PRO

UT7: Expresiones Lambda y conceptos avanzados de Java

José Antonio
Benítez Chacón

Índice

1. Interfaces y clases abstractas

Una **interfaz** en Java es un conjunto de métodos abstractos que otras clases pueden implementar. No puede contener implementaciones de métodos (salvo default y static) ni atributos de instancia (solo constantes: public static final).

Las interfaces definen un **contrato** que las clases que las implementan deben cumplir, favoreciendo la POO.

```
interface Vehiculo {
 void conducir();
}

class Coche implements Vehiculo {
 public void conducir() {
   System.out.println("Conduciendo coche");
 }
}
```

1. Interfaces y clases abstractas

Una **clase abstracta** es una clase que no puede instanciarse directamente. Puede tener métodos abstractos (sin implementación) y métodos concretos. Sirven como base para otras clases. Las clases que la extiendan deben implementar los métodos abstractos.

```
abstract class Figura {
 abstract double area(); // Método abstracto

 void mostrarInfo() {
   System.out.println("Esto es una figura");
 }
}

class Circulo extends Figura {
 double radio;

 Circulo(double radio) { this.radio = radio; }

 double area() { return Math.PI * radio * radio; }
}
```

1. Interfaces y clases abstractas

Diferencias entre **interfaces** y **clases abstractas**:

Característica	Interfaz	Clase Abstracta
Métodos	Solo declaraciones (excepto default y static)	Puede tener métodos con o sin implementación
Atributos	Solo constantes (public static final)	Puede tener variables de instancia (atributos)
Herencia	Una clase puede implementar varias interfaces	Una clase solo puede extender una clase abstracta
Instanciación	No se pueden instanciar directamente	No se pueden instanciar directamente
Uso típico	Contrato para clases no relacionadas	Base para clases relacionadas

2. Expresiones Lambda en Java

Una **expresión lambda** es una función anónima que permite escribir código de forma clara y concisa (se introdujeron en Java 8).

Sintaxis básica de una expresión lambda:

```
(parametros) -> { cuerpo de la expresión }
```

Ejemplos:

1. Lambda sin parámetros:

```
() -> System.out.println("Hola, mundo");
```

2. Lambda con un solo parámetro:

```
nombre -> System.out.println("Hola, " + nombre);
```

2. Expresiones Lambda en Java

3. Lambda con varios parámetros y una única línea de código:

```
(a, b) -> a + b;
```

4. Lambda con varios parámetros y un bloque de código:

```
(a, b) -> {
   int resultado = a + b;
   System.out.println("Llamada lambda para cálculo de una suma a + b");
   return resultado;
};
```

2. Expresiones Lambda en Java

Antes de Java 8, se usaban clases anónimas (se definían al crear el objeto) para este tipo de pequeñas funcionalidades.

Ejemplo con clase anónima

```
interface Operacion {
   int calcular(int a, int b);
}

public class Main {
   public static void main(String[] args) {
       Operacion suma = new Operacion() {
           @Override
           public int calcular(int a, int b) {
               return a + b;
           }
       };
       System.out.println(suma.calcular(5, 3));
   }
}
```

Ejemplo con expresión lambda

```
interface Operacion {
   int calcular(int a, int b);
}

public class Main {
   public static void main(String[] args) {
       Operacion suma = (a, b) -> a + b;
       System.out.println(suma.calcular(5, 3));
   }
}
```

2. Expresiones Lambda en Java

Uso de lambdas con interfaces funcionales

Una **interfaz funcional** es una interfaz que tiene un solo método abstracto. Estas interfaces son ideales para las expresiones lambda. Java proporciona varias interfaces funcionales en el paquete java.util.function.

Interfaz	Método	Descripción
Predicate<T>	boolean test(T t)	Evalúa una condición sobre un objeto y devuelve true o false.
Function<T,R>	R apply(T t)	Recibe un argumento y devuelve un resultado.
Consumer<T>	void accept(T t)	Recibe un argumento y no devuelve nada (operaciones sin retorno).
Supplier<T>	T get()	No recibe argumentos, pero devuelve un resultado.

2. Expresiones Lambda en Java

Uso de lambdas con interfaces funcionales

Ejemplo de su uso:

```
import java.util.function.*;

public class MainLambda {
 public static void main(String[] args) {
   // Predicate: Comprueba si un número es par
   Predicate<Integer> esPar = num -> num % 2 == 0;
   System.out.println(esPar.test(10));

   // Function: Convierte un número a su representación en cadena
   Function<Integer, String> convertirAString = num -> "Número: " + num;
   System.out.println(convertirAString.apply(5));

   // Consumer: Imprime un mensaje
   Consumer<String> imprimir = mensaje -> System.out.println("Mensaje: " + mensaje);
   imprimir.accept("Hola");

   // Supplier: Genera un número aleatorio
   Supplier<Double> generarAleatorio = () -> Math.random();
   System.out.println(generarAleatorio.get());
 }
}
```

2. Expresiones Lambda en Java

Referencia a métodos y constructores

Las referencias a métodos (::) permiten reutilizar métodos ya existentes como si fueran lambdas.

1) Referencia a un método estático

```
import java.util.function.Function;

public class ReferenciaMetodo {
 static int cuadrado(int x) {
   return x * x;
 }

 public static void main(String[] args) {
   Function<Integer, Integer> f = ReferenciaMetodo::cuadrado;
   System.out.println(f.apply(5));
 }
}
```

2. Expresiones Lambda en Java

Referencia a métodos y constructores

2) Referencia a un método de instancia

```
class Persona {
 void decirHola() {
   System.out.println("¡Hola!");
 }
}

public class ReferenciaMetodoInstancia {
 public static void main(String[] args) {
   Persona p = new Persona();
   Runnable r = p::decirHola;
   r.run();
 }
}
```

2. Expresiones Lambda en Java

Referencia a métodos y constructores

3) Referencia a un constructor

```
class Persona3 {
 private String nombre;
 Persona3() {
   this.nombre = "Sin nombre";
 }
 Persona3(String nombre) {
   this.nombre = nombre;
 }
 public String getNombre() {return nombre;}
}

public class ReferenciaConstructor {
 public static void main(String[] args) {
   Supplier<Persona3> nuevaPersona = Persona3::new;
   Persona3 p = nuevaPersona.get();
   Function<String, Persona3> nuevaPersona2 = Persona3::new;
   Persona3 p2 = nuevaPersona2.apply("Ana");
   System.out.println(p.getNombre()+", "+p2.getNombre());
 }
}
```

2. Expresiones Lambda en Java

Streams

Los **Streams** en Java son una API que permite trabajar con colecciones de datos de forma funcional, aplicando operaciones como **filter**, **map** o **reduce**.

Un **Stream** es una secuencia de datos que permite realizar operaciones en ellos de forma funcional. No almacena datos, sino que actúa sobre una fuente de datos.

Características:

- No modifican la colección original (crean una nueva con los resultados)
- Pueden ser secuenciales (**stream()**) o paralelos (**parallelStream()**, mejorando rendimiento para grandes volúmenes de datos).
- Usan operaciones intermedias (devuelven otro Stream) y terminales (devuelven un valor o colección)

2. Expresiones Lambda en Java

Streams

Creación desde una lista

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class EjemploStreams {
   public static void main(String[] args) {
       List<String> nombres = Arrays.asList("Ana", "Juan", "Pedro", "María");
       Stream<String> stream = nombres.stream();
       stream.forEach(System.out::println);
   }
}
```

Creación desde un array

```
String[] palabras = {"Java", "Python", "C++"};
Stream<String> stream2 = Arrays.stream(palabras);
stream2.forEach(System.out::println);
```

2. Expresiones Lambda en Java

Streams

Creación desde un conjunto

```
Set<Integer> numeros = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));
Stream<Integer> stream3 = numeros.stream();
```

Creación desde un rango de números

```
IntStream.range(1, 6).forEach(System.out::print);
```

Creación desde un archivo

```
try {
   Stream<String> stream4 = Files. lines(Paths.get("src/main/java/com/ut7/dam1/archivo.txt" ));
   stream4.forEach(System. out::println);
} catch (IOException e) {
   e.printStackTrace();
}
```

2. Expresiones Lambda en Java

Operaciones con Streams

Operaciones intermedias (devuelven un nuevo Stream)

Operación	Descripción
filter()	Filtra elementos según condición
map()	Transforma cada elemento del Stream
flatMap()	Convierte un elemento en una secuencia (Stream) y la aplana en un solo Stream
distinct()	Elimina duplicados
sorted()	Ordena los elementos
limit(n)	Toma los primeros n elementos

Operaciones terminales (devuelven un valor o colección)

Operación	Descripción
collect(Collectors.toList())	Convierte el Stream en List
forEach()	Itera sobre los elementos
count()	Cuenta los elementos
reduce()	Reduce los elementos a un solo valor
anyMatch() allMatch() noneMatch()	Verifica si se cumple una condición

2. Expresiones Lambda en Java

Operaciones con Streams

Ejemplo 1: Filtro de nombres que empiezan por “A”

```
List<String> lNombres = Arrays.asList("Ana", "Juan", "Alberto",
"Pedro");
List<String> nombresConA = lNombres.stream()
        .filter(n -> n.startsWith("A"))
        .collect(Collectors.toList());

System.out.println(nombresConA);
```

Ejemplo 2: Convertir a mayúsculas

```
List<String> nombresMayus = lNombres.stream()
        .map(String::toUpperCase)
        .collect(Collectors.toList());

System.out.println(nombresMayus);
```

2. Expresiones Lambda en Java

Operaciones con Streams

Ejemplo 3: Ordenación por edad

```
ist<Persona> personas = Arrays. asList(
        new Persona("Ana", 25),
        new Persona("Juan", 20),
        new Persona("Pedro", 30)
);

List<Persona> ordenadas = personas.stream()
        .sorted(Comparator. comparingInt(p -> p.getEdad()))
        .collect(Collectors. toList());

System.out.println(ordenadas);
```

NOTA: Enlace a documentación oficial de Collectors:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

2. Expresiones Lambda en Java

Operaciones con Streams

Ejemplo 4: Obtener una lista con las palabras de una lista de frases

```
List<String> frases = Arrays.asList("hola mundo", "soy una frase en Java",
"trabajemos con Stream");

List<String> lPalabras = frases.stream()
        .flatMap(frase -> Arrays.stream(frase.split(" ")))
        .collect(Collectors.toList());

System.out.println(lPalabras);
```

NOTA: Enlace a documentación oficial de Collectors:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

3. Aserciones

Una **aserción** en Java es una instrucción que permite verificar suposiciones en tiempo de ejecución. Se usan para comprobar que ciertas condiciones se cumplen durante la ejecución del programa (y si no se cumplen lanzan un **AssertionError**).

Sintaxis básica

```
assert condicion;
```

o

```
assert condicion : mensajeDeError;
```

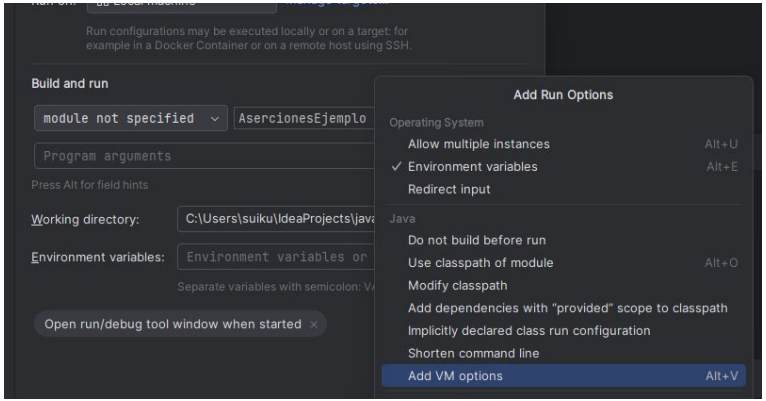
Por ejemplo:

```
public class AsercionesEjemplo {
 public static void main(String[] args) {
   int edad = -5;

   assert edad >= 0 : "La edad no puede ser negativa";

   System.out.println("Edad válida: " + edad);
 }
}
```

3. Aserciones

Por defecto, las aserciones no están activas, para que surtan efecto hay que ejecutar el programa por consola con la opción -ea o, si se lanza desde un IDE (como IntelliJ), hay que configurar las opciones de máquina virtual de Java con ese parámetro.
En IntelliJ, por ejemplo: Run > Run… > Edit Configurations…

Then Modify options > Add VM options



Salida sin activar aserciones

```
Edad válida: -5
```

Salida activando aserciones

```
Exception in thread "main" java.lang.AssertionError Create breakpoint : La edad no puede ser negativa
    at AsercionesEjemplo.main(AsercionesEjemplo.java:5)
```

3. Aserciones

¿Cuándo usar aserciones?

- Durante el desarrollo y pruebas, para detectar errores.
- Para verificar estados internos que no deberían fallar en condiciones normales.
- No se recomienda en código de producción para validar entradas de usuario, ya que se pueden desactivar en tiempo de ejecución. (para errores en producción es mejor utilizar excepciones).

4. Reflexión

La **reflexión** es una característica de Java que permite inspeccionar y manipular clases, métodos, atributos y anotaciones **en tiempo de ejecución**.

¿Para qué sirve la reflexión?

- Obtener de información sobre una clase en ejecución
- Instanciar objetos sin conocer la clase en tiempo de compilación
- Acceder y modificar atributos privados
- Llamar a métodos sin conocerlos en tiempo de compilación
- Leer anotaciones y utilizarlas dinámicamente

4. Reflexión

Ejemplo 1: Obtener información de una clase

Podemos usar **Class<?>** para obtener detalles de una clase en tiempo de ejecución.

```
class Persona4 {
 private String nombre;
 public int edad;

 public Persona4(String nombre, int edad) {
   this.nombre = nombre;
   this.edad = edad;
 }

 public void saludar() {
   System.out.println("Hola, soy " + nombre);
 }
}

public class ReflexionEjemplo1 {
 public static void main(String[] args) {
   Class<?> clase = Persona4.class;

   System.out.println("Nombre de la clase: " + clase.getName());
   System.out.println("Atributos públicos:");
   for (var campo : clase.getFields()) {
     System.out.println(" - " + campo.getName());
   }
 }
}
```

```
Nombre de la clase: Persona4
Atributos públicos:
 - edad
```

4. Reflexión

Ejemplo 2: Crear una instancia dinámicamente

Esto es útil en frameworks como **Spring**, donde los objetos se crean sin instanciarlos manualmente.

```
public class ReflexionInstancia {
 public static void main(String[] args) throws Exception {
   Class<?> clase = Class.forName("Persona4");
   Object obj = clase.getDeclaredConstructor(String.class, int.class).newInstance("Juan", 30);
   System.out.println(obj);
 }
}
```

```
Persona4@4e50df2e
```

4. Reflexión

Ejemplo 3: Acceder y modificar atributos privados

Esto lo utilizan ORMs como **Hibernate** para mapear objetos a bases de datos.

```
import java.lang.reflect.Field;

public class ReflexionAtributos {
 public static void main(String[] args) throws Exception {
   Persona4 p = new Persona4("Ana", 25);
   Field campoNombre = Persona4.class.getDeclaredField("nombre");
   campoNombre.setAccessible(true); // Habilita el acceso a atributos privados
   campoNombre.set(p, "Laura");

   System.out.println("Nombre modificado: " + campoNombre.get(p));
 }
}
```

```
Nombre modificado: Laura
```

4. Reflexión

Ejemplo 3: Acceder y modificar atributos privados

Esto lo utilizan ORMs como **Hibernate** para mapear objetos a bases de datos.

```
import java.lang.reflect.Field;

public class ReflexionAtributos {
 public static void main(String[] args) throws Exception {
   Persona4 p = new Persona4("Ana", 25);
   Field campoNombre = Persona4.class.getDeclaredField("nombre");
   campoNombre.setAccessible(true); // Habilita el acceso a atributos privados
   campoNombre.set(p, "Laura");

   System.out.println("Nombre modificado: " + campoNombre.get(p));
 }
}
```

```
Nombre modificado: Laura
```

4. Reflexión

Ejemplo 4: Llamar métodos dinámicamente

Esto se usa en APIs como **JavaBeans** o en pruebas unitarias dinámicas:

```
import java.lang.reflect.Method;

public class ReflexionMetodos {
 public static void main(String[] args) throws Exception {
   Persona4 p = new Persona4("Carlos", 40);
   Method metodoSaludar = Persona4.class.getMethod("saludar");
   metodoSaludar.invoke(p);  // Llama al método saludar()
 }
}
```

```
Hola, soy Carlos
```

5. Anotaciones

Las **anotaciones** en Java son metadatos que se pueden agregar al código para proporcionar información adicional a compiladores, herramientas o incluso en tiempo de ejecución. No afectan directamente a la lógica del programa, pero pueden ser utilizadas por frameworks y librerías. Se caracterizan por ir precedidos por una @.

Java incluye anotaciones predefinidas como:

@Override: indica que un método sobrescribe otro.
@Deprecated: Marca un método como obsoleto.
@SuppressWarnings: Suprime advertencias del compilador.

Pero también se pueden definir nuevas anotaciones para utilizar en código propio.