Java Record y JDK 14

Los Records de Java **son un nuevo tipo de clase en Java**. Este tipo de clases van a permitir modelar objetos de datos simples de una manera más simple y eliminando ese boilerplate para transformar a objetos inmutables.
El concepto de Java Record Class que nos permite generar una clase “Record” o registro.
Estas clases son muy habituales cuando trabajamos con conceptos como los DTO(Data Transfer Object) ya que nos ayudan a generar código de una forma muy rápida y sin esfuerzo a la hora de gestionar DTOS.

Entonces podemos usar el Records para escribir la misma estructura y obtener el mismo resultado, ahorrando el instanciar el constructor, los getter y setter de la clase, pero esto no quiere decir que sean reemplazados.

CARACTERÍSTICAS:
- Son inmutables
- Solo requiere tipo y nombre de los datos
- No permiten atributos de instancia, pero si estáticos
- No permiten herencia pero si implementaciones (interfaces), ya que estas son públicas y abstractas
- Se puede tener varios constructores asociados
- Se puede tener métodos adicionales
- No es recomendable usar para persistencia de base de datos

Vamos a implementar una clase típica de DTO:

```
private final boolean status;

public ProductClass(int idProduct, String name, double price, boolean status) {
    this.idProduct = idProduct;
    this.name = name;
    this.price = price;
    this.status = status;
}

public int getIdProduct() {
    return idProduct;
}

public String getName() {
    return name;
}

public double getPrice() {
    return price;
}

public boolean isStatus() {
    return status;
}
```

```
@Override
public String toString() {
    return "ProductClass{" +
            "idProduct=" + idProduct +
            ", name='" + name + '\'' +
            ", price=" + price +
            ", status=" + status +
            '}';
}
```

Hemos generado los métodos set/get el constructor los métodos equals y hashcode donde hemos decidido que la igualdad al ser un DTO se basa en la combinación de todas las propiedades del objeto. Ademas hemos generado también un método toString().

La realidad es que el código del DTO es muy grande para lo poco que tiene que hacer y todas las casuísticas a nivel de DTO son idénticas ya que todos tienen sus propiedades se comparan por igualdad comprobando todas sus propiedades.

A partir de la versión 14 del JDK y en modo preview disponemos de los Java Records un concepto que nos permite simplificar de forma fuerte la construcción de estos DTOs.
Se incorpora el RECORD remplazando el CLASS.

Este código funciona de la misma forma que el código anterior que son unas 50 lineas. Poco a poco estos nuevos conceptos se empezarán a usar en los desarrollos y hay que irlos conociendo.

No existen métodos set y get sino que son directamente propiedades name() , price(), status(). Esta simplificación puede ser natural para algunos o excesiva para otros ya que no genera homogeneidad.

Y en aquellos atributos instanciados en RECORD podríamos inicializarlos de la siguiente manera en nuestro main:

```java
package com.mitocode.app;

import com.mitocode.records.ProductRecord;

public class App {

    public static void main(String[] args) {
        ProductRecord pr1 = new ProductRecord( idProduct: 1,  name: "PS5",  price: 999.99,  status: true);
        System.out.println(pr1);
    }
}
```

devolviendo por consola:

```
App
"C:\Program Files\Eclipse Adoptium\jdk-19.0.0.36-hotspot\bin\java.exe
ProductRecord[idProduct=1, name=PS5, price=999.99, status=true]

Process finished with exit code 0
```

Como podemos ver, no habría necesidad de **sobreescribir un toString** al utilizar RECORD.

Ahora en el caso que no queramos que todos los atributos sean inmutables, **sólo algunos pocos,** se realizaría de la siguiente manera:

```java
package com.mitocode.records;

public record ProductRecord(int idProduct,
                            String name,
                            double price,
                            boolean status) {

    public ProductRecord(String name){
        this( idProduct: 0, name,  price: 599.99,  status: true);
    }
}
```

En este caso, le estamos pasando valores a nuestros atributos suscribiendo el constructor, excepto al name, y luego price,
dejándolos inmutables.

```java
public ProductRecord(String name, double price){
    this( idProduct: 0, name, price,  status: true);
}
```

Inicializando solo los valores que dejamos inmutables, porque lo demás en teoría variarían:

```java
public class App {

    public static void main(String[] args) {
        ProductRecord pr1 = new ProductRecord( idProduct: 1,  name: "PS5",  price: 999.99,  status: true);
        ProductRecord pr2 = new ProductRecord( name: "PS4");
        ProductRecord pr3 = new ProductRecord( name: "PS4",  price: 799.99);

        System.out.println(pr1);
        System.out.println(pr2);
        System.out.println(pr3);
    }
}
```

```
"C:\Program Files\Eclipse Adoptium\jdk-19.0.0.36-hotspot\bin\jav
ProductRecord[idProduct=1, name=PS5, price=999.99, status=true]
ProductRecord[idProduct=0, name=PS4, price=599.99, status=true]
ProductRecord[idProduct=0, name=PS4, price=799.99, status=true]

Process finished with exit code 0
```

Tambien podriamos implementar **métodos** a nuestro Record:

```java
public String nameLowerCase(){
    return name.toLowerCase();
}
```

Como transformar nuestro name en minúscula, por ejemplo
Y como no utilizamos métodos de Getters y Setters, en nuestro main llamamos solamente al atributo:

```java
    //System.out.println(pr2);
    System.out.println(pr3.name());
    System.out.println(pr3.nameLowerCase());
}
```

```
"C:\Program File
PS3
ps3
```

Y dentro de esto, también podríamos implementar **métodos estáticos**:

```java
//static
public static void printYearOfProduct(){
    System.out.println("2022");
}
```

En donde, en nuestro main, lo llamaremos únicamente por la
clase Product Record:

```java
    //System.out.println(pr3.name());
    //System.out.println(pr3.nameLowerCas
    ProductRecord.printYearOfProduct();
}
```

Como se dijo anteriormente, en Java Record no se puede implementar herencia pero si podemos con interfaces.

Creando una **interfaz**, por ejemplo:

```java
package com.mitocode.records;

public interface ProductMachine {

    void printCountryMachine();
}
```

Con un constructor público y abstracto por definición.
Y luego agregando **implements** al ProductRecord para generar la interfaz.

Sobrescribiendo aquel método en esta clase:

```java
public record ProductRecord(int idProduct,
                            1 usage
                            String name,
                            double price,
                            boolean status) implements ProductMachine{
```

```java
    @Override
    public void printCountryMachine() {
        System.out.println("PERU");
    }
```

Y este método se sobreescribira en nuestra clase records para que este pueda utilizarla.

Podriamos tener el caso de un **Record**, dentro de otro **Record**:

Instanciamos un **Record Category** con sus respectivos atributos:

```java
package com.mitocode.records;

public record Category(int idCategory, String name) {
}
```

Y luego, otro Record donde agregaremos Category entre sus atributos:

```java
package com.mitocode.records;

public record SuperProduct(int idProduct, String name, Category category) {
}
```

Por lo que en nuestro main, se podrá llamar de la siguiente manera:

```java
SuperProduct sp = new SuperProduct( idProduct: 1,  name: "RTX3080TI", new Category( idCategory: 1,  name: "VIDEO-CARDS"));
System.out.println(sp);
```

Obteniendo la información propia:

```
"C:\Program Files\Eclipse Adoptium\jdk-19.0.0.36-hotspot\bin\java.exe" "-javaagent:D:\Program
SuperProduct[idProduct=1, name=RTX3080TI, category=Category[idCategory=1, name=VIDEO-CARDS]]
```

Si quisiéramos **instanciar nuevos atributos**, únicamente estos deberán ser estáticos e inmutables:

```
public static final String DEFAULT_PRODUCT = "COMPUTER";
```

Y llamamos el atributo de la siguiente manera en el main:

```
//System.out.println(pr3);
System.out.println(ProductRecord.DEFAULT_PRODUCT);
```