Tema 1C – Java moderno

Unai Aguilera Irazabal

unai.aguilera@deusto.es

Grado en Ingeniería Informática

Facultad de Ingeniería – Universidad de Deusto

Características de Java

- Cada versión de Java suele incluir nuevas características tanto en el lenguaje como en el API.
- En esta asignatura vamos a hacer uso de las características y modificaciones que fueron incluidas en la versión 8 de Java.
  - Es decir, no vamos a hacer uso de nuevas características incluidas en versiones posteriores.
  - El examen y los ejercicios supondrán que se encuentra instalada dicha versión, o una versión compatible con Java 8.
- Sin embargo, para el desarrollo del proyecto, los equipos pueden hacer uso de características más modernas de Java, si así lo deciden.
  - Esto se contará como un aspecto positivo, siempre que se pueda explicar esta nueva característica y su uso dentro del proyecto.

Características principales Java 8

- En Java 8 se incluyeron las siguientes características importantes, que iremos viendo durante el curso con más detalle, ya que suponen un gran cambio respecto a las versiones anteriores:
  - Stream API
  - Expresiones lambda
  - Interfaces funcionales
  - Referencias a métodos
  - Nueva API para aspectos relacionados con tiempo
  - Mejoras en el API de concurrencia
- Pero, primero, vamos a revisar otros cambios que se han realizado en el lenguaje en Java 8 y que pueden ayudar a realizar un código más eficiente.

Cambios en Interface en Java 8

- **Métodos estáticos.** Desde Java 8 es posible añadir *métodos estáticos* en las interfaces. Estos métodos no puede ser sobrescritos por las clases que implementen la interfaz.
- **Métodos default (por defecto).** Un cambio importante es que ahora las interfaces pueden declarar *métodos con código*. Son accesibles desde las instancias de la clase que implementa la interfaz.

```
default String getType() {
   return “something”;
}
```

  - Esta funcionalidad se añadió para mantener la compatibilidad al introducir las interfaces funcionales y lambdas.

  - En cierta forma, ahora las interfaces se parecen a clases abstractas y puede existir herencia múltiple para métodos *default* heredados desde distintas interfaces.
    - **Cuidado** a la hora de realizar el diseño y utilizar estos métodos por defecto.

Características principales Java 9

- Sistema modular – Proyecto Jigsaw
  - Introduce el concepto de módulos que pueden tener dependencias entre ellos.
  - Cada módulo puede exportar un API y mantener una implementación privada.
  - Se introdujo principalmente para hacer la JVM más modular, ya que en diferentes dispositivos se pueden cargar distintas librerías módulos.
  - Para más información se puede consultar: https://openjdk.java.net/projects/jigsaw/quick-start
- Un nuevo cliente HTTP localizado en el paquete *java.net.http*.
  - Soporte para HTTP/2 y WebSocket
  - Mejora la funcionalidad *HttpURLConnection* que se había quedado antigua.
- Mejora en el API para gestionar procesos del sistema operativo (Process API).
- Consola interactiva para Java: JShell.
- Otros cambios menores

Características principales de Java 10

- **Inferencia local de tipos de variables.** Java puede ahora determinar el tipo de la variable a partir del contexto.

```
var s = “Hello, world!
```

- Cambios internos en la JVM y el funcionamiento del Garbage Collector.
- Cambios menores en colecciones y algunos métodos del API
- En este versión, los cambios han sido principalmente de eficiencia.

Características principales en Java 11

- Es la versión LTS (Long Term Support) actual, siendo la versión recomendada para instalar para desarrollo en la actualidad.
- Desde esta versión, es necesaria una licencia para hacer un uso comercial del JDK de Oracle. Sin embargo, existen otras alternativas como OpenJDK.
- Características:
  - Nuevos métodos en distintas partes del API: String, Collections, File, etc.
  - Mejora de rendimiento
  - Eliminación de paquetes del API obsoletos.
  - JavaFX ya no se incluye como parte del JDK. Debe ser descargado aparte.
- Versiones más recientes de Java (12, 13, 14, 15, 16, 17, …)
  - Se puede obtener más información en internet sobre las características que se van añadiendo https://en.wikipedia.org/wiki/Java_version_history
  - Las características nuevas tardan tiempo en aceptarse, así que ahora es suficiente con dominar muy bien las características hasta Java 8 y empezar a conocer las funcionalidades nuevas disponibles hasta Java 11.

Enumeraciones - enum

- Las enumeraciones (*enum*) en Java son clases de un tipo especial que permiten representar un grupo de constantes.

```
public static final int LUNES = 1;
public static final int MARTES = 2;
public static final int MIERCOLES = 3;
public static final int JUEVES = 4;
public static final int VIERNES = 5;
public static final int SABADO = 6;
public static final int DOMINGO = 7;
```

- La solución anterior es difícil de mantener y puede producir errores si utilizamos valores incorrectos en el código.
- Se recomienda el uso de *enum* para representar casos como el anterior.
  - Esto crea un nuevo tipo *DiaSemana* que puede ser usado como nuevo tipo en el programa.

```
public enum DiaSemana {
        Lunes,
        Martes,
        Miercoles,
        Jueves,
        Viernes,
        Sabado,
        Domingo
}
```

Usando *enum (I)*

- Una vez definido un *enum* podemos usarlo para declarar variables con dicho tipo.

```
DiaSemana dia = DiaSemana.Miercoles;
```

- Recibirlo como parámetro en un método o comparar valores

```
public static boolean isWeekend(DayOfWeek day) {
   // Comparar el valor de la variable con la constante
   return day == DayOfWeek.Saturday || day == DayOfWeek.Sunday;
}
```

- Usarlo en una sentencia *switch*

```
switch (day) {
    case Monday:    System.out.println("Es el primer día");
                    break;

    case Tuesday:   System.out.println("Es el segundo día");
                    break;

    case Wednesday: System.out.println("Es el tercer día");
                    break;

    case Thursday:  System.out.println("Es el cuarto día");
                    break;

    default:        System.out.println("Es el resto de días");
                    break;
}
```

Usando *enum (II)*

- Obtener la constante a partir de su representación en String

```
DayOfWeek newDay = DayOfWeek.valueOf("Tuesday");
```

- Iterar sobre los valores de la constante

```
for (DayOfWeek d : DayOfWeek.values()) {
   System.out.println(d);
}
```

- Obtener su representación en String

```
System.out.println(day.toString()); // El toString() puede ser implícito
```

- Obtener su ordinal (identificador numérico) en el grupo de constantes

```
System.out.println(day.ordinal());
```

- **¡Nota!** Se recomienda utilizar *enum* siempre que haya grupos de constantes relacionadas en el código.

Boxing/Unboxing

- En Java existen representaciones como objeto de los tipos primitivos.
  - Integer, Float, Boolean, etc.
- Java es capaz de convertir automáticamente entre las representaciones primitivas y los objetos. Se conoce como “boxing”.

```
Integer i = 3; // Aquí se construye un objeto automáticamente
```

- Es equivalente a realizar esta operación `Integer i = new Integer(3);`
- La operación contraria es unboxing. El valor primitivo se obtiene del objeto.
- Estas operaciones automáticas pueden tener un impacto en la eficiencia

```
int a = i;
```

```
Integer total = 0;
for (int i = 0; i < Integer.MAX_VALUE; i++) {
          total += i; // se realiza una operación de boxing cada vez
}

// el bucle más rápido si total se declara como int (tipo primitivo)
```

Argumentos variables

- En Java es posible crear un método que reciba un número de argumentos variable.
- Se declara con los puntos suspensivos (. . .).
- Solamente puede existir una declaración de parámetros variables en un método. Y debe estar al final de la lista.

```
public static int sumar(int ...values) {
   int total = 0;

   for (int v : values) {
      total += v;
   }

   return total;
}
```

- El método anterior puede recibir cualquier número de parámetros enteros. Las siguientes llamadas son válidas ahora

```
System.out.println(sumar());
System.out.println(sumar(2));
System.out.println(sumar(2, 5));
System.out.println(sumar(2, 5, 3, 7, 12));
```

static import

- En Java es posible hacer un *static import* para incluir en el programa todos los símbolos estáticos de determinada clase.
  - Evita tener que usar el nombre de la clase para hacer referencia a los métodos.

```
public class Utilidad {

        public static sumar(int a, int b) {
                . . .
        }

        public static restar(int a, int b) {
                . . .
        }

}
```

- Dada la clase anterior que contiene métodos estáticos, podemos usarlos en otra clase de la siguiente manera

```
import static Utilidad.sumar;
import static Utilidad.restar;

...

// ahora podemos llamar directamente a los métodos sin el nombre de la clase

sumar(a, b);
restar(a, b);
```

Anotaciones

- Las anotaciones en Java proporcionan información al programa pero no son parte del programa como tal.
  - Se indican con el símbolo @
  - Por ejemplo, la anotación *@Override* indica que el método anotado con ella quiere sobrescribir un método de la clase antecesora.
  - También se usan en los test unitarios (@Test).

```
@Override
public String toString() {
            . . .
}
```

- Es posible incluso crear anotaciones personalizadas
  - Este un aspecto de Java un poco más avanzado que no se estudiará en este curso.
  - Existe documentación en internet/libros para aprender su uso en más detalle.

```
public @interface MiAnotacion {
            . . .
}
```

Clases genéricas en Java

- Como programadores podemos hacer uso de la capacidad que tiene Java para definir clases que hagan uso de ***Generics***.
- Las clases definidas para utilizar Generics permiten crear "plantillas" que pueden ser parametrizadas en el momento de la instanciación de las clases. Esto permite al programador crear clases que pueden ser adaptadas y reutilizadas para distintos tipos de datos.
- En la definición de la clase utilizaremos el operador < > para indicar un tipo que todavía no se encuentra definido y que será especificado en el momento de utilizar dicha clase.

```
class Ejemplo<T> {

            private T p;

            public Ejemplo() {
            }

            public T getPropiedad() {
                        return p;
            }

            public void setPropiedad(T valor) {
                        this.p = v;
            }
}
```

```
Ejemplo<Integer> e = new Ejemplo<>();

e.setPropiedad(new Integer(3));

Integer i = e.getPropiedad();
```

```
Ejemplo<Persona> e = new Ejemplo<>();

e.setPropiedad(new Persona());

Persona p = e.getPropiedad()
```

Programación funcional en Java

- Como ya hemos visto, Java 8 supuso un cambio importante al introducir aspectos novedosos en el lenguaje y en el API de Java.
- Uno de los aspectos importantes introducidos es el soporte para programación funcional introducido en esa versión.
  - Java no es un lenguaje funcional puro (Haskell) pero incluye características que permiten utilizar este paradigma en el desarrollo de programas junto con la orientación a objetos.
- **Programación funcional:** paradigma de programación declarativa donde los programas se construyen aplicando y componiendo funciones.
  - Las funciones se tratan como "ciudadanos de primera clase", es decir, que pueden ser asignados a variables, pasadas como argumentos y devueltas por otras funciones, como cualquier otro tipo de datos.
  - Esto permite desarrollar los programas componiendo funciones.

Expresiones lambda

- Para introducir el paradigma funcional anterior, en Java 8 se añadieron las expresiones lambda:
  - Representación concisa de una función anónima que puede ser pasada como argumento, devuelta como retorno o asignada a una variable.
- Características:
  - **Función:** se llaman funciones para distinguirlas de los métodos en la POO que deben estar asociados a clases. Las expresiones lambda no están asociadas a clases.
  - **Anónima:** define la funcionalidad pero no tiene un nombre asociado, como sí que ocurre con los métodos en Java.
  - **Pasada como valor:** puede ser pasada como argumento a un método, devuelta como retorno o asignada a una variable.
  - **Concisa:** únicamente se escribe la funcionalidad, se evita escribir código innecesario como ocurre con las declaraciones de métodos en Java.

Sintaxis de las expresiones lambda (I)

- Toda expresión lambda en Java tiene 3 partes:
  - **Lista de parámetros:** cero o más parámetros separados por comas.
  - **Flecha ->:** que separa la lista de parámetros del cuerpo de la expresión.
  - **Cuerpo de la expresión lambda:** se considera el retorno de la expresión.
- Por ejemplo, esto es una expresión lambda válida:

```
x -> x + 1
```

  - Antes de la flecha están los parámetros, que en este caso es un único parámetro llamado x (el tipo del parámetro lo suele inferir Java).
  - Después de la flecha está el cuerpo que dice que esta expresión coge el valor de x recibido como parámetro y devuelve el resultado de sumarle la unidad a dicho valor.

Sintaxis de las expresiones lambda (II)

- De forma general, la sintaxis de una expresión lambda es

```
(parameter1, parameter2, parameter3, ...) -> expresión
```

- También se pueden incluir bloques de código en el cuerpo

```
(parameter1, parameter2, parameter3, ...) -> { sentencia; }
```

- Es importante recordar que la lista de parámetros puede estar vacía, lo que se representa así

```
()-> expresión
```

```
()-> { sentencia; }
```

- Y que, si es necesario se pueden especificar tipos en los parámetros

```
(String s, Integer i) -> s + " " + i
```

Interfaces funcionales

- En Java 8 también se introdujeron las interfaces funcionales (*Functional Interface*)
  - Una interfaz funcional es una *interface* de Java que especifica un único método abstracto (sin código).
  - Puede especificar uno o más métodos con código (*default*).
  - Suele ser usual marcarlas con la anotación *@FuncionalInterface*. Esto es para que el compilador nos avise si no cumple que sea funcional.
- La siguiente interfaz es una *interfaz funcional* ya que tiene un único método abstracto

```
public interface Runnable {
   void run(); // el único método sin código de la interfaz
}
```

Ejemplo de uso de una interfaz funcional

- Supongamos que hemos definido la interfaz funcional siguiente:

```
@FuncionalInterface
public interface Operacion {
   public int incrementar(int x);
}
```

- Esta interface define un método cuyo código debe ser proporcionado por cualquier clase que implemente dicha interface.
- Por ejemplo,

```
class MiOperacion implements Operacion {
   public int incrementar(int x) {
      return x + 1;
   }
}
```

Usando la implementación de la interfaz

- Si nos fijamos en la implementación de la interfaz, vemos que hemos proporcionado una funcionalidad al método abstracto de la interfaz.
- Para utilizar dicha implementación deberíamos instanciar la clase y llamar al método.
  - Supongamos, por ejemplo, que queremos aplicar la operación de incrementar al número 3 e imprimir el resultado (4) por consola.

```
Operacion op = new MiOperacion()
int v = op.incrementar(3);
System.out.println(v);
```

- Hasta ahora, no hay nada nuevo. Es el proceso usual que seguiríamos para utilizar interfaces y clases en Java.

Interfaces funcionales y lambdas

- ¿Y qué relación hay entre las interfaces funcionales y las lambdas?
  - Una expresión lambda se puede utilizar en el lugar de una interfaz funcional siempre que proporcione la implementación para el método abstracto correspondiente.
- Es decir, es una forma directa de proporcionar una implementación para un interfaz sin necesidad de realizar todos los pasos de implementación anteriores.

```
Operacion op = x -> x + 1
int v = op.incrementar(3);
System.out.println(v);
```

  - En el ejemplo estamos diciendo que "`x -> x + 1`" es la implementación directa de la interfaz *Operacion*.
  - Esto es posible porque la expresión lambda dice que recibe un parámetro (int) y devuelve un resultado de incrementar (int), lo que es compatible con el único método abstracto de la interfaz funcional Operacion.

Asignando lambdas

- Recordemos que podemos utilizar una expresión lambda en el lugar donde existe una interfaz funcional compatible.
  - Ya hemos visto como asignarla a una referencia compatible.
  - Pero podemos pasarla también como parámetro.

```
public void hacerAlgo(int x, Operacion op) {
   int v = op.incrementar(x);
   System.out.println(v);
}
```

```
hacerAlgo(3, x -> x + 1);
```

  - O como retorno de una función.

```
public Operacion getOperacion() {
    return x -> x + 1;
}
```

```
hacerAlgo(3, getOperacion());
```

Reutilizando interfaces funcionales

- Hemos visto que podemos definir nuestras propias interfaces funcionales en Java.
- Pero en el API de Java se han definido ya diferentes interfaces funcionales que pueden ser utilizadas junto con expresiones lambda directamente.
  - Tenemos un listado de interfaces funcionales ya definidas en https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html
- Esto permite reducir la cantidad de código que se tienen que escribir cada vez que se quiere proporcionar la implementación de una interfaz existente.
  - Por ejemplo, cuando utilizamos *Runnable*, *Listener* en Swing, *Comparator*, etc.
  - Iremos viendo ejemplos de esto durante la asignatura.
- Las expresiones lambda son una herramienta potente a partir de Java 8 y permiten utilizar aspectos más avanzados como la programación funcional y la programación reactiva.
  - Estos aspectos no van a ser cubiertos directamente en la asignatura.

Práctica IE – Parte I

- Descarga el código disponible en ALUD y complétalo.
- El objetivo es utilizar y entender el uso de las expresiones lambda y de las interfaces funcionales.
- Recuerda que las expresiones lambda están relacionadas con las interfaces de Java.
  - Por lo que si no sabes solucionar algún caso piensa como lo harías implementado una interfaz y después como puedes transformar eso en el uso de una interfaz lambda.
- Piensa si en algunos casos puedes utilizar referencias a métodos para substituir a las expresiones lambda que has escrito.

Práctica IE – Parte II

- Ya hemos entendido un poco mejor cómo las expresiones lambda se pueden utilizar en el lugar de las interfaces y clases anónimas.
- Completa ahora el código de la Parte IV de la Práctica IE, donde aprenderás como utilizar este conocimiento para pasar expresiones lambda a métodos que reciban interfaces compatibles.
- Como en las prácticas anteriores piensa poco a poco en cada paso.
- El objetivo es entender lo que estás haciendo, no completar el código hasta que compile o funcione sin entenderlo.

Práctica IE – Parte III

- Implementa un programa que pida dos números enteros al usuario y aplique las operaciones de +, -, * y / a esos números, sacando por pantalla el resultado de cada una de ellas.
  - Debes hacer uso de expresiones lambda para construir una lista con las distintas operaciones. Cada operación la puedes representar como una expresión lambda compatible con la interfaz funcional *BiFunction<T,U,R>*.
  - A la hora de obtener los resultados, debes aplicar en un bucle cada operación (cada expresión lambda) a los operandos introducidos por el usuario.
- Intenta ahora crear una versión alternativa del programa para que imprima por pantalla la operación como un String.

```
5 * 3 = 15,00
5 + 3 = 8,00
5 - 3 = 2,00
5 / 3 = 1,67
```

  - Ahora te será útil construir un mapa que relacione cada operador con su expresión lambda.

Optional<T>

- Es una clase que sirve como contenedor para valores que pueden ser *null*. Se ha añadido con la finalidad de *eliminar* los errores de tipo *NullPointerException*.
- La idea es que los métodos no deben devolver nunca referencias directamente, sino instancias de esta clase contenedora a la que se preguntará si contiene o no un valor nulo.
  - Supongamos el siguiente ejemplo, un método que obtiene de la base de datos el nombre de un usuario a partir de un id. Si no podemos encontrar el usuario en la base de datos, el método devolverá un String nulo.

```
String getNombre(String id) {
  . . .
}
```

  - Esto tiene el problema de que el código que use a este método debe comprobar que el String devuelto no sea nulo, lo que puede ser olvidado en ocasiones y dar como resultado un *NullPointerException* en ejecución.

Creación de Optional<T>

- Utilizando Optional<T> el método anterior debería ser modificado para que devuelva una instancia de esta clase parametrizada para utilizar Strings.

```
Optional<String> getNombre(String id) {
  . . .
  return optional;
}
```

- Internamente, el método podrá devolver diferentes valores
  - **Valor vacío:** el contenedor *Optional* devuelto no tiene nada dentro.

```
Optional<String> optional = Optional.empty();
```

  - **Valor no-nulo:** el contenedor *Optional* devuelto tiene un valor dentro.

```
Optional<String> optional = Optional.of(str);
```

Uso de Optional<T>

- Cuando un método, como el ejemplo anterior, devuelve un valor de tipo *Optional<T>*, podemos realizar comprobaciones para determinar de forma segura si contiene o no un valor no-nulo.

```
Optional<String> optional = getNombre(id);
```

  - **Comprobar si contiene un valor no nulo.** El método *isPresent()* devuelve true o false si el valor interno es distinto de o nulo, respectivamente.

```
if (optional.isPresent()) { ... }
```

  - **Obtener el valor no-nulo contenido.** El método get() obtiene el valor contenido en el objeto *Optional<T>*, solamente si es no-nulo.

```
String s = optional.get();
```

  - **Intentar obtener el valor no-nulo o devolver otro valor si es nulo.** Si el optional contiene nulo, se devuelve el String vacío.

```
String s = optional.orElse("");
```

Práctica IE – Parte IV

- En esta práctica vamos a ver cómo se puede utilizar la nueva clase Optional para representar retornos nulos de funciones.
- Implementa la siguiente funcionalidad:
  - Crea una lista de Strings e iniciala con los valores que quieras.
  - Añade un método *buscar(…)* que reciba un String como parámetro y devuelva otro String como resultado.
    - Este método debe comprobar uno a uno los Strings de la lista y devolver el primer String de la lista cuyos primeros caracteres sean iguales que los pasados como parámetros al método buscar.
    - Como puede ser que el método no encuentre un String compatible, haz que el método devuelva un *Optional<String>* para poder representar correctamente estos casos.
  - Usa el método implementado para comprobando los distintos casos posibles del resultado de Optional<>.