede hacer referencia a un método estático directamente usando el nombre de la clase y el nombre del método.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
// Method reference to static method 'println'
numbers.forEach(System.out::println);
```

2. **Referencia a un método de instancia de un objeto en particular**: Puede utilizar una referencia a un método cuando este se invoca en una instancia específica.

```
// Lambda calling instance method
names.forEach(name -> cat.speak(name));
```

3. **Referencia a un método de instancia de un objeto arbitrario de un tipo particular**: Puede utilizar referencias a métodos cuando el método se invocará en un objeto arbitrario de un tipo particular (por ejemplo, para ordenar una lista).

```
List<String> names = new ArrayList<>(List.of("Tom", "Jerry", "Spike"));
names.sort(String::compareToIgnoreCase);
```

4. **Referencia a un constructor**: Puede utilizar una referencia a un método para llamar a un constructor.

```
List<String> names = Arrays.asList("Tom", "Jerry", "Spike");
List<Cat> cats = names.stream().map(Cat::new).collect(Collectors.toList());
```

2.3. ¿Cuándo usar expresiones *lambda* y cuándo referencias a métodos?

Las expresiones *lambda* se deben usar cuando se necesita implementar lógica o comportamiento personalizado, o cuando la lógica es más compleja y requiere más que simplemente invocar un método.

Ejemplo:

```
(x, y) -> x + y (una simple suma).
```

Las referencias a métodos se deben usar cuando la expresión *lambda* se puede reemplazar llamando a un método ya existente, especialmente si esto mejora la legibilidad.

Ejemplo:

```
Referencia a un método existente para convertir una cadena a mayúsculas
String::toUpperCase
```

3. Velocidad al usar funciones *lambda*

La velocidad de las funciones *lambda* puede variar según varios factores, como la implementación de la JVM, el tamaño de los datos que se procesan y cómo se utiliza la *lambda*. En general, las expresiones *lambda* no son más lentas que las implementaciones de clases anónimas equivalentes. Sin embargo, hay que tener en cuenta lo siguiente:

1. **Optimización JIT**: El compilador *Just-In-Time* (**JIT**) de las JVM modernas optimiza las *lambdas* con la misma eficacia que los métodos tradicionales o las clases anónimas. Esto significa que las *lambdas* pueden ser igual de rápidas en su ejecución.
2. **Sobrecarga**: Si se utilizan expresiones *lambda* en código crítico para el rendimiento, puede haber una sobrecarga mínima debido a la creación de identificadores de método u objetos de función adicionales en tiempo de ejecución. Sin embargo, esta sobrecarga suele ser insignificante, a menos que se realicen operaciones de alta frecuencia en bucles cerrados.
3. **Operaciones con flujos**: Al usar *lambdas* con flujos (por ejemplo, `map`, `filter`, etc.), el rendimiento depende de cómo se procesa el flujo. En ocasiones, el procesamiento en paralelo puede mejorar el rendimiento con grandes conjuntos de datos, pero puede generar una sobrecarga debido a la paralelización. En casos más sencillos, la diferencia de rendimiento es mínima.

4. Expresiones Lambda de Java

Las expresiones *lambda* de Java, introducidas en Java 8, permiten a los desarrolladores escribir código conciso y funcional mediante la representación de funciones anónimas. Permiten pasar código como parámetros o asignarlo a variables, lo que resulta en programas más limpios y legibles.

- Las expresiones *lambda* implementan una interfaz funcional (una interfaz con una única función abstracta).
- Permiten pasar código como datos (*argumentos de método*).
- Las expresiones *lambda* solo pueden acceder a variables finales o efectivamente finales del ámbito que las contiene.
- Las expresiones *lambda* no pueden lanzar excepciones verificadas a menos que la interfaz funcional las declare.
- Permiten definir el comportamiento sin crear clases separadas.

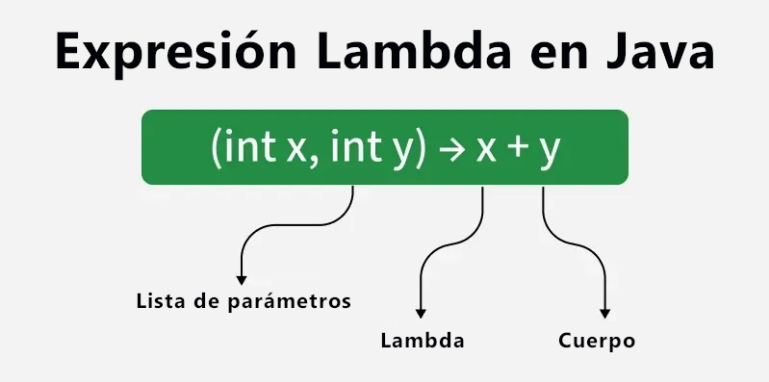
```
interface Add{

    int addition(int a, int b);
}

public class GFG{

    public static void main(String[] args){

        // Expresión Lambda para sumar dos números
        Add add = (a, b) -> a + b;

        int resultado = add.addition(10, 20);
        System.out.println("Suma: " + resultado);
    }
}
```

Salida

```
Suma: 30
```

4.1. Sintaxis de las expresiones *Lambda*

La sintaxis de las expresiones *Lambda* en Java es la siguiente:



- **Lista de parámetros**: Parámetros para la expresión *lambda*.
- **Signo de flecha** (->): Separa la lista de parámetros del cuerpo.
- **Cuerpo**: Lógica a ejecutar.

4.2. Interfaz funcional

Una interfaz funcional tiene exactamente un método abstracto. Las expresiones lambda proporcionan su implementación. La anotación `@FunctionalInterface` es opcional, pero se recomienda para garantizar esta regla en tiempo de compilación.

```
interface FuncInterface{

    void abstractFun(int x);
    default void normalFun(){
        System.out.println("Hello");
        }
}

public class GFG{

    public static void main(String[] args){

        FuncInterface fobj = (int x) -> System.out.println(2 * x);
        fobj.abstractFun(5);
    }
}
```

Salida

```
10
```

4.3. Tipos de parámetros *Lambda*

A continuación se describen tres tipos de parámetros para expresiones Lambda:

4.3.1. *Lambda* sin parámetros

Sintaxis:

```
() -> System.out.println("Zero parameter lambda");
```

```
@FunctionalInterface
interface ZeroParameter{
    void display();
}

public class Geeks{
    public static void main(String[] args){

        // Expresión lambda con cero parámetros
        ZeroParameter zeroParamLambda = () -> System.out.println(
            "¡Esta es una expresión lambda sin parámetros!");

        // Invocar el método
        zeroParamLambda.display();
    }
}
```

Salida

```
¡Esta es una expresión lambda sin parámetros!
```

4.3.2. Función *lambda* con un solo parámetro

Sintaxis:

```
(p) -> System.out.println("Un parámetro: " + p);
```

No es obligatorio usar paréntesis si el tipo de la variable se puede inferir del contexto.

Los paréntesis son opcionales si el compilador puede inferir el tipo del parámetro a partir de la interfaz funcional.

```
import java.util.ArrayList;

public class GFG{
    public static void main(String[] args){
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);

        System.out.println("Todos los elementos:");
        list.forEach(n -> System.out.println(n));

        System.out.println("Elementos pares:");
        list.forEach(n -> {
            if (n % 2 == 0)
                System.out.println(n);
        });
    }
}
```

Salida

```
Todos los elementos:
1
2
3
4
5
Elementos pares:
2
4
```

Nota: El método `forEach()` utiliza internamente la interfaz funcional `Consumer<T>`, que recibe un argumento y realiza una acción.

4.3.3. Expresión Lambda con Múltiples Parámetros

Sintaxis:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

```
@FunctionalInterface
interface Functional {
    int operation(int a, int b);
}

public class Test {
    public static void main(String[] args) {

        // Utilizar expresiones lambda para definir las operaciones
        Functional add = (a, b) -> a + b;
        Functional multiply = (a, b) -> a * b;

        // Utilizando las operaciones
        System.out.println(add.operation(6, 3));
        System.out.println(multiply.operation(4, 5));
    }
}
```

Salida

```
9
20
```

Nota: Las expresiones lambda son como funciones y aceptan parámetros, al igual que las funciones.

5. Ejemplos en colecciones y flujos

Las expresiones *lambda* se utilizan ampliamente con colecciones y flujos de Java para operaciones concisas.

```
import java.util.Arrays;
import java.util.List;

public class GFG{
    public static void main(String[] args){
        List<String> names = Arrays.asList(
            "Alice", "Bob", "Charlie", "Adam");

        System.out.println("Todos los nombres:");
        names.forEach(name -> System.out.println(name));

        System.out.println("\nNombres que empiezan con 'A':");
        names.stream()
            .filter(n -> n.startsWith("A"))
            .map(n -> n.toUpperCase())
            .forEach(System.out::println);
    }
}
```

Salida

```
All names:
Alice
Bob
Charlie
Adam

Nombres que empiezan por 'A':
ALICE
ADAM
```

Ventajas:

- **Código conciso**: Reduce el código repetitivo en comparación con las clases anónimas.
- **Programación funcional**: Trata las funciones como elementos de primera clase.
- **Mayor legibilidad**: El código es más fácil de leer y mantener.
- **Colecciones y flujos mejorados**: Simplifica operaciones como el filtrado, el mapeo y la iteración.

Interfaces funcionales integradas comunes

Interfaz	Método	Propósito
Predicado	boolean test(T t)	Comprueba una condición determinada y devuelve verdadero o falso.
Consumidor	void accept(T t)	Realiza una acción sobre el argumento dado sin devolver ningún resultado.
Proveedor	T get()	Proporciona o genera un resultado sin tomar ninguna entrada.
Comparador <T>	int compare(T o1, T o2)	Compara dos objetos para determinar su orden.
Comparable <T>	int compareTo(T o)	Define el orden natural de los objetos de una clase.

Verificación de validez: Ejemplos de expresiones *Lambda*

Expresión	Validez	Motivo
`() ->{ }`	Válido	Sin parámetros, cuerpo vacío
`() ->"geeksforgeeks"`	Válido	Una sola expresión devuelve un valor
`() ->{ return "geeksforgeeks";}`	Válido	Utiliza llaves con la palabra clave return
`(Integer i) ->{return "geeksforgeeks" + i;}`	Válido	Sintaxis correcta con parámetro tipado
`(String s) ->{return "geeksforgeeks";}`	Parámetro válido	Parámetro no utilizado pero válido
`() ->{return "Hello"}`	Inválido	Falta punto y coma después de la instrucción `return`
`x ->{return x + 1;}`	No válido	No válido si no es posible la inferencia de tipos
`(int x, y) ->x + y`	No válido	Si un parámetro tiene tipo, todos deben ser

6. Conclusión

Las expresiones *lambda* y las **referencias a métodos** son dos potentes características de Java que mejoran la legibilidad, la concisión y la mantenibilidad del código. Si bien las expresiones *lambda* son ideales para definir lógica personalizada, las referencias a métodos ofrecen una forma más limpia de referirse a métodos existentes. La elección entre una u otra depende de la complejidad del código que se esté escribiendo y de si se desea simplificar las llamadas a métodos existentes. Ambas características contribuyen a incorporar prácticas de programación funcional a Java, haciéndolo más expresivo y conciso.

7. Referencia

https://www.geeksforgeeks.org/java-lambda-expressions-parameters/?ref=oin_asr3
https://www.youtube.com/watch?v=DELCbBuCHHE
https://www.geeksforgeeks.org/java/lambda-expressions-java-8/