Nuevas Tecnologías de la Programación

Tema 1: programación funcional aportada por Java

Curso 2017-18

Índice

Paradigmas de programación

Un **paradigma** de programación representa un enfoque o filosofía particular para diseñar soluciones software. Los paradigmas difieren unos de otros en:

- conceptos
- forma de abstraer los elementos involucrados en un problema
- pasos a seguir para resolverlo

Paradigmas de programación

Principales paradigmas (varias clasificaciones, diferentes nombres):

1. imperativo
2. funcional
3. lógico

Orientación a objetos tiene cabida en cualquiera de ellos. **Java** y **C++** son imperativos y orientados a objetos. **Scala** es funcional y orientado a objetos.

Paradigmas de programación

Paradigma imperativo: elementos básicos

1. modificación de variables
2. sentencias: indican cómo hacer las operaciones
3. estructuras de control

Basado en arquitectura Von Neumann: relación entre elementos de los lenguajes de programación y componentes de la arquitectura. No hay mayor nivel de abstracción en última instancia.

Paradigmas de programación

Paradigma funcional: elementos básicos

1. tratar la computación como **evaluación de funciones**
2. basado en **inmutabilidad**
3. estructuras de control
4. **funciones como valores** (pueden ser argumentos de funciones, ser devueltas por funciones, tienen tipo asignado....)

Scala pertenece a este paradigma y es orientado a objetos.

Paradigmas de programación

Paradigma lógico: elementos básicos

1. computación como forma de obtener información a partir de datos (hechos) y relaciones entre ellos (predicados)
2. **mecanismo fijo de inferencia** (backtracking)
3. basado en consultas

Índice

Conceptos básicos de programación funcional

Consideramos los básicos:

- funciones como elemento clave, tratado como cualquier otro tipo de dato (argumentos a funciones, devolución de funciones, ...)
- inmutabilidad

Funciones como elemento clave

- **función como concepto matemático**: hace corresponder cada elemento de un dominio con algún elemento de otro dominio
- **funciones como valores de cualquier otro tipo**: pueden ser argumentos, pueden ser resultado de una función, tienen tipo asociado, ...

Inmutabilidad

- **las funciones no cambian valores**: producen nuevos valores como resultado del mapeo. Imaginemos una operación de suma sobre un objeto de la clase NumeroComplejo. El método debería devolver un nuevo objeto y dejar sin modificar el objeto sobre el que se hizo la llamada
- la ausencia de cambios lleva a programas más fáciles de entender, de estudiar y de depurar
- facilidad para elaborar programas concurrentes, ya que no hay que proteger el acceso a datos compartidos (si son inmutables)

Índice

Programación funcional en Java SE8

Antes de presentar código de programación funcional en Java consideramos algunos elementos que la hacen posible:

- mejora de las interfaces, interfaces funcionales
- abstracción e inmutabilidad
- expresiones lambda
- flujos

Índice

Programación funcional en Java SE8: interfaces

Interfaces Java: permiten definir constantes y métodos. Abstraen la idea de componente funcional.

Las clases que implementan la interfaz se comprometen a aportar dicha funcionalidad.

Programación funcional en Java SE8: interfaces

Ahora pueden incorporar métodos por defecto (palabra reservada **default**).

¿Qué ventajas aporta este cambio?

Es posible definir comportamiento por defecto, que puede ser **refinado** por las clases que implementen la interfaz.

Programación funcional en Java SE8: interfaces

También pueden proporcionar métodos estáticos.

¿Qué ventajas aporta este cambio?

Permite ofrecer métodos de utilidad (antes había que ubicarlos en clases creadas a tal efecto). Estos métodos no pueden ser sobrescritos en las clases que implementan la interfaz.

Programación funcional en Java SE8: interfaces

Interfaz funcional: aquella que ofrece un único método. Ya se podían crear en versiones previas, pero ahora se convierten en elemento esencial (en relación a las **expresiones lambda**).

Programación funcional en Java SE8: interfaces

Las interfaces funcionales pueden considerarse como un **modelo abstracto de función**. Veamos algunos ejemplos (todos ellos están definidos en el paquete **java.util.function**). Hay interfaces específicas para tipos generales de funciones u operaciones: **operaciones unarias**, **operaciones binarias**, **operaciones que no producen resultados**, etc.

Interfaz funcional para operacion binaria: representa una operación sobre dos operandos del mismo tipo, produciendo un resultado (de dicho tipo):

```
Interface BinaryOperator<T>
```

El método básico que declara (heredado de otra interfaz más genérica) se denomina **apply** (al ser heredado de otra interfaz más genérica declara un tipo diferente para cada argumento y para el resultado):

```
R apply(T t, U u)
```

Programación funcional en Java SE8: interfaces

Interfaz funcional para operación con argumento único produciendo un resultado:

```
Interface Function<T,R>
```

Esta interfaz declara el método **apply** que aplica la transformación al argumento, devolviendo el resultado producido (que puede ser de otro tipo):

```
R          apply(T t)
```

Programación funcional en Java SE8: interfaces

Interfaz funcional para operacion con argumento único y que no produce resultado: representa una operación que acepta un único valor como argumento y no devuelve resultado alguno (por ejemplo, pensad en la necesidad de mostrar por pantalla):

```
Interface Consumer<T>
```

El método que que declara es **accept**, que realiza una cierta operación (por ejemplo println) sobre el objeto pasado como argumento:

```
void accept(T t)
```

Interfaz funcional para predicado lógico:

```
Interface Predicate<T>
```

Aporta el método **test**, que evalúa una condición sobre el argumento y devuelve un valor booleano:

```
boolean        test(T t)
```

Programación funcional en Java SE8: interfaces

Interfaz funcional para origen de datos:

```
Interface Supplier<T>
```

Aporta la declaración del método **get**, que ofrece un objeto del tipo adecuado (por ejemplo leyendo de un archivo):

```
T          get()
```

Programación funcional en Java SE8: interfaces

Interfaz funcional para operador unario:

```
Interface UnaryOperator<T>
```

Ofrece el método **apply**, que aplica la función (transformación) sobre el argumento y dvuelve el resultado correspondiente:

```
R apply(T t)
```

Índice

Programación funcional en Java SE8: abstracción e inmutabilidad

En el paradigma de programación imperativa se indica qué y cómo (iteración, definiendo variables de control y de acumulación....). Veamos un ejemplo:

```
.....................................................
// Iteracion sobre array valores
int suma=0;
for(int i=0; i < valores.length; i++){
   suma=suma+valores[i];
}
.....................................................
```

Características:

- **iteración externa**: es el programador quién especifica la forma de iterar y qué hacer en cada iteración
- necesidad de variables mutables: *i*, *suma*

Programación funcional en Java SE8: abstracción e inmutabilidad

¿Cuántas veces hemos programado bloques de código de este estilo?

Este código:

- es propenso a errores
- si es tan habitual, ¿por qué no incorporar esta funcionalidad a una librería? Tendríamos entonces iteración interna: podemos olvidarnos del cómo, de la forma de iterar...(como si la colección de datos ofreciese un método de suma que hiciera el trabajo evitando los detalles al usuario)

Índice

Programación funcional en Java SE8: expresiones lambda

Representan métodos anónimos (forma simplificada de interfaz funcional, similar a clase anónima, y **coincidiendo en tipo con interfaces funcionales**), pero simplificando la escritura de código. La sintaxis básica es:

```
(argumentos) -> {sentencias}
```

Ejemplos:

```
(int x, int y) -> {return x+y;}
.....................................
(x,y) -> x+y
.....................................
valor -> System.out.printf(" %d", valor);
.....................................
() -> System.out.println("Mensaje.....");
```

Programación funcional en Java SE8: expresiones lambda

Ya que cada expresión lambda se corresponde con una interfaz funcional podemos ver cuáles son las correspondientes a las expresiones de ejemplo.

Ejemplo:

```
(int x, int y) -> {return x+y;}
.....................................................
// Primera expresion lambda; si hay a la derecha un
// bloque hay que usar return
IntBinaryOperator exp1 = (int x, int y) -> {return (x + y);};

// Simplificando su escritura
IntBinaryOperator exp2 = (x, y) -> (x + y);

System.out.println("Operacion binaria enteros: "+
                   exp2.applyAsInt(7,10));
```

También podemos trabajar, por supuesto, con otros tipos de datos:

Ejemplo:

```
// Tambien podemos usar otros tipos de datos
DoubleBinaryOperator exp3 = (double x, double y) -> {
        return x + y;
};

System.out.println("Operacion binaria con dobles: "+
                   exp3.applyAsDouble(2.4,7.9));
```

Ejemplo de expresión lambda con argumento y **void** como tipo de devolución:

```
// Expresion que recibe un argumento y no devuelve nada
Consumer<Integer> exp4 = valor ->
                                System.out.println(valor);

// Forma de uso
exp4.accept(34);
```

Incluso con expresiones sin argumentos y sin devolución de resultados:

```
// Expresion lambda sin argumentos y devolviedo void
Runnable exp5 = () ->
   System.out.println("Expresion lambda sin argumentos");

// Forma de ejecucion (uso)
exp5.run();
```

Aunque pueden ejecutarse, se suelen crear para pasar como argumento a otras funciones, como veremos más adelante. El mecanismo de las expresiones lambda hace que sea muy fácil definir funciones específicas para tareas concretas de forma sencilla. A alto nivel podremos hacer cosas como las siguientes:

```
para cada elemento de  una coleccion
   transformalo usando una determinada operacion
fin para
```

Y esto se hará sin usar los mecanismos de iteración que conocemos hasta ahora.

Índice

Podemos pensar en ellos como **contenedores** de datos creados específicamente para su procesamiento mediante funciones. Serán objetos de clases que implementan la interfaz Stream o alguna de sus interfaces relacionadas y especiales para algún tipo de datos (son clases que pertenecen al paquete **java.util.stream**).

No son un almacén de datos: al finalizar su trabajo no contienen nada. Las operaciones a realizar con flujos pueden ser:

- **intermedias**: realizan alguna operación con los datos del flujo y producen un nuevo flujo
- **terminales**: generan resultados a partir de los datos (no producen nuevo flujo)

El procesamiento de datos es **perezoso**: sólo se desencadena el trabajo real cuando se produce una operación terminal y el procesamiento se detiene al obtenerse el primer resultado válido.

Veamos algunas de las operaciones **intermedias**:

- **filter**: filtra los elementos del flujo de entrada y produce un nuevo flujo como salida
- **distinct**: elimina elementos repetidos (produce nuevo flujo como salida)
- **limit**: quita el número de elementos indicado, tomados desde el principio (produce nuevo flujo)
- **map**: aplica una determinada operación a cada elemento del flujo y genera un nuevo flujo de salida (con tantos elementos como el flujo de entrada, aunque pueden ser de tipos diferentes)

Más operaciones **intermedias**:

- **flatMap**: se aplica sobre una serie de colecciones y devuelve al final una colección única (no una serie de colecciones)
- **sorted**: ordena los elementos del flujo (igual número de elementos en flujo de entrada y salida)

Operaciones **terminales**:

- forEach: itera sobre los elementos del flujo, lo que permite hacer con ellos alguna operación determinada
- **operaciones de reducción**: producen un único valor como resultado. Ejemplos: average, count, max, min, reduce. Algunas son mutables al proporcionar contenedores de objetos: collect, toArray
- **operaciones de búsqueda**: findFirst, findAny, anyMatch, allMatch

Índice

Índice

Flujo de enteros: IntStream

Clase **EjemplosIntStream**: permite trabajar con un array de valores enteros (en este caso, generados de forma aleatoria y almacenados en un dato miembro de la clase):

```
private int valores[];
```

Flujo de enteros: IntStream

El constructor de la clase es el siguiente:

```
// Constructor de la clase
public EjemplosIntStream(int numeroValores) {
    // Se reserva espacio para el array
    valores = new int[numeroValores];

    // Se generan valores aleatorios entre 0 y 100
    // para rellenar el array
    Random generador = new Random();
    for (int i = 0; i < numeroValores; i++) {
        valores[i] = generador.nextInt(101);
    }
}
```

Flujo de enteros: IntStream

Consideraremos los siguientes ejemplos de procesamiento de esta colección de datos (alguno de ellos con ambos enfoques):

- listado de valores almacenados
- obtener suma de valores
- suma mediante operación de reducción
- cálculo de la media
- filtrado y ordenación
- predicados
- operación sobre todos los valores

Flujo de enteros: IntStream

Listado de valores (paradigma imperativo):

```java
// Metodo para mostrar los valores de la forma habitual
public void listadoValoresImperativo() {
    // Se muestran los valores mediante un bucle
    for (int i = 0; i < valores.length; i++) {
        System.out.printf("%d ", valores[i]);
    }
    System.out.println();
}
```

Flujo de enteros: IntStream

Listado de valores (paradigma funcional):

```
// Metodo para mostrar los valores mediante expresiones
// lambda y flujos
public void listadoValoresFuncional() {
    // Primer paso: crear flujo
    IntStream flujo = IntStream.of(this.valores);

    // Especificar la operacion a aplicar a cada elemento
    IntConsumer operacion = valor -> System.out.printf("%d ", valor);

    // Desencadeno la iteracion sobre el flujo
    flujo.forEach(operacion);
    System.out.println();
}
```

Flujo de enteros: IntStream

Comentarios:

- of: método estático que devuelve flujo de la clase IntStream construido para procesar los datos pasados como argumento
- forEach: iteración sobre elementos del flujo, aplicando sobre cada uno de ellos la operación indicada por la expresión lambda

Flujo de enteros: IntStream

La declaración de forEach en la clase IntStream es:

```
void forEach(IntConsumer action)
```

El método que incorpora la interfaz IntComsumer tiene la forma:

```
void accept(int value)
```

Es decir, no produce ningún valor como resultado del procesamiento.

Flujo de enteros: IntStream

Otra forma más simple (y más habitual) de trabajar:

```
public void listadoValoresFuncional2() {
    IntStream.of(valores).forEach(valor ->
                                    System.out.printf("%d "));
    System.out.println();
}
```

Se crea el flujo y se itera directamante sobre él (mediante **forEach**), pasando como argumento la operación a realizar con cada elemento.

Flujo de enteros: IntStream

Y más simple aún:

```
public void listadoValoresFuncional3() {
      IntStream.of(valores).forEach(System.out::println);
      System.out.println();
   }
```

Observad la forma de llamar al método usando **::** tras **System.out**. Sólo puede hacerse si el argumento de la expresión lambda se usa directamente en la parte derecha.

Flujo de enteros: IntStream

Suma de valores con enfoque imperativo:

```
public long obtenerSumaImperativo(){
   long suma=0;

   // Iteracion externa
   for(int i=0; i < valores.length; i++){
      suma=suma+valores[i];
   }

   // Se devuelve el valor de suma
   return suma;
}
```

Necesitamos:

- variable para acumular (mutable)
- variable índice para recorrer el conjunto de datos (mutable)
- conocer la forma de determinar el número de elementos en la colección

Flujo de enteros: IntStream

Suma de valores del array mediante programación funcional:

```
// Metodo para obtener la suma de todos los valores con
// aproximacion funcional
public long obtenerSumaFuncional(){
   return(IntStream.of(valores).sum());
}
```

Todo se simplifica: no hay variables mutables en uso, no necesitamos conocer el número de elementos en la colección, ...

Flujo de enteros: IntStream

Otro enfoque: suma mediante operación de reducción:

```
// Metodo para obtener suma con reduce y expresiones
// lambda
public long obtenerSumaReduceExpresionesLambda(){
    return(IntStream.of(valores).reduce(0, (x, y) -> x + y));
}
```

Cada par de elementos se reduce a su suma:

- se inicia el proceso con 0 y el primer elemento de la colección
- el resultado se opera con el segundo elemento
- el resultado se opera con el tercer elemento
- y así sucesivamente hasta llegar al fin de la colección

Flujo de enteros: IntStream

Otros ejemplos de reducción:

```
....reduce(0,(x,y) -> x+y*y)
.....................................
....reduce(1,(x,y) -> x*y)
```

count, min, max y sum como implementaciones especializadas de operaciones de reducción

Flujo de enteros: IntStream

Suma de valores al cuadrado:

```
// Metodo para obtener la suma de los valores al cuadrado: uso de
// reduce y expresion lambda
public double obtenerSumaCuadradosFuncional(){
    return(IntStream.of(valores).reduce(0, (x, y) -> x + y * y));
}
```

Flujo de enteros: IntStream

Obtención del mínimo con programación funcional y operación de reducción:

```
public int obtenerMinimoReduce() {
 return (IntStream.of(valores).reduce(Integer.MAX_VALUE, (x, y) -> {
         if (x < y) return x;
         else return y;
 }));
}
```

Flujo de enteros: IntStream

Haciendo uso de la operación **min**:

```
public int obtenerMinimo() {
   OptionalInt min = IntStream.of(valores).min();
   return (min.orElse(-1));
}
```

La operación devuelve un objeto de la clase **OptionalInt**, que permite tratar con situaciones en que la colección esté vacía (por ejemplo). Hay varias formas de recuperar el valor almacenado:

- **orElse**: si hay valor almacenado (todo fue bien), se devuelve el valor; en caso contrario el valor pasado como argumento
- **getAsInt**: si hay valor almacenado (todo OK), se devuelve el valor; en caso contrario se lanza excepción de tipo **NoSuchElementException**

Flujo de enteros: IntStream

Trabajando sobre colección vacía se puede observar que el método de obtención del mínimo devuelve -1:

```
public int obtenerMinimoConColeccionVacia(){
    // Se crea vector vacio con enteros
    Vector<Integer> valores=new Vector();

    // Se convierte en IntStrem mediante la operacion mapToInt que
    // conviert cada elemento del vector en un dato tipo int. La
    // forma de generar el valor int resultante es extraer el valor
    // almacenado en cada objeto de la clase Integer que contuviese
    // el vector de valores
    OptionalInt min=valores.stream().mapToInt(Integer::valueOf).min();

    // Al estar vacio el metodo debe devolver -1
    return (min.orElse(-1));
}
```

Flujo de enteros: IntStream

Cálculo de la media:

```
// Metodo para obtener la media con aproximacion funcional
public double obtenerMediaFuncional(){
   return(IntStream.of(valores).average().getAsDouble());
}
```

El método **average()** de la clase **IntStream** devuelve un objeto de la clase **OptionalDouble**. El método **getAsDouble()** recupera el valor almacenado en él o genera excepción.

Flujo de enteros: IntStream

Método de filtrado de valores pares, paso a paso:

```
public void listarPares() {
  // Creacion del flujo
  IntStream flujo = IntStream.of(valores);

  // Creacion del predicado para filtrado
  IntPredicate condicionPar=(x -> x %2 == 0);

  // Filtrado usando el predicado: se genera filtro resultado
  IntStream flujoTrasFiltro = flujo.filter(condicionPar);

  // Se muestran los valores
  flujoTrasFiltro.forEach(x -> System.out.printf("%d ", x));
  System.out.println();
}
```

Flujo de enteros: IntStream

Con un enfoque más funcional:

```
public void listarPares2(){
   IntStream.of(valores).filter(x -> (x % 2) == 0).
         forEach(x -> System.out.printf("%d ",x ));
   System.out.println();
}
```

Flujo de enteros: IntStream

El resultado de una operación (como flujo) puede almacenarse mediante una colección de valores del tipo deseado:

```
public int[] obtenerPares() {
   return (IntStream.of(valores).filter(x -> x % 2 == 0).toArray());
}
```

El resultado del procesamiento se almacena en un array de enteros, mediante el uso de la función **toArray()**.

Flujo de enteros: IntStream

También pueden encadenarse fácilmente otras operaciones: eliminar repetidos y ordenación

```
public int[] obtenerParesSinRepetidos() {
   return (IntStream.of(valores).filter(x -> x % 2 == 0).
           distinct().sorted().toArray());
}
```

El orden de las operaciones es importante: primero eliminar duplicados y después hacer la ordenación. Hacerlo en orden inverso haría que la operación fuese más ineficiente.

Flujo de enteros: IntStream

Filtrado con predicados compuestos:

```
public int[] filtrarPredicados() {
    // Se crean los predicados por separado: pares
    IntPredicate par = x -> x % 2 == 0;

    // Y ahora predicado para valores mayores que 5
    IntPredicate mayor5 = x -> x > 5;

    // Se componen los predicados mediante AND
    IntPredicate total = par.and(mayor5);

    // Se devuelve el resultado del filtrado y se convierte
    // en array
    return (IntStream.of(valores).filter(total).toArray());
}
```

Flujo de enteros: IntStream

Uso de operación **map** para transformar cada dato del flujo:

```
public void ordenarParesMultiplicados(double factor) {
  // Creacion del flujo y filtrado de valores pares
  IntStream flujo1 = IntStream.of(valores).filter(x -> x % 2 == 0);

  // Se mapean los valores para multiplicarlos por el factor pasado
  // como argumento
  DoubleStream flujo2 = flujo1.mapToDouble(x -> x * factor);

  // Se ordenan y se muestran iterando con forEach
  flujo2.sorted().forEach(System.out::println);
}
```

Flujo de enteros: IntStream

Operación sobre todos los valores:

- se genera flujo
- se filtra el flujo para dejar únicamente los valores pares
- se hace corresponder a cada valor el resultado de multiplicarlo por un factor (pasado como argumento y de tipo double). Es decir, el mapeo genera una colección de un tipo distinto
- se ordenan los valores
- se muestran por pantalla

Se trocea la operación para ir viendo los tipos de datos obtenidos.

Índice

Flujo de enteros: Stream<Integer>

La clase Arrays ofrece métodos estáticos para crear flujos a partir de una colección de objetos. En los ejemplos siguientes se considera esta clase así como la posibilidad de recoger los objetos resultantes en una nueva colección para poder usarlos posteriormente

Flujo de enteros: Stream<Integer>

Creación del flujo:

```
// Dada una coleccion de valores....
Integer[] valores = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
.................................................
Stream<Integer> flujo = Arrays.stream(valores);
```

Flujo de enteros: Stream<Integer>

Ordenación de valores (se observa que se muestra el resultado de la operación, una colección final de valores):

```
// Se ordenan los valores en orden ascendente
List<Integer> lista = Arrays.stream(valores)
                .sorted()
                .collect(Collectors.toList()));

// Se muestra la lista
System.out.printf("Valores ordenados: %s%n", lista);
```

Flujo de enteros: Stream<Integer>

Filtrado de valores:

```
// Filtrado: se quedan solo los valores mayores de 4
// en una coleccion
List<Integer> mayores4 = Arrays.stream(valores)
          .filter(valor -> valor > 4)
          .collect(Collectors.toList());
```

Flujo de enteros: Stream<Integer>

Filtrado y ordenación sobre la colección generada previamente:

```
// Se muestran los valores mayores de 4 y ordenados
System.out.printf("Valores mayores que 4 y ordenados: %s%n",
        mayores4.stream().sorted().collect(Collectors.toList()));
```

Índice

Flujo de cadenas: Stream<String>

Gracias a la clase Arrays se crea un flujo para trabajar con cadenas de caracteres (objetos de la clase String). Las cadenas a procesar usan mayúsculas y minúsculas de forma indistinta. Los ejemplos considerados son:

- creación del flujo
- conversión de cadenas a mayúscula y recogida del resultado en una colección
- filtrado (sólo quedan aquellas cadenas con letras posteriores a m) y ordenación en orden ascendente y descendente

Flujo de enteros: Stream<String>

A partir de un array de objetos de la clase String la creación del flujo se hace de la siguiente forma:

```
String[] cadenas =
         {"Rojo", "Naranja", "Amarillo", "Verde", "azul",
          "indigo", "Violeta"};
.........................................
Stream<String> flujo = Arrays.stream(cadenas);
```

Flujo de enteros: Stream<String>

Conversión a mayúsculas:

```
// Se pasan a mayuscula y se almacenan en una lista
List<String> lista = flujo.map(String::toUpperCase)
                          .collect(Collectors.toList());
```

Flujo de enteros: Stream<String>

Filtrado de algunas cadenas y ordenación:

```
// las cadenas mayores que "m" (sin tener en cuenta mayusculas
// o minusculas) se ordenan de forma ascendente
List<String> lista2 = Arrays.stream(cadenas)
                            .filter(s -> s.compareToIgnoreCase("m") > 0)
                            .sorted(String.CASE_INSENSITIVE_ORDER)
                            .collect(Collectors.toList());

System.out.printf("Resultado (orden ascendente): %s%n", lista2);
```

Flujo de enteros: Stream<String>

Filtrado de algunas cadenas y ordenación (descendente):

```
// las cadenas mayores que "m" se filtran y se ordenan
// en modo descendente
List<String> lista3 = Arrays.stream(cadenas)
                .filter(s -> s.compareToIgnoreCase("m") > 0)
                .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
                .collect(Collectors.toList());
System.out.printf("Resultado (orden descendente): %s%n", lista3);
```

Índice

Flujos sobre clases propias

Debe ser posible crear flujos para tratar con este paradigma colecciones de objetos de clases propias. Veremos un ejemplo donde se trata con objetos de una clase propia: Empleado. Los datos miembro de la clase son:

- nombre (String)
- primerApellido (String)
- sueldo (double)
- departamento (String)

También ofrece los típicos métodos de acceso a datos miembro y método toString

Flujos sobre clases propias

Tareas realizadas en el ejemplo:

- creación del flujo
- listado de empleados
- filtrado y ordenación
- mapeo y agrupamiento
- conteo, suma y media de salarios

Flujos sobre clases propias

Creación del flujo partiendo de la colección:

```
Empleado[] empleados = {
   new Empleado("Juan", "Lopez", 5000, "IT"),
   new Empleado("Antonio", "Garcia", 7600, "IT"),
   new Empleado("Mateo", "Insausti", 3587.5, "Ventas"),
   new Empleado("Joaquin", "Fernandez", 4700.77, "Marketing"),
   new Empleado("Lucas", "Martinez", 6200, "IT"),
   new Empleado("Pedro", "Garcia", 3200, "Ventas"),
   new Empleado("Fernado", "Gonzalez", 4236.4, "Marketing")};

// Creacion de la lista a partir del array
List<Empleado> lista=Arrays.asList(empleados);
```

Flujos sobre clases propias

Se usa el flujo para obtener el listado:

```
// Muestra un listado de todos los empleados
System.out.println("Lista completa de empleados:");
flujo.forEach(System.out::println);
```

Flujos sobre clases propias

Filtrado y ordenación de empleados con sueldo en un determinado rango de valores: se hace uso de un predicado para definir la condición de interés:

```
Predicate<Empleado> condicion =
    empleado -> (empleado.obtenerSueldo() >= 4000 &&
                 empleado.obtenerSueldo() <= 6000);
```

Flujos sobre clases propias

Ordenación según sueldo:

```
// Muestra los empleados con salarios entre 4000 y 6000
// ordenados de forma ascendente. Sorted recibe como argumento
// un objeto de alguna clase que implemente la interfaz
// Comparator. comparing es un metodo estatico de la clase
// que devuelve un comparador que se usa la funcion indicada
// (pasada como argumento)
Comparator<Empleado> comparador =
      Comparator.comparing(Empleado::obtenerSueldo);

      System.out.printf("%nEmpleados seleccionados y ordenados:%n");
      Arrays.stream(empleados)
                .filter(condicion)
                .sorted(comparador)
                .forEach(System.out::println);
```

Flujos sobre clases propias

Búsqueda de objeto con determinada condición: primer empleado con sueldo en el rango 4000 – 6000:

```
// Muestra el primer empleado con sueldo rango entre 4000 y 6000
Empleado empleado = Arrays.stream(empleados).filter(condicion)
                                            .findFirst()
                                            .get();

System.out.printf("%nPrimer empleado sueldo $4000-$6000:%n%s%n",
                  empleado);
```

Flujos sobre clases propias

Ordenación de objetos con diferentes criterios: nombre y apellidos. Se crean funciones a tal efecto (ambas trabajan con un objeto de la clase **Empleado** y devuelve un **String**):

```
// Functiones para obtener el nombre y apellido de un empleado
Function<Empleado, String> refNombre = Empleado::obtenerNombre;

Function<Empleado, String> refPrimerApellido =
    Empleado::obtenerPrimerApellido;
```

Flujos sobre clases propias

Especificación de comparador a usar después:

```
// Comparador para comparar Empleados, por nombre y primer apellido
Comparator<Empleado> comparador =
                Comparator.comparing(refPrimerApellido)
                                  .thenComparing(refNombre);
```

Flujos sobre clases propias

Ordenación:

```
// Ordena los empleados por primer apellido + nombre
System.out.printf("%nOrden ascendente por apellido + nombre:%n");
Arrays.stream(empleados)
                  .sorted(comparador)
                  .forEach(System.out::println);
```

Flujos sobre clases propias

Mapeo para obtener la lista de apellidos, sin repeticiones:

```
// Muestra los apellidos de los empleados sin repeticiones. Se hace
// paso a paso para ver los tipos obtenidos
System.out.printf("%nApellidos sin repeticiones y ordenados:%n");
Stream<String> flujoCadenas = Arrays.stream(empleados)
                .map(Empleado::obtenerPrimerApellido);

// Se ordenan y se muestran
flujoCadenas.distinct().sorted().forEach(System.out::println);
```

Flujos sobre clases propias

Agrupamiento de empleados por departamento:

```
// Agrupa los empleados por departamentos
System.out.printf("%nEmpleados por departmento:%n");
Map<String, List<Empleado>> agrupadosPorDepartamentos
   = Arrays.stream(empleados)
     .collect(Collectors.groupingBy(Empleado::obtenerDepartamento));

// Se muestra el resultado del agrupamiento
agrupadosPorDepartamentos.
   forEach((departmento, employeesInDepartment) -> {
         System.out.println(departmento);
         employeesInDepartment.forEach(System.out::println);
   }
);
```

Flujos sobre clases propias

Conteo del número de empleados por departamento:

```
// Cuenta el numero de empleados en cada departamento
System.out.printf("%nCuenta de empleados por departamento:%n");
Map<String, Long> cuentaEmpleadosPorDepartamento
  = Arrays.stream(empleados)
    .collect(Collectors.groupingBy(Empleado::obtenerDepartamento,
             TreeMap::new, Collectors.counting()));

// Se muestra el resultado
cuentaEmpleadosPorDepartamento.forEach(
    (departamento, cuenta) -> System.out.printf(
           "%s tiene %d empleados(s)%n", departamento, cuenta));
```

Flujos sobre clases propias

Suma de sueldos:

```
// Suma de sueldos de empleados
System.out.printf("%nSuma de sueldos de empleados: %.2f%n",
                  Arrays.stream(empleados)
                     .mapToDouble(Empleado::obtenerSueldo)
                     .sum());

// Calculo de suma de sueldos mediante reduce
System.out.printf("Suma de sueldos de empleados con reduce: %.2f%n",
                  Arrays.stream(empleados)
                     .mapToDouble(Empleado::obtenerSueldo)
                     .reduce(0, (value1, value2) -> value1 + value2));
```

Flujos sobre clases propias

Cálculo de sueldo medio:

```
// Media de sueldos de empleados
System.out.printf("Sueldo medio: %.2f%n",
     lista.stream()
          .mapToDouble(Empleado::obtenerSueldo)
          .average()
          .getAsDouble());
```

Índice

El objetivo del ejemplo es:

- procesar el contenido del archivo, generando una colección de tipo clave (palabras) - valor (contador de aparición)
- mostrar finalmente el resultado del procesamiento, mostrando ordenadas alfabéticamente todas las palabras

Se irán considerando paso a paso las operaciones necesarias para el procesamiento

Creación de patrón para representar varios espacios en blanco seguidos ($\backslash s$) indica espacio en blanco y la barra adicional se usa como carácter de escape, para que no se interprete la barra como tal):

```
// Se define un patron que representa varios espacios en blanco
// (uno o mas consecutivos)
Pattern pattern = Pattern.compile("\\s+");
```

Flujos y tratamiento de archivos de texto

El procesamiento se hace siguiendo estos pasos:

- se se obtienen todas las líneas
- se transforma cada línea mediante un patrón que excluye todos los signos de puntuación, excepto el apóstrofe
- obtiene una lista con todas las palabras, usando el patrón que evita espacios en blanco
- se filtra el conjunto de palabras para evitar palabras nulas
- se construye un **TreeMap** donde la clave es la palabra y el valor el contador de ocurrencias

Flujos y tratamiento de archivos de texto

```
Map<String, Long> contadores = Files.lines(
                Paths.get("alicia.txt"), StandardCharsets.ISO_8859_1)
      .map(linea -> linea.replaceAll("(?!’)\\p{Punct}", ""))
      .flatMap(linea -> pattern.splitAsStream(linea))
      .filter(palabra -> !palabra.isEmpty())
      .collect(Collectors.groupingBy(String::toLowerCase,
                TreeMap::new, Collectors.counting()));
```

Se muestra el contenido de la colección:

```
// Se muestran las entradas de la coleccion
contadores.entrySet()
        .stream()
        .collect(Collectors.groupingBy(
                            entrada -> entrada.getKey().charAt(0),
                            TreeMap::new, Collectors.toList()))
        .forEach((letra, listaPalabras) -> {
                System.out.printf("%n%C%n", letra);
                listaPalabras.stream().forEach(
                   entrada -> System.out.printf(
                      "%13s: %d%n", entrada.getKey(),
                                    entrada.getValue()));
         });
```

Índice

Flujos para generación de números aleatorios

El objetivo del ejemplo consiste en generar valores aleatorios representando tiradas de dados (números enteros entre 1 y 6). Al final se muestra la probabilidad de aparición de cada resultado, analizando la secuencia de valores generados.

Flujos para generación de números aleatorios

Creación del generador y especificación del número de muestras a generar:

```
// Se crea el generador de numeros aleatorios
SecureRandom random = new SecureRandom();

// el valor 100_000_000 se indica de esta manera para facilitar
// la legibilidad del codigo
long muestras = 100_000_000;
```

Flujos para generación de números aleatorios

Generación de muestra y presentación de resultados:

```
// Se generan las muestras que constituyen un flujo
random.ints(muestras, 1, 7)
      .boxed()
      .collect(Collectors.groupingBy(Function.identity(),
               Collectors.counting()))
      .forEach((resultado, contador) ->
                  System.out.printf("%-10d%-10d%f%n",
                                    resultado, contador,
                                ((double) contador / muestras)));
```

Índice

Resumen

Tema de toma de contacto con:

- paradigma de programación funcional en java
- expresiones lambda
- flujos
- interfaces funcionales, referencias a métodos, ...