Tema 6 – Java 8

Programación Orientada a Objetos

Curso 2014/2015

DIS
Departamento de
Informática y Sistemas

UNIVERSIDAD DE
MURCIA

Contenido

- Motivación.
- Caso de estudio.
- Expresiones lamdba.
- Interfaces funcionales.
- Referencias a métodos y constructores.
- Streams.
- Nuevas características de las interfaces.

Motivación

- En la edad del bronce de la informática, un programador, ya jubilado, podía escribir este código en C:

```c
typedef int (*opBinaria)(int, int);

int suma(int a, int b) {  return a + b; }

int multiplicacion(int a, int b) { return a * b; }

void ejecuta(opBinaria f, int op1, int op2) {

    int resultado = f(op1, op2);
    printf("%d\n", resultado);
}

int main() {
  ejecuta(suma, 1, 2);
  ejecuta(multiplicacion, 1, 2);
  return 0;
}
```

Motivación

- En 2013, el nieto de este programador, un experto programador certificado en Java, no podía escribir código en Java como el que escribía su abuelo …
- En la Programación Orientada a Objetos, las variables sólo pueden contener datos. Así pues, no es posible declarar un método que acepte como parámetro una función.
  - Los programadores han conseguido salvar esta limitación a través de patrones (recetas), como por ejemplo, el patrón estrategia.
- Java 8 resuelve esta limitación acercando la Programación Orientada a Objetos a la **Programación Funcional**.

Caso de estudio

- **Ordenar una lista**.
- La clase `Collections` ofrece el método *static* `sort` para ordenar listas:

```
public static <T> void sort(List<T> lista,
                            Comparator<T> comparador);
```

- El método `sort` es un ejemplo de aplicación del **patrón estrategia**.
- El método `sort` es un método genérico que acepta como primer parámetro una lista (interfaz `java.util.List<T>`) y como segundo parámetro un comparador (estrategia, interfaz `java.util.Comparator<T>`).

Caso de estudio

- Tenemos una lista de objetos de la clase `Usuario`:

```
class Usuario {

    private String nombre;
    private LocalDate nacimiento;

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {

        // Propiedad calculada ...
    }

    ...
}
```

Caso de estudio

- La clase `ComparadorUsuarios` implementa un criterio de ordenación de usuarios por edad (de menor a mayor):

```
class ComparadorUsuarios implements Comparator<Usuario> {

    @Override
    public int compare(Usuario o1, Usuario o2) {

        return o1.getEdad() - o2.getEdad();
    }
}
```

- **Patrón estrategia**: esta clase está implementando una estrategia de comparación. Los objetos de esta clase son utilizados por el método `sort` para comparar objetos.

Caso de estudio

- Ordenamos una lista de usuarios (`LinkedList<Usuario>`) utilizando el criterio de ordenación anterior:

```
Collections.sort(usuarios, new ComparadorUsuarios());
```

- Observa que el método genérico se aplica correctamente:
  - La colección es de tipo `LinkedList<Usuario>` compatible con `List<Usuario>`.
  - El primer parámetro permite inferir que el tipo `<T>` es `Usuario`.
  - El comparador es un objeto compatible con la interfaz `Comparator<Usuario>`.

Caso de estudio

- **Problema**: la necesidad de nuevos criterios de ordenación conlleva la proliferación de clases que implementen comparadores.
- Para evitar tener que declarar una clase que sólo va a ser utilizada en un punto del código, Java permite crear **clases anómimas**.

```
Collections.sort(usuarios,
    new Comparator<Usuario>() {

        @Override
        public int compare(Usuario o1, Usuario o2) {

            return o1.getNombre().compareTo(o2.getNombre());
        }
    }
);
```

Expresiones lambda

- Java 8 introduce el concepto de **expresión lambda**, también conocido como *closure*.
- Una expresión lambda es un bloque de código que representa a una función.

- El uso de una expresión lambda reduce la necesidad clases anónimas:

```
Collections.sort(usuarios,
    (Usuario o1, Usuario o2) -> {
        return o2.getNombre().compareTo(o1.getNombre());
    }
);
```

Expresiones lambda

- La **sintaxis** de una expresión lambda se puede simplificar cuando se conocen los tipos de los parámetros y el bloque de código es una sola sentencia:

```
Collections.sort(usuarios,
        (o1, o2) -> o2.getEdad() - o1.getEdad());
```

- El compilador conoce que el segundo parámetro es de tipo `Comparator<Usuario>` y contiene un solo método con la siguiente signatura:

```
int compare(Usuario o1, Usuario o2);
```

- El tipo `T` se infiere del tipo de la lista (`Usuario`). Por tanto, no es necesario indicar el tipo de los parámetros.
- La segunda parte de la expresión es un valor entero, el tipo de retorno del método `compare`.

Expresiones lambda

- Las expresiones lambda tienen **acceso a las variables y atributos** del contexto del código:
  - Variables locales y parámetros, siempre que éstos no sean modificados después de la declaración de la expresión lambda.
  - Acceso a atributos, que sean visibles, sin restricción.
- **Ejemplo**:
  - Hace uso de un método `filtro` previamente declarado.

```
public static void seleccionPorRangoEdad(List<Usuario> lista,
                                         int edadInicio, int edadFin) {

  filtro(lista,
       u -> u.getEdad() >= edadInicio && u.getEdad() < edadFin);
}
```

Interfaces funcionales

- Las expresiones lambda son bloques de código (funciones) compatibles con **interfaces funcionales**.
- Una interfaz funcional es aquella que **contiene un solo método abstracto**.
- La interfaz `Comparator<T>` es una interfaz funcional:

```
@FunctionalInterface
interface Comparator<T> {

    int compare(T o1, T o2);
}
```

- La anotación `@FuntionalInterface` es opcional. Indica al compilador el propósito de la interfaz.

Interfaces funcionales

- La librería de Java incluye varias interfaces funcionales de utilidad en el paquete `java.util.function`.

- Interfaz **`Predicate<T>`**:

```
interface Predicate<T> {

    boolean test(T obj);
}
```

- Representa funciones que evalúan un objeto retornando un valor booleano. Son útiles para filtrar elementos.

Interfaces funcionales

- Interfaz **`Function<T, R>`**:

```
interface Function<T, R> {

    R apply(T obj);
}
```

  - Representa las funciones que hacen corresponder un objeto de tipo `T` con otro de tipo `R`.

- Interfaz **`Supplier<T>`**:

```
interface Supplier<T> {

    T get();
}
```

  - Representa funciones de las que podemos obtener objetos.

Interfaces funcionales

- Interfaz **`Consumer<T>`**:

```
interface Consumer<T> {

    void accept(T obj);
}
```

  - Representa procedimientos que realizan una acción con un objeto, sin retornar nada. Ejemplo: mostrarlo por la consola, almacenarlo en disco, etc.

- En general, las interfaces funcionales incluidas en la librería representan un abanico amplio de funciones (estrategias).

Referencias a métodos

- Un tipo de datos que corresponda con una interfaz funcional puede aceptar como valor tanto una expresión lambda como una referencia a un método ya existente.

- **Ejemplo 1**: referencia a **métodos *static***

```
Function<String, Integer> funcion1 = Integer::parseInt;

int valor1 = funcion1.apply("20");
```

  - Utilizamos `::` para hacer referencia al nombre del método.
  - No se especifican los tipos de los parámetros: es implícito al ser asignado a una interfaz funcional.

Referencias a métodos

- **Ejemplo 2**: referencia a **métodos de instancia**

```
Function<Usuario, String> funcion2 = Usuario::toString;

funcion2 = o -> o.toString(); // Traducción del compilador

System.out.println(
                    funcion2.apply(usuario));
```

  - Se hace referencia a un método de instancia igual que si fuera un método *static*.
  - No obstante, **el compilador traduce la referencia** al método de instancia en una expresión lambda.

Referencias a métodos

- **Ejemplo 3**: referencias a **mensajes**
  - Un mensaje es la aplicación de un método sobre un objeto concreto.

```
LinkedList<Integer> lista = new LinkedList<Integer>(
                                    Arrays.asList(1, 2, 3));

Supplier<Integer> funcion3 = lista::removeLast;

System.out.println(funcion3.get()); // 3
System.out.println(funcion3.get()); // 2
```

  - El método `removeLast` elimina el último elemento y lo retorna.

Referencias a constructores

- Los constructores son considerados como funciones que retornan un objeto del tipo de la clase.
- Por tanto, se pueden utilizar como referencias a métodos utilizando el **identificador** `new`
- **Ejemplo 4**:

```
Supplier<Collection<Integer>> funcion4 =
                                LinkedList<Integer>::new;

Collection<Integer> col1 = funcion4.get(); // lista

funcion4 = HashSet<Integer>::new;

Collection<Integer> col2 = funcion4.get(); // conjunto
```

Referencias a métodos

- Al establecer la referencia a un método o constructor no se identifican los parámetros.
- Sin embargo, Java permite **sobrecarga de métodos**.
- En caso de que una operación esté sobrecargada, se elegirá la definición que se ajuste a la interfaz funcional del tipo de la variable.

- **Ejemplo 5**: referencia al método `println`

```
Consumer<String> funcion5 = System.out::println;
```

  - En el ejemplo, se utiliza la versión `println(String)`

Streams

- Un *stream* es una secuencia de datos que son procesados en una aplicación (`java.util.Stream`).
- Las colecciones de Java ofrecen *streams* de los objetos que contienen (métodos `stream` y `parallelStream`).
- Los *streams* soportan dos **tipos de operaciones**:
  - Intermedias: filtran o transforman la secuencia de datos.
  - Terminales: finalizan el procesamiento, retornando un valor o realizando una acción sobre los datos.
- **Ejemplo**: contar los usuarios cuyo nombre comienza por "j"

```
long contador = usuarios.stream()
    .filter(u -> u.getNombre().startsWith("j"))
    .count();
```

Streams

- Método **`filter`**:
  - Filtra los elementos de la secuencia.
  - Acepta como parámetro un *predicado* (`Predicate<T>`)
  - Retorna un *stream* con aquellos elementos que cumplen el predicado.
- Método **`sorted`**:
  - Ordena la secuencia de datos según un criterio de ordenación.
  - Acepta como parámetro un *comparador* (`Comparator<T>`).
  - Retorna un *stream* con la secuencia ordenada.
- El método `sorted` tiene una versión sobrecargada sin parámetros que aplica el orden natural de los elementos.

Streams

- Método **`map`**:
  - Retorna un *stream* resultado de la correspondencia de cada elemento de la secuencia original en otro dato.
  - Acepta como parámetro una *función* (`Function<T, R>`)
- Método **`forEach`**:
  - Operación **terminal** que aplica una acción sobre cada objeto de la secuencia.
  - Acepta como parámetro un *consumidor* (`Consumer<T>`)
- Métodos **`anyMatch`**, **`noMatch`** y **`allMatch`**:
  - Operaciones **terminales** que retornan un booleano indicando si se cumple un predicado en algún (`any`), ningún (`no`) o todos (`all`) los objetos del *stream*.
  - Acepta como parámetro un *predicado* (`Predicate<T>`)

Streams

- Método **`count`**:
  - Operación **terminal** que retorna el número de objetos resultado del procesamiento de la secuencia.

- Método **`collect`**:
  - Permite transformar el resultado del procesamiento de la secuencia en una colección.
  - Acepta como parámetros métodos que transforman *streams* en colecciones:

```
Collectors.toList(),Collectors.toSet()
```

Streams – Ejemplos

- Muestra por la consola los nombres de los usuarios:

```
usuarios.stream()
    .map(u -> u.getNombre())
    .forEach(System.out::println);
```

- Obtiene una lista con los nombres de los usuarios:

```
List<String> nombres = usuarios.stream()
    .map(u -> u.getNombre())
    .collect(Collectors.toList());
```

Streams – Ejemplos

- Muestra los nombres de los usuarios en orden alfabético:

```
usuarios.stream()
    .sorted((u1, u2) ->
            u1.getNombre().compareTo(u2.getNombre()))
    .map(u -> u.getNombre())
    .forEach(System.out::println);
```

- Igual que el anterior, ordenando el flujo con los nombres:
  - La clase `String` implementa el orden natural (`Comparable`)

```
usuarios.stream()
    .map(u -> u.getNombre())
    .sorted()
    .forEach(System.out::println);
```

Streams – Ejemplos

- Consultar si la colección tiene algún usuario mayor de 20 años:

```
boolean resultado = usuarios.stream()
                         .anyMatch(u -> u.getEdad() > 20);
```

- Obtener los nombres de los usuarios que empiecen por “j”:

```
Set<String> resultado = usuarios.stream()
    .filter(u -> u.getNombre().startsWith("j"))
    .map(u -> u.getNombre())
    .collect(Collectors.toSet());
```

Programación funcional

- La introducción de expresiones lambda y referencias a métodos en el lenguaje Java ha sido motivada por:
  - Aplicaciones concurrentes (paralelismo).
  - Desarrollo dirigido por eventos (interfaces gráficas de usuario).

- El modelo de procesamiento de flujos de datos (*streams*) ofrece la posibilidad de **distribuir el procesamiento**.
- Así mismo, se traslada la tarea del recorrido de datos a la librería (iteradores internos), frente al modelo actual basado en iteradores externos (el programador realiza el recorrido).

Streams paralelos

- El procesamiento de los datos de un *stream* se realiza elemento a elemento hasta el final de la cadena de procesamiento (*pipeline*).
- **Ejemplo**:

```
usuarios.stream()
    .map(u -> u.getNombre())
    .filter(nombre -> nombre.startsWith("j"))
    .forEach(System.out::println);
```

  - Realiza la correspondencia: `Usuario` en `String`.
  - Evalúa si el objeto `String` cumple la condición.
  - Si cumple la condición, lo muestra por la consola.
- La secuencia de procesamiento se realiza para cada elemento del flujo, uno a uno.

Streams paralelos

- El procesamiento de las secuencias elemento a elemento permite **distribuir el trabajo** en varios hilos de ejecución:

```
usuarios.parallelStream()
    .map(u -> u.getNombre())
    .filter(nombre -> nombre.startsWith("j"))
    .forEach(System.out::println);
```

- Las colecciones ofrecen el método `parallelStream` para realizar el procesamiento del flujo de datos en paralelo.
- La máquina virtual aprovecha los hilos de ejecución que ofrezca el sistema operativo para procesar el flujo de datos.

Iteradores internos y externos

- Hasta Java 7 el procesamiento de las colecciones se ha realizado con un **iterador *externo***. El programador es el encargado realizar la iteración:

```
for (Usuario usuario : usuarios) {
    System.out.println(usuario);
}
```

- En Java 8, todas las colecciones ofrecen el método `forEach` (**iterador *interno***) que acepta un *consumidor* para realizar una acción sobre cada elemento de la colección.

```
usuarios.forEach(System.out::println);
```

Interfaces

- ¿Qué consecuencias tiene romper la especificación de una interfaz en Java?
  - Ejemplo: introducir un nuevo método en la interfaz `Comparator`.
- Las interfaces suponen un importante problema de mantenimiento para el código Java:
  - Añadir un nuevo método a una interfaz implica que todas las clases que previamente hayan implementado la interfaz ya no compilan.
- Los cambios en Java 8, como la introducción de iteradores internos en las colecciones, han motivado la aparición de **métodos de extensión** en interfaces.

Interfaces – métodos por defecto

- Un método de extensión o **método por defecto** es un **método implementado en una interfaz**.
- Una clase que implemente una interfaz con un método por defecto tiene dos opciones: 1) aceptar la implementación que ofrece la interfaz o 2) proporcionar otra implementación.

- Gracias a los métodos por defecto, ya no resulta problemático añadir operaciones a la interfaz:
  - Por ejemplo, en Java 8 se ha introducido la operación `forEach` en la interfaz `Collection`. Este cambio no ha afectado a las implementaciones de las colecciones de Java.

Interfaces – métodos por defecto

- **Ejemplo**:
  - Supuesto: queremos añadir el método `skip` en la interfaz `java.util.Iterator<T>`

```
default void skip() {
    if (hasNext()) next();
}
```

  - En el ejemplo anterior, el método `skip` sería equivalente a un método plantilla de una clase abstracta: un método implementado que se apoya en métodos abstractos.
- En general, un método por defecto puede contener cualquier código. No tiene la obligación de usar métodos de la interfaz.

Interfaces – métodos por defecto

- La introducción de métodos por defecto en interfaces conlleva la aparición de algunos **problemas** de los lenguajes con **herencia múltiple** (C++).
- **Ejemplo 1**:

```
interface Interfaz1 {

    default int metodo() {
        return 1;
    }
}
interface Interfaz2 {

    default int metodo() {
        return 2;
    }
}
class A implements Interfaz1, Interfaz2 { ...
```

Interfaces – métodos por defecto

- En el ejemplo anterior la clase `A` no compila:
  - Se produce un conflicto de implementaciones.

- La solución al error es redefinir el método.
  - Se podrían reutilizar las implementaciones de las interfaces utilizando “super”.

```
@Override
public int metodo() {

    return Interfaz1.super.metodo();
}
```

Interfaces – métodos por defecto

- **Ejemplo 2**:
  - En el ejemplo anterior, una de las interfaces (por ejemplo `Interfaz2`) declara el método abstracto.
  - De nuevo, se produce un error de compilación y se resolvería del mismo modo.
- **Ejemplo 3**:
  - **Herencia en rombo**: `InterfazBase` es padre de `Interfaz1` e `Interfaz2`.
  - Método `metodo` se declara en `InterfazBase`. `Interfaz1` lo implementa por defecto.
  - En este caso, no se produce ningún error en la clase `A`, puesto que la duplicidad de `metodo` tiene un origen común. Se toma como implementación la de `Interfaz1`, al ser más específica.

Interfaces – métodos por defecto

- La introducción de métodos por defecto en las interfaces ha **enriquecido la funcionalidad de las colecciones** en Java 8.
- **Ejemplo**: nuevos métodos en los mapas

```
HashMap<String, Integer> mapa =
                new HashMap<String, Integer>();

mapa.putIfAbsent("juan", 10);

int valor = mapa.getOrDefault("pedro", 0);
```

  - El método `putIfAbsent` realiza la inserción si la clave no está previamente registrada en el mapa.
  - El método `getOrDefault` retorna un valor por defecto en el caso de no existir la clave en el mapa.

Interfaces – métodos *static*

- Dado que hasta Java 7 las interfaces no podían tener código, se han desarrollado clases de utilidad para completar la funcionalidad que ofrecen las interfaces.
- **Ejemplo**: colecciones de Java y clase `Collections`.
  - La clase `Collections` ofrece métodos *static* con funcionalidad para las colecciones como ordenar listas, ofrecer vistas no modificables, etc.

- En Java 8 es posible **implementar métodos *static*** en las interfaces.
- De este modo, ya no es necesario introducir clases con funcionalidad complementaria como `Collections`.