Tabla de Contenidos

¿Qué es un Java Lambda? . Esta es una de las preguntas más habituales que se hacen todos los programadores cuando comienzan a trabajar em Java . La programación funcional no son conceptos sencillos de entender . Vamos a explicarlo a través de varios ejemplos apoyándonos en el concepto de ordenación de colecciones . Para ello usaremos de entrada un ejemplo de Java Comparator que use el concepto de clase Persona que contiene como propiedades el nombre los apellidos y la edad .


```
package com.arquitecturajava;

public class Persona {
```

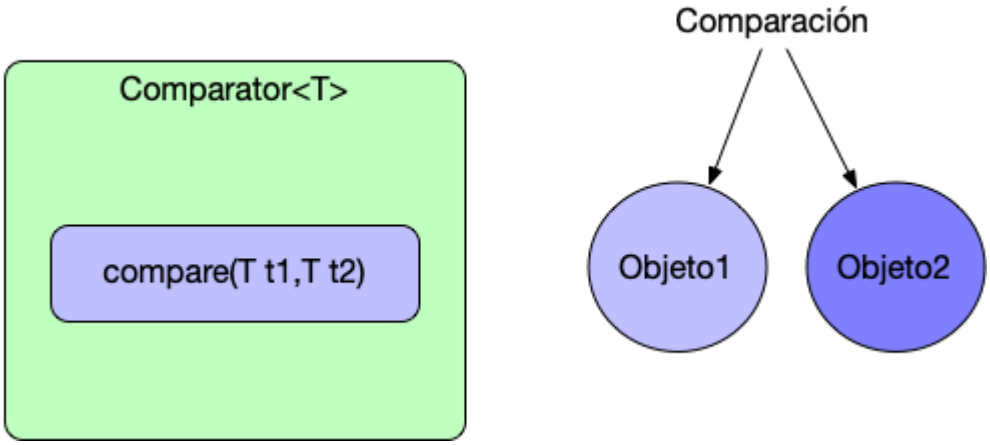
```
    private String nombre;
    private String apellidos;
    private int edad;
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
    public Persona(String nombre, String apellidos, int edad) {
        super();
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }

}
```

Java Comparator

Una vez que hemos construido la clase Persona vamos a construir un PersonaEdadComparator que encarga de comparar las personas por edad permitiéndonos su posterior ordenación dentro de una lista para ello deberemos obligar a la clase a implementar o el interface Comparable o el interface Comparator que es de tipo Genérico. En este caso optaremos por la segunda opción.

```
package com.arquitecturajava;

import java.util.Comparator;

public class PersonaEdadComparator implements Comparator<Persona> {
        @Override
        public int compare(Persona p1, Persona p2) {
                return p1.getEdad()>p2.getEdad()? 1:-1;
        }

}
```

Recordemos que el interface Comparator solo contiene un método compare que recibe dos parámetros y nos devuelve un valor numérico (1 ,0 o -1) dependiendo si un objeto es mayor que otro igual o menor a nivel de la propiedad o conjunto de propiedades que comparamos con nuestra clase. En este caso estamos comparando por edades usando el operador ternario para simplificar.



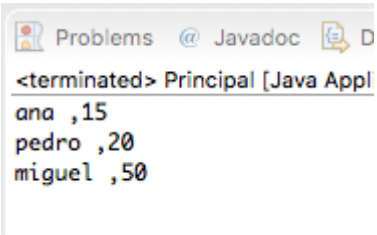
Listas y Ordenaciones

Una vez tenemos el comparador construido el siguiente paso es construir un programa Main que construya una lista de Personas y se encargue de ordenarlas por edad:

```
package com.arquitecturajava;

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

public class Principal {

    public static void main(String[] args) {

        Persona personaA = new Persona("pedro", "perez", 20);
        Persona personaB = new Persona("ana", "blanco", 15);
        Persona personaC = new Persona("miguel", "alvarez", 50);

        List < Persona > lista = Arrays.asList(personaA, personaB,
personaC);

        lista.sort(new PersonaEdadComparator());
        for (Persona p: lista) {
```

```
            System.out.println(p.getNombre() + " ," + p.getEdad());

        }

    }

}
```

Como el programa muestra se ha generado varios objetos Persona y se han añadido de forma rápida a la lista para posteriormente invocar al método sort del interface List que recibe un objeto de tipo Comparator y se encarga de comparar cada uno de los elementos de la lista . En nuestro caso hemos usado PersonaEdadComparator y nos ordenará por edad:



Sintaxis de Java Lambda vs InnerClasses.

Un muchos casos la gente suele preguntar si para ordenar la lista de elementos hace falta crear una nueva clase por cada tipo de ordenación ya que no es algo que resulte cómodo y genera una explosión de clases a nivel de código. La respuesta hasta Java 7 es que no es obligatorio y que es posible usar una clase Anónima o Inner class.

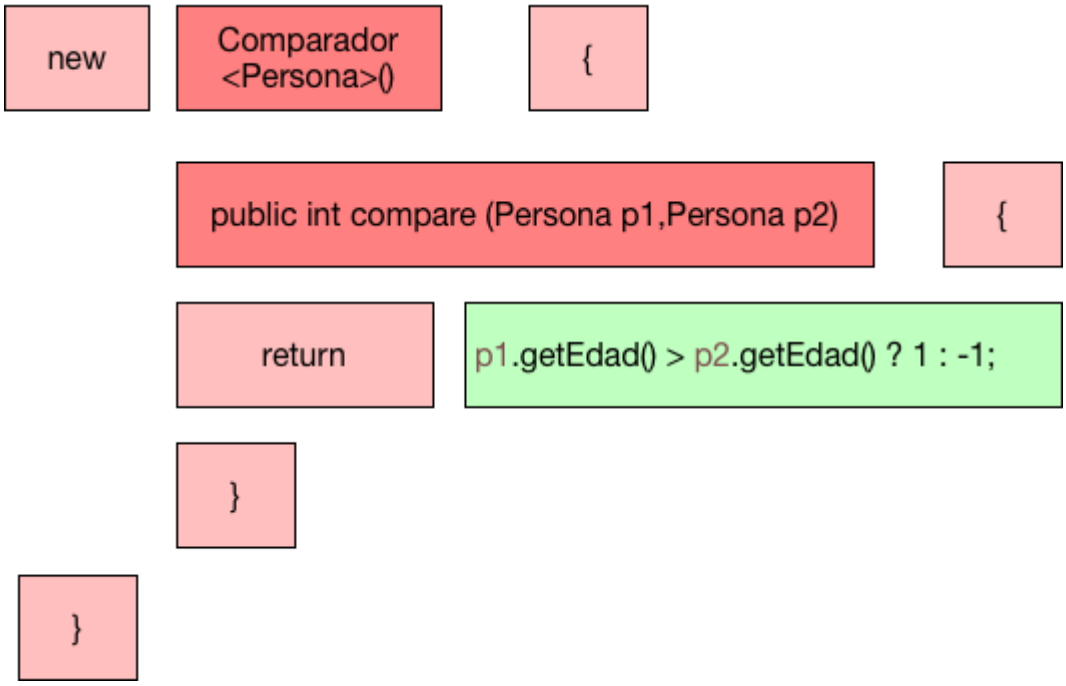
APUNTATE!!

```
import java.util.Comparator;
import java.util.List;

public class Principal2 {

    public static void main(String[] args) {

        Persona personaA = new Persona("pedro", "perez", 20);
        Persona personaB = new Persona("ana", "blanco", 15);
        Persona personaC = new Persona("miguel", "alvarez", 50);

        List < Persona > lista = Arrays.asList(personaA, personaB,
personaC);
        lista.sort(new Comparator < Persona > () {
            @Override
            public int compare(Persona p1, Persona p2) {
                return p1.getEdad() > p2.getEdad() ? 1 : -1;
            }
        });
        for (Persona p: lista) {
            System.out.println(p.getNombre() + " ," + p.getEdad());
        }

    }

}
```

Esta clase se define dentro de nuestro propio código y no puede ser reutilizada en ningún otro lugar ya que solo se aplica a una situación muy muy concreta.

Inner Classes y su sintaxis

El resultado en la pantalla será el mismo , sin embargo ya no hará falta construir una clase PersonaEdadComparator . Es evidente que hay ventajas pero también es evidente que la sintaxis es muy compleja y no ayuda a clarificar las cosas. Vamos a hacer un análisis de esta sintaxis.



Sintaxis y Reflexiones

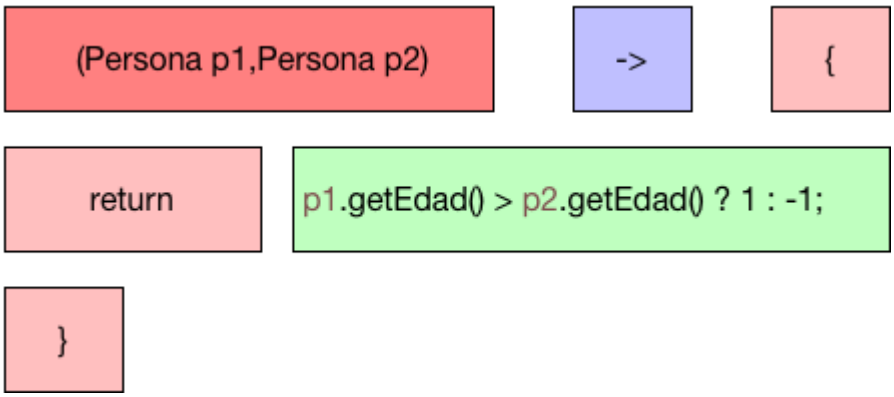
Realmente si nos ponemos a revisar el c´ódigo nos daremos cuenta que solo la condición de la comparación de edades es relevante (en verde) el resto del código es superfluo y no aporta gran cosa. Por mucho que nos duela esa es la realidad.

```
p1.getEdad()>p2.getEdad()?1:-1
```

Ahora bien al estar Java tan basado en el manejo de clases , este código era totalmente obligatorio hasta la versión de Java 7

Java Lambda y simplificaciones

La realidad es que tenemos mucho código que parece que aporta poco , sin embargo recordemos que si estamos todavía sobre el paraguas de Java 7 , la sintaxis es obligatoria. ¿Cómo podemos simplificarla con Java 8? . Podemos hacer uso de expresiones lambda y eliminar gran parte del código:



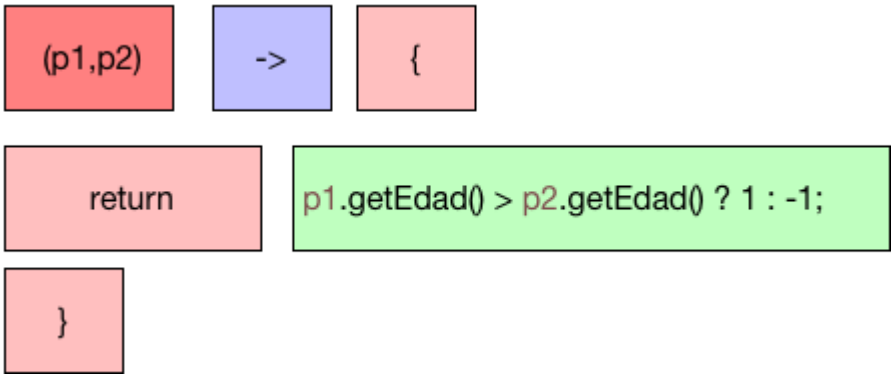
En este caso lo que hemos hecho es eliminar la clase anónima y quedarnos únicamente con lo que se denomina un bloque lambda el cual implementa la funcionalidad. de una manera funcional . Es decir define una función dentro de nuestro código con sus parámetros de entrada y retorno. Si mostramos el código las cosas quedan mucho más sencillas.

```
package com.arquitecturajava;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Principal3 {

    public static void main(String[] args) {

        Persona personaA = new Persona("pedro", "perez", 20);
        Persona personaB = new Persona("ana", "blanco", 15);
```

```
        Persona personaC = new Persona("miguel", "alvarez", 50);

        List < Persona > lista = Arrays.asList(personaA, personaB,
personaC);

        lista.sort((Persona p1, Persona p2) - > {
            return p1.getEdad() - > p2.getEdad() ? 1 : -1;

        });
        for (Persona p: lista) {

            System.out.println(p.getNombre() + " ," + p.getEdad());

        }

    }

}
```

Java Lambda inferencia de tipos

Es una simplificación importante . Ahora bien las expresiones Lambda soportan más posibilidades , una de ellas es lo que denomina inferred types es decir el compilador es capaz de entender de que tipo son las variables que el método sort necesita ya que estamos trabajando con una lista genérica de Personas. Así pues podemos simplificar todavía un poco más.



Omitimos a nivel de parámetros su tipología.

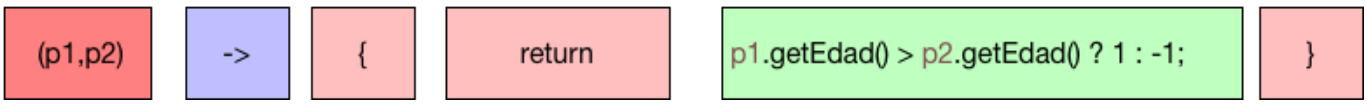
Veamos el código:

```
package com.arquitecturajava;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class Principal4 {

        public static void main(String[] args) {

                Persona personaA = new Persona("pedro", "perez", 20);
                Persona personaB = new Persona("ana", "blanco", 15);
                Persona personaC = new Persona("miguel", "alvarez",
50);

                List<Persona> lista = Arrays.asList(personaA,
personaB, personaC);

                lista.sort(( p1, p2)-> {return p1.getEdad() ->
p2.getEdad() ? 1 : -1;});
```

```
                for (Persona p : lista) {

                        System.out.println(p.getNombre() + " ," +
p.getEdad());

                }

        }

}
```

Java Lambda y bloques

Hemos optimizado el código bastante , pero todavia no es suficiente . Estamos ante un Lambda block , existe la posibilidad de compactar más el código y pasar a una Expresión Lambda . Esto únicamente ocupa una linea.



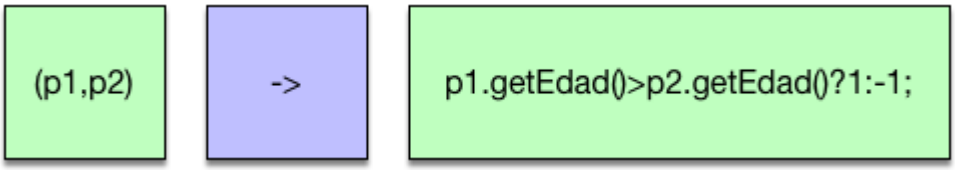
Esta sintaxis ya más directa veamosla en código:

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

public class Principal4 {

        public static void main(String[] args) {

                Persona personaA = new Persona("pedro", "perez", 20);
                Persona personaB = new Persona("ana", "blanco", 15);
                Persona personaC = new Persona("miguel", "alvarez",
```

```
50);

                List<Persona> lista = Arrays.asList(personaA,
personaB, personaC);

                lista.sort(( p1, p2)-> return p1.getEdad() ->
p2.getEdad() ? 1 : -1;);
                for (Persona p : lista) {

                        System.out.println(p.getNombre() + " ," +
p.getEdad());

                }

        }

}
```

Simplificación y operador ternario

Podemos dar un paso adicional más. El compilador es capaz de entender que al ser en una única linea se pueden eliminar los paréntesis y el return.



Ahora nos hemos quedado con la sintaxis más compacta de todas tenemos un Java Lambda puro de Java 8. La reducción del código es grande y la sintaxis es natural.

```
lista.sort(( p1, p2)->p1.getEdad() > p2.getEdad() ? 1 : -1);
```

Conclusiones

El uso de Java Lambdas , es cada día más importante para conseguir un código más compacto y mas sencillo. Tenemos poco a poco que ir mejorando nuestro conocimiento de programación funcional. Recordemos que muchos de los lenguajes modernos hacen uso intensivo de estos conceptos.

Otros artículos relacionados:

1. Lambda Expressions (I)
2. Reduce y wrappers con lambdas
3. Java 8 Lambda y forEach (II)
4. Reference method y reutilización
5. Java Stream if else y Optionals
6. Curso Java 8 Gratis ,manejo de lambdas
7. Java String to Date utilizando Java 8

Links Externos:

Expresiones Lambda