Java 8

Profesor:
Máster Carlos Carranza Blanco
Programación III
Ingeniería del Software

31/7/2023

Características introducidas en Java 8

- *Métodos default en interfaces.*
- *java.util.function.*
- ***Interfaces funcionales.***
- *Lambda.*
- *Predicate<T>*
- *java.util.Stream.*
- *java.time.*
- *Nuevos métodos en clases de uso común.*

Métodos default en interfaces

- *Si hasta ahora en una interfaz solo podíamos tener constantes y métodos abstractos (cabecera de método), ahora tendremos la posibilidad de declarar métodos con un comportamiento por defecto en una interfaz. Antes de Java 8 podíamos definir que toda clase que implementa a una interfaz debe construir, declarar, dar cuerpo a los distintos métodos que contenga la interfaz.*
- *En cambio, en Java 8, toda clase que implemente una interfaz debe declarar los distintos métodos que contenga la interfaz salvo aquellos que estén definidos como métodos por defecto. Estos métodos se caracterizan porque son métodos que están declarados en la propia interfaz y pueden ser utilizados directamente en la clase si nos interesa su comportamiento por defecto.*

Ejemplo calculadora

java.util.function.

- *Nueva API en Java 1.8 que se conoce como el paquete que contiene interfaces funcionales, es decir, interfaces que declaran funciones de uso común. Las interfaces en este paquete son anotadas como @FunctionalInterface.*
- *Por otro lado cabe destacar que dicha anotación, @FunctionalInterface, podemos utilizarla en una interfaz que queramos declararla como funcional. Estas interfaces funcionales se caracterizan porque pueden ser instanciadas por lambdas.*

Interfaces funcionales

- *Quizás el agregado más importante de Java 8 sean los lambdas. Muy relacionado a los lambdas se encuentra el concepto de "interfaces funcionales". Una interfaz es funcional si declara exactamente 1 método abstracto. Por ejemplo, java.lang.Runnable es una interfaz funcional porque define un único método abstracto.*
- *Los métodos predeterminados (marcados con "default") no son abstractos, por lo cual una interfaz funcional puede definir varios métodos predeterminados.*
- *Se agrega una nueva anotación: @FunctionalInterface, esta anotación puede usarse para declarar la intención que la interfaz es funcional. Esta anotación la utiliza el compilador, y sirve para evitar error y agregar métodos a interfaces que se desean sean funcional. Es como la anotación @Override, que denota intención.*

Interfaces funcionales…

- *Las interfaces funcionales son un potente mecanismo que proporciona Java 8 para poder pasar funciones como argumentos a los métodos. Realmente esta posibilidad ya estaba presente en versiones anteriores de Java por ejemplo con la interfaz Comparator.*
- Java 8 lo que hace es extender el número de interfaces funcionales y su aplicabilidad definiendo un conjunto de métodos cuyos argumentos de entrada son estas interfaces.

Interfaces funcionales…

- *Una interfaz funcional define "objetos" que no guardan valores como los objetos tradicionales sino que sirven para guardar "funciones". Nótese que tanto "objetos" como "funciones" están entrecomillados porque realmente no son tales pero sí es un mecanismo para que un método reciba argumentos de tipo funcional.*

Interfaces funcionales…

- *Ejemplo Comparator:*

Ejemplo. La clase Comparator que ordena objetos de tipo Vuelo por precio de menor a mayor es:

```
public class ComparadorVueloPrecio implements Comparator<Vuelo> {

        public int compare(Vuelo v1, Vuelo v2){
                return v1.getPrecio().compareTo(v2.getPrecio());
        }
}
```

Y la invocación podría ser:

```
Collections.sort(listaVuelos, new ComparadorVueloPrecio());
```

Como se puede observar el método sort recibe un argumento de tipo funcional, ya que el objeto de tipo Comparator que recibe le sirve a sort para saber cómo debe ordenar los valores de listaVuelos.

Ejemplo Comparator

Interfaces funcionales…

- *Ejemplo interface funcional:*

Estudiemos otra posible aplicación de las interfaces funcionales distinta del Comparator. Hay numerosos algoritmos que usan una condición booleana en su esquema. Uno de los más simples es el patrón contador, algoritmo que nos devuelve el número de elementos de una colección que cumplen una determinada condición: ¿Cuántos vuelos completos salen hoy? ¿Cuántos vuelos hay a Madrid esta semana? etc. Sabemos que el esquema de este algoritmo es el siguiente:

```
Esquema contador
        Para todo elemento de la colección
                Si cumple condición
                        Incrementa contador
                Finsi
        FinPara
}
```

Interfaces funcionales…

- *Ejemplo interface funcional…:*

```
public Integer contadorVuelosFecha(Fecha f){
        Integer contador=0;
        for(Vuelo v:vuelos){
                if (v.getFecha().compareTo(f)>0){
                        contador++;
                }
        }
        return contador;
}

public Integer contadorVuelosDestino(String d){
        Integer contador=0;
        for(Vuelo v:vuelos){
                if (v.getDestino().equals(d)){
                        contador++;
                }
        }
        return contador;
}
```

Interfaces funcionales…

- *Ejemplo interface funcional…:*

```
Integer cont1 = aeropuertoSVQ.contadorVuelosDestino("Madrid");
System.out.println("El número de vuelos a Madrid es "+cont1);
Integer cont2 = aeropuertoSVQ.contadorVuelosFecha(new Fecha(16,07,2014));
System.out.println("El número de vuelos a partir del 16 de julio es "+cont2);
```

Supongamos que se pudiera pasar la condición del if como argumento al método. Entonces el método contador podría generalizarse y su código sería algo así como:

```
public Integer contadorVuelosGenerico(Condicion<Vuelo> filtro[1]){
        Integer contador=0;
        For (Vuelo v:vuelos){
                if (condición de filtro sobre v[2]){
                        contador++;
                }
        }
        return contador;
}
```

[1] Con posterioridad este supuesto tipo Condicion será realmente el tipo Predicate

[2] Esta expresión será realmente filtro.test(v), ya que test es el método que implementa Predicate

Interfaces funcionales…

- *Ejemplo interface funcional…:*

De esta manera, tendríamos un método contador genérico para Aeropuerto que una vez codificado, podría invocarse con distintas expresiones booleanas y tener diferentes funcionalidades. Para poder pasar una expresión booleana como argumento necesitaríamos un tipo (una interfaz) Condicion que implementara un método que recibiera un objeto del tipo implicado (Vuelo en este caso) y devolviera un valor de tipo Boolean con la condición requerida. Esta posibilidad es la interfaz funcional Predicate que será el primer ejemplo que se estudiará en la siguiente sección.

La invocación de este método genérico sería:

```
Integer cont1 = aeropuertoSVQ.contadorVueloGenerico
                                        (v->v.getDestino.equals("Madrid")[3]);
System.out.println("El número de vuelos a Madrid es "+cont1);

Fecha f = new Fecha(16,07,2014)
Integer cont2 = aeropuertoSVQ.contadorVueloGenerico
                                        (v->v.getFecha().compareTo(f)>0[4]);
System.out.println("El número de vuelos a partir del 16 de julio es "+cont2);
```

Expresiones Lambda

- *Hay otra cuestión que también cambia en Java 8 y es la forma de proporcionar la interfaz funcional como argumento en la invocación de un método. Como hemos visto en el ejemplo anterior para usar un Comparator en Java 7, se define una clase externa con el método compare y la invocación se realiza con un objeto de la clase que o bien se crea antes o se construye directamente en la invocación.*
- *Java 8 tiene dos mecanismos más flexibles para definir las interfaces funcionales: las lambda expresiones y las referencias a métodos.*

Expresiones Lambda…

- *Una lambda expresión es una simplificación de un método en el que los parámetros de entrada y la expresión de salida se separan por un operador flecha '->'. Los argumentos de entrada se escriben entre paréntesis separados por comas. Si la interfaz sólo tiene un argumento de entrada no es necesario poner los paréntesis.*

- *Así el principio de una lambda expresión es de la forma () -> si no hay argumento de entrada, x-> si sólo tiene un argumento o (x, y) -> para dos argumentos. Normalmente no es necesario definir los tipos de los argumentos porque Java es capaz de deducirlos del contexto en el que se definen. A continuación del operador -> se debe escribir la expresión que será el valor que devuelve la interfaz que se está declarando.*

Expresiones Lambda…

- *Ejemplos Lambda:*

Ejemplo 1. Una interfaz funcional que reciba un Vuelo y devuelva su precio sería:

```
x-> x.getPrecio();
```

Ejemplo 2. Una interfaz funcional que reciba un String representado un número entero y devuelva un Integer con el valor correspondiente, se escribe:

```
x -> new Integer(x);
```

Ejemplo 3. Pongamos un ejemplo de nuevo con la interfaz Comparator. La invocación al método sort de Collections se puede hacer construyendo el Comparator mediante una lambda expresión directamente en el mismo código de la llamada:

```
Collections.sort(listaVuelos,(x,y)->x.getPrecio().compareTo(y.getPrecio()));
```

Expresiones Lambda…

- *Las lambda expresiones suelen usarse directamente en la llamada del método que tiene como argumento una interfaz funcional, pero si una misma lambda expresión se va a usar en varias ocasiones, se puede declarar mediante un identificador.*

Ejemplo. Para definir si un Vuelo está completo escribimos:

```
Predicate<Vuelo> vueloCompleto = x->
                                 x.getNumPasajeros().equals(x.getNumPlazas());
```

De esta forma, el identificador vueloCompleto puede sustituir a una interfaz Predicate en todas aquellas invocaciones a métodos que tengan un argumento de tipo Predicate.

Expresiones Lambda…

- *Si la interfaz funcional va a tener argumentos de entrada distintos de los propios de la interfaz a la que define entonces la mejor manera es definirla como si de un método se tratara.*

Ejemplo. Si se necesita definir una interfaz Predicate para Vuelo que reciba un argumento de tipo Fecha para preguntar si un Vuelo es posterior a una determinada fecha, se define:

```
Predicate<Vuelo> fechaIgual(Fecha f){
        return x -> x.getFecha().compareTo(f)>0;
}
```

Expresiones Lambda...

- *Otro ejemplo de uso de lambdas:*

Actualmente, para agregarle un listener a un botón podemos hacer:

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hola, mundo!");
    }
});
```

Con expresiones lambda de Java 8 podemos hacer exactametne lo mismo con la siguiente expresión:

```
btn.setOnAction(
    event -> System.out.println("Hola, mundo!")
);
```

Predicate<T>

- *Como se ha señalado anteriormente la interfaz Predicate implementa una condición lógica para métodos que necesiten un filtro o condición. Predicate implementa un método denominado test que devuelve un boolean a partir de un objeto de tipo T.*
- *Por tanto, el tipo Predicate se usa para clasificar los objetos de un tipo T según cumplan o no una determinada propiedad. Por ejemplo, dado un Vuelo si está completo, dado un Libro si tiene su título contiene una determinada palabra, dada una Canción si dura más de x minutos, dada un String si comienza por un determinado carácter.*

Predicate<T>…

- *Ejemplos Predicate:*

Ejemplo 1. El Predicate que devuelve si un Vuelo está completo es:

```
Predicate<Vuelo> vueloCompleto = x->
                                x.getNumPasajeros().equals(x.getNumPlazas());
```

Ejemplo 2. Si se necesita definir una condición a partir de un parámetro se le puede poner un argumento a la interfaz funcional. Por ejemplo, si necesitamos preguntar si un Vuelo es de una determinada fecha, se define:

```
Predicate<Vuelo> fechaIgual(Fecha f){
        return x -> x.getFecha().equals(f);
}
```

Predicate<T>…

- ***Métodos default:***
  - *La interfaz Predicate tiene tres métodos que implementan las operaciones lógicas: negate(), and(Predicate) y or(Predicate). Por ejemplo, si se necesita un argumento de tipo Predicate que diga si un Vuelo de una determinada fecha f está completo se escribiría:*

```
fechaIgual(f).and(vueloCompleto)
```

java.util.Stream

- *Este nuevo API provee utilidades para para permitir operaciones "estilo programación funcional" sobre flujo de valores. La forma de obtener un Stream a partir de una colección es:*

  Stream<T> stream = collection.stream();

- Los stream son parecidos a los iteradores. Los valores van "fluyendo", y se van. Los stream sólo pueden recorrerse 1 vez. Los stream también pueden ser infinitos.

java.util.Stream...

- *Un ejemplo más concreto de uso de streams:*

```
72          List<Integer> listaNumeros = new ArrayList<Integer>();
73          listaNumeros.add(4);
74          listaNumeros.add(6);
75          listaNumeros.add(5);
76
77          Stream st = listaNumeros.stream();
78
79          //Contar cuántas veces el número 5 se encuentra en una lista de números
80          System.out.println( st.filter(Predicate.isEqual(5)).count());
```

```
int sumOfWeights = blocks.stream().filter(b -> b.getColor() == RED)
                                  .mapToInt(b -> b.getWeight())
                                  .sum();
```

java.util.Stream…

- *Los stream tienen un API fluído que permite encadenar invocaciones. Hay dos tipos de invocaciones: "intermedias" y "finales". Las invocaciones intermedias permiten continuar con el flujo de las operaciones, mientras que las operaciones terminales "consumen" al stream y se tienen que invocar para finalizar la operacion. sum() es una operación terminal.*

- *En general, el uso de streams involucra:*
  - *Obtener un stream.*
  - *Realizar una o más operaciones intermedias.*
  - *Realizar una operación final.*

java.util.Stream…

- ***Método filter(Predicate):***
  - *Sirve para devolver otro Stream con sólo aquellos elementos del que invoca que cumplen el predicado. Se suele componer con métodos como count para calcular el número de elementos que cumplen una determinada condición. Por ejemplo:*

```
public Long getNumVuelosDia(Fecha f) {
        return vuelos.stream().
                filter(x->x.getFecha().equals(f)).
                count();
}
```

O cuántos vuelos completos hay:

```
public Long getNumVuelosCompletos() {
        return vuelos.stream().
                filter(x->x.getNumPasajeros().equals(x.getNumPlazas())).
                count();
}
```

java.util.Stream…

- ***Método filter(Predicate):***

Si un objeto de tipo Predicate se va a usar a menudo, debe definirse previamente y después invocarse. Por ejemplo, la condición de si un vuelo es de una determinada fecha, se definiría:

```
Predicate<Vuelo> fechaIgual(Fecha f){
        return x -> x.getFecha().equals(f);
}
```

Y después se invocaría:

```
public Long getNumVuelosDia(Fecha f){
        return vuelos.stream().
                filter(fechaIgual(f))
                count();
}
```

También se puede combinar con métodos como findFirst que devuelve el primer objeto de un stream o limit(long) que devuelve un stream limitado al número de elementos que se recibe como argumento.

java.util.Stream…

- ***Método allMatch(Predicate):***
  - *Nos devuelve un valor de cierto si todos los elementos de una colección cumplen una condición. Por ejemplo si nos preguntamos si todos los vuelos de una fecha están completos:*

```
public Boolean todosCompletos(Fecha f){
      return vuelos.stream().
            filter(fechaIgual(f)).
            allMatch(x->x.getNumPasajeros().equals(x.getNumPlazas()));
}
```

En este ejemplo el Predicate del filter se podía haber añadido al Predicate del método allMatch con un método and del tipo Predicate.

```
Predicate<Vuelo> vueloCompleto = x->
                        x.getNumPasajeros().equals(x.getNumPlazas());

public Boolean todosCompletos(Fecha f){
      return vuelos.stream().
                  allMatch(fechaIgual(f).and(vueloCompleto()));
}
```

java.util.Stream…

- ***Método anyMatch (Predicate):***
  - *Nos devuelve un valor de cierto si algún elemento de una colección cumple una condición. Por ejemplo si nos preguntamos si hay algún vuelo a un destino en una fecha concreta:*

```
public Boolean hayVueloDestinoFecha(Fecha f, String d){
      return vuelos.stream().
            anyMatch(x->x.getFecha().equals(f) &&
                                    x.getDestino().equals(d));
}
```

O si hay algún vuelo no completo un determinado día, definiendo previamente los objetos Predicate fechaIgual y vueloCompleto:

```
public Boolean hayVueloFechaNoCompleto(Fecha f){
      return vuelos.stream().
            anyMatch(fechaIgual(f).and(vueloCompleto().negate()));
}
```

java.util.Stream…

- ***Método map(Function):***
  - *Este método es realmente una familia de métodos con sus adaptaciones mapToInt (ToIntFunction), mapToLong (ToLongFunction) y mapToDouble (ToDoubleFunction) devolviendo un Stream de objetos de otro tipo obtenidos a partir del tipo base aplicando una Function.*

```
public Double getMayorOcupacion(Fecha f) {
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		mapToDouble(x->x.getNumPasajeros()/x.getNumPlazas()).
		max().
		getAsDouble();
}
```

java.util.Stream…

- ***Método map(Function):***
  - *Los métodos map también nos permiten facilitar una serie de operaciones sobre los valores devueltos. Por ejemplo si quisiéramos calcular la recaudación total de los vuelos a un determinado destino escribiríamos:*

```
public Double sumaRecaudacionDestino(String d) {
       return vuelos.stream().
              filter(x->x.getDestino().equals(d)).
              mapToDouble(x->x.getNumPasajeros()*x.getPrecio()).
              sum();
}
```

java.util.Stream…

- ***Método collect(Collector):***
  - *La funcionalidad collect es un potente mecanismo para reducir un stream en otra colección, estructura Map o dato.*
  - *Reducción a List o Set. El método collect proporciona un mecanismo para transformar un stream en distinto tipo de colecciones de datos como List y Set. Por ejemplo, si se quisiera devolver una lista con las duraciones de los vuelos a un determinado destino, se escribiría:*

```
public List<Duracion> getDuracionesDestino(String d){
	return vuelos.stream().
		filter(x->x.getDestino().equals(d)).
		map(x->x.getDuracion()).
		collect(Collectors.toList());
}
```

java.util.Stream…

- ***Método forEach(Consumer):***
  - *El método forEach sirve para llevar a cabo una acción definida por el objeto Consumer que se pasa como argumento sobre todos los elementos del stream. Por ejemplo para cambiar las duraciones de los vuelos a un determinado destino en un número determinado de minutos, se construiría un método como el siguiente:*

```
public void incrementaDuracionDestino(String d, Integer min){
      vuelos.stream().
      filter(x->x.getDestino().equals(d)).
      forEach(x->x.setDuracion(x.getDuracion().suma(new
                                                DuracionImpl(0,min))));
}
```

java.util.Stream…

- ***Método forEach(Consumer)…:***
  - *O para incrementar los precios de los vuelos un 10% a partir de un día determinado:*

```
Consumer<Vuelo> incrementaPrecio10p =  x->x.setPrecio(x.getPrecio()*1.1);

public void incrementaPrecios10pAPartirFecha(Fecha f) {
       vuelos.stream().
       filter(x->x.getFecha().compareTo(f)>0).
       forEach(incrementaPrecio10p);
}
```

```
entryList.forEach(entry -> {
  if(entry.getA() == null){
    printA();
  }
  if(entry.getB() == null){
    printB();
  }
  if(entry.getC() == null){
    printC();
  }
});
```

java.time

- *Hace rato que se mencionaba la necesidad de un nuevo API de fechas/horas/calendarios que tenga una mejor interfaz que el (viejo y odiado) java.util.Date. Para esto llega el nuevo paquete java.time que contendrá al nuevo API.*

- La integración con el API actual de fechas es muy simple a partir de nuevos métodos:
  - Date.toInstant()
  - Date.from(Instant)
  - Calendar.toInstant()

java.time…

- *Las nuevas clases más importantes:*
- ***Instant*** *es, básicamente, un timestamp numérico. Es una clase útil para realizar logs y usar para frameworks de persistencia.*
- ***LocalDate*** *sirve para almacenar una fecha sin hora. Por ejemplo, un cumpleaños como "16-12-1979".*
- ***LocalTime*** *sirve para almacenar una hora sin fecha. Por ejemplo, un horario de apertura a las "9:30".*
- ***LocalDateTime*** *sirve para almacenar una fecha con hora. Por ejemplo, puede almacenar "1979-12-16T12:30".*
- ***ZonedDateTime*** *almacena hora y fecha con información de uso horario.*

java.time...

```
//imprimir la fecha actual
LocalDate date = LocalDate.now();
System.out.printf("%s-%s-%s",date.getYear(), date.getMonthValue(), dat
e.getDayOfMonth());

//vamos a sumar 5 horas
LocalTime time = LocalTime.now();
LocalTime newTime;
newTime = time.plus(5, HOURS);

// o también
newTime = time.plusHours(5);

// o también
Period p = Period.of(5, HOURS);
newTime = time.plus(p);
```

Nuevos métodos en clases de uso común

- *En la clase String el método estático join que nos permite unir distintas cadenas de caracteres en una.*

```
public static String join(CharSequence delimiter, CharSequence... elements)
```

- En la clase Integer nos encontramos con nuevos métodos estáticos que nos permiten:
  - Dividir y comparar números sin tener en cuenta su signo:

```
public static int divideUnsigned(int dividend, int divisor)

public static int compareUnsigned(int x, int y)
```

  - Las operaciones de suma, resta, obtener el máximo y el mínimo entre dos números:

```
public static int sum(int a, int b)

public static int max(int a, int b)
```