JAVA 8

ADOLFO SANZ DE DIEGO

PRONOIDE

2 ACERCA DE

2.1 AUTOR

- **Adolfo Sanz De Diego**
  - Blog: asanzdiego.com
  - Correo: asanzdiego@gmail.com
  - GitHub: github.com/asanzdiego
  - Twitter: twitter.com/asanzdiego
  - LinkedIn: in/asanzdiego
  - SlideShare: slideshare.net/asanzdiego

2.2 LICENCIA

- **Esta obra está bajo una licencia:**
  - Creative Commons Reconocimiento-CompartirIgual 3.0

3 INTERFACES

3.1 DEFAULT

- Para declarar **métodos por defecto en interfaces**.
- Este método es sobreescribible por las implementaciones.
- Si una **clase hereda 2 interfaces con el mismo método default se le obliga a sobreescribir**.

```java
interface EjemploDefault{
    default void porDefecto(){
        System.out.println("Este por defecto");
    }
}
public class EjemploDefaultApp implements EjemploDefault{
    public static void main(String[] args) {
        EjemploDefaultApp obj = new EjemploDefaultApp();
        obj.porDefecto();
    }
}
```

3.2 STATIC

- Ahora también podemos declarar **métodos estáticos en una interfaces**.

```
interface EjemploConStatic {
    static void esteEsStatic(String mensaje){
       System.out.println("Pues estatico desde interface con : " + mensaje);
    }
}
public class EjemploConStaticApp implements EjemploStatic {
    public static void main(String[] args) {
        EjemploConStatic.esteEsStatic("un par");
    }
}
```

4 PROGRAMACIÓN FUNCIONAL

4.1 VENTAJAS

- En vez de escribir "el cómo" el código escribe "**el qué**".
- Los programas son **más cortos**.
- **No tiene efectos colaterales**.
- Se **paraleliza** mejor.

4.2 DESVENTAJAS

- **Difícil de leer** para programadores no habituados.
- **Curva de aprendizaje** alta.

5 INTERFACES FUNCIONALES

5.1 CARACTERÍSTICAS

- Son interfaces con un método abstracto y con la anotación **@FunctionalInterface**.
- Puede poseer métodos estáticos o por defecto, pero **solo un método abstracto**.
- Se le conoce como interfaces **SAM o Simple Abstract Method**.
- Permite una aproximación a la **programación funcional** en Java.

5.2 EJEMPLO SENCILLO

```java
@FunctionalInterface // No es obligatorio ponerlo
interface Cultivable{
    void cultivar(String cultivo);
}
public class InterfazFuncional implements Cultivable{
    public void cultivar(String cultivo){
        System.out.println("cultivando " + cultivo);
    }
    public static void main(String[] args) {
        InterfazFuncional funcional = new InterfazFuncional();
        funcional.cultivar("patatas");
    }
}
```

5.3 EJEMPLO COMPLEJO

```java
interface PorDefecto {
    default void porDefecto() { System.out.println("Por defecto"); }
}
@FunctionalInterface
interface MiInterfazFuncional extends PorDefecto {
    void funciona(String mensaje);
}
public class EjemploComplejo implements MiInterfazFuncional {
    public void funciona(String mensaje) { System.out.println(mensaje);}

    public static void main(String[] args) {
        EjemploComplejo funcional = new EjemploComplejo();
        funcional.funciona("Y es la leche!");
        funcional.porDefecto();
    }
}
```

5.4 FUNCIONES PREDEFINIDAS

- Java provee de una serie de interfaces funcionales muy útiles:

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

5.5 CONSUMERS

- Reciben uno o más parámetros y no devuelven nada.

```
(T, U)->{}
```

5.6 SUPPLIERS

- No reciben parámetros y devuelven un resultado.

```
()->{return R}
```

5.7 FUNCTIONS

- Reciben uno o más parámetros y retornan un resultado.

```
(T, U)->{return R}
```

5.8 PREDICATES

- Reciben uno o más parámetros y retornan un boolean.

```
(T, U)->{return B}
```

5.9 OPERATORS

- Reciben uno o más parámetros del mismo tipo y retornan un resultado del mismo tipo.

```
(T, T)->{return T}
```

6 EXPRESIONES LAMBDA

6.1 CARACTERÍSTICAS

- Son **azúcar sintáctico** para las antiguas Clases Abstractas.
- Eliminan código inútil y repetitivo.
- Son **implementaciones de interfaces funcionales**.

6.2 SINTAXIS

- **(args) : Lista de argumentos entre paréntesis**, que puede estar vacía o no.
- **-> : Flecha** que enlaza la lista de argumentos con el cuerpo de la expresión.
- **{body} : Cuerpo entre llaves**, que contiene el código de la expresión lambda.

```
(args)->{body}
```

6.3 COMENTARIOS

- En la mayoría de los casos **el tipo de los argumentos es opcional** y el compilador lo resuelve por inferencia.
- Los **parantesis son opcionales** si hay uno (y solo un) argumento.
- Las **llaves opcionales** si hay una (y solo una) línea.
- La **palabra reservada return es opional** si hay una (y solo una) línea.

6.4 EJEMPLO SIN LAMBDA

```
interface Buscar{
    public void buscar();
}
public class SinExpresionLambda {
    public static void main(String[] args) {
        int encontrados=3;
        Buscar b = new Buscar() {
            public void buscar() {
                System.out.println("encontrado "+encontrados);
            }
        };
        b.buscar();
    }
}
```

6.5 EJEMPLO CON LAMBDA

```java
@FunctionalInterface
interface Buscar{
    public void buscar();
}

public class ConExpresionLambda {
    public static void main(String[] args) {
        int encontrados=3;
        Buscar b=()->System.out.println("encontrado "+encontrados);
        b.buscar();
    }
}
```

7 REFERENCIA A MÉTODOS

7.1 CARACTERÍSTICAS

- Nos permite referenciar métodos mediante el **operador ::**

7.2 MÉTODO ESTÁTICO

```
Clase::nombreDeMetodoEstatico
```

```
interface Construible{
    void construir();
}
public class MetodoEstatico {
    public static void construyendoAlgo(){
        System.out.println("Dandole al pico y la pala");
    }
    public static void main(String[] args) {
        Construible construible = MetodoEstatico::construyendoAlgo;
        construible.construir();
    }
}
```

7.3 MÉTODO DE INSTANCIA

```
objetoInstanciado::nombreDeMetodoDeInstancia
```

```
interface Vendible{
    void vender();
}
public class MetodoDeInstancia {
    public void vendiendoAlgo(){
        System.out.println("Ahora tienes pasta en vez de móvil.");
    }
    public static void main(String[] args) {
        MetodoDeInstancia metodo = new MetodoDeInstancia();
        Vendible vendible = metodo::vendiendoAlgo;
        vendible.vender();
        Vendible otraForma = new MetodoDeInstancia()::vendiendoAlgo;
        otraFormaDeVender.vender();
    }
}
```

7.4 CONSTRUCTOR

```
Clase::new
```

```
interface Almacen{
    Arma getArma(String arma);
}
class Arma{
    Arma(String arma){
        System.out.print("Toma una " + arma + " del calibre 42");
    }
}
public class ReferenciaAlConstructor {
    public static void main(String[] args) {
        Almacen almacen = Arma::new;
        Arma porra = almacen.getArma("Porra");
    }
}
```

8 OPTIONAL

8.1 CARACTERÍSTICAS

- Nueva clase que se utiliza para **gestionar los NullPointerExceptions**.
- Utiliza el nuevo paradigma funcional.

8.2 EJEMPLO SIN OPTIONAL

```java
public class EjemploSinOptional {
    public static void main(String[] args) {
        String[] cadena = new String[3];
        String cadenaEnMinusculas = cadena[2].toLowerCase();
        System.out.print(cadenaEnMinusculas);
    }
}
```

8.3 EJEMPLO CON OPTIONAL

```java
public class EjemploConOptional {
    public static void main(String[] args) {
        String[] cadena = new String[3];
        Optional<String> testNull = Optional.ofNullable(cadena[2]);
        if (testNull.isPresent()) {
            String cadenaEnMinusculas = cadena[2].toLowerCase();
            System.out.println(cadenaEnMinusculas);
        } else {
            System.out.println("No hay cadena en la posición indicada");
        }
    }
}
```

9 STREAM

9.1 CARACTERÍSTICAS

- Nuevas interfaces, clases y enum para **procesar datos mediante el paradigma funcional**.
- Las operaciones realizadas en el stream **no modifican el origen de datos**.
- Pueden tener **varias operaciones intermedias** (filter(), map(), distinct(), etc.).
- Pero **solo una operación final** (collect(), findAny(), count(), etc.).
- Las **operaciones intermedias son perezosas** y sólo se invocarán si se ejecuta una operación final.
- El pipeline de operaciones **se ejecutará para cada elemento**.

9.2 EJEMPLO SIN STREAM

```java
public class EjemploSinStream {
    public static void main(String[] args) {
        List<Producto> productos = new ArrayList<Producto>();
        for (int i = 0; i < 200000000; i++) {
            int random = new Random().nextInt(10 - 100) + 10;
            productos.add(new Producto(random));
        }
        List<Float> precios = new ArrayList<Float>();
        for (Producto producto : productos) {
            if (producto.precio < 20) {
                precios.add(producto.precio);
            }
        }
    }
}
```

9.3 EJEMPLO CON STREAM

```
public class EjemploConStream {
    public static void main(String[] args) {
        List<Producto> productos = new Random().ints(200000000, 10, 100)
            .map(random -> new Product(random))
            .collect(Collectors.toList());
        List<Float> preciosStream = productos.stream()
            .filter(product -> product.getPrecio() < 20)
            .map(product -> p.getPrecio())
            .collect(Collectors.toList());
    }
}
```

10 FOREACH

10.1 CARACTERÍSTICAS

- Nuevo método para **iterar** elementos.
- Se define en las interfaces Iterable y Stream.

```
default void forEach(Consumer<super T> action)
```

10.2 EJEMPLO FOREACH

```java
public class EjemploForEach {
    public static void main(String[] args) {
        List<String> crimenes = new ArrayList<String>();
        crimenes.add("Asesinamiento");
        crimenes.add("Tener mal gusto");
        crimenes.add("Faltas de ortografía");
        crimenes.add("Bocata de chorizo con nocilla");
        crimenes.forEach(crimen -> {
            System.out.println("Es un crimen el " + crimen)
        });
    }
}
```

11 COLLECTOR

11.1 CARACTERÍSTICAS

- Permite realizar operaciones de **reducción**.
- Similar al **group by** de sql.

11.2 EJEMPLO DE USO DE COLLECTOR

```
public class EjemploCollectors {
    public static void main(String[] args) {
        List<Pelicula> peliculas = new ArrayList<>();
        peliculas.add(new Pelicula(1, "ESDLA", 25));
        peliculas.add(new Pelicula(2, "Piratas del Caribe", 30));
        peliculas.add(new Pelicula(3, "Spiderman", 28));
        peliculas.add(new Pelicula(4, "Wonder Woman", 27));
        List<Float> precios = peliculas.stream()
            .map(p -> p.getPrecio())
            .collect(Collectors.toList());
        System.out.println(precios);
        DoubleSummaryStatistics total = peliculas.stream()
            .collect(Collectors.summarizingDouble(p -> p.getPrecio()));
        System.out.println("Total: " + total.getSum());
    }
}
```

12 ORDENACIÓN PARALLEL ARRAY

12.1 CARACTERÍSTICAS

- Java permite **ordenar elementos en paralelo**.

12.2 EJEMPLO

```java
import java.util.Arrays;
public class OrdenarArraysEnParalelo {
    public static void main(String[] args) {
        int[] aOrdenar = {5,8,1,0,6,9};
        System.out.print("A Ordenar : ");
        for (int i : aOrdenar) {
            System.out.print(i+" ");
        }
        Arrays.parallelSort(aOrdenar);
        System.out.println("Array Ordenado: ");
        for (int i : aOrdenar) {
            System.out.print(i+" ");
        }
    }
}
```

13 JAVA TIME

13.1 CARACTERÍSTICAS

- Nuevas interfaces, clases y enums que simplifican el procesamiento de fechas.
- **DateTimeFormatter** para formatear.
- **ZonedDateTime** para manejar fechas y horas en distintos usos horarios.
- **Duration** para manejar duraciones de tiempos (horas, minutos, segundos, etc.)
- **Period** para manejar periodos de fechas (años, meses, días, etc.)

13.2 EJEMPLO DE LA FECHA ACTUAL

```
public class HoraActual {
  public static void main(String[] args) {
    DateTimeFormatter formattter =
        DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
    ZonedDateTime now = ZonedDateTime.now();
    System.out.println(formattter.format(now));
    System.out.println(formattter.format(now.plus(Duration.ofHours(2))));
    System.out.println(formattter.format(now.plus(Period.ofDays(2))));
  }
}
```

14 BIBLIOGRAFÍA

14.1 OFICIAL DE ORACLE

- Optional
- Fechas y horas
- Expresiones Lambda y API Stream – Parte 1
- Expresiones Lambda y API Stream – Parte 2https://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-api-stream-java-2737544-esa.html
- Procesamiento de datos con streams