Programación

09 Gestión de Bases de Datos mediante código

José Luis González Sánchez

Contenidos

¿Qué voy a aprender?

Contenidos

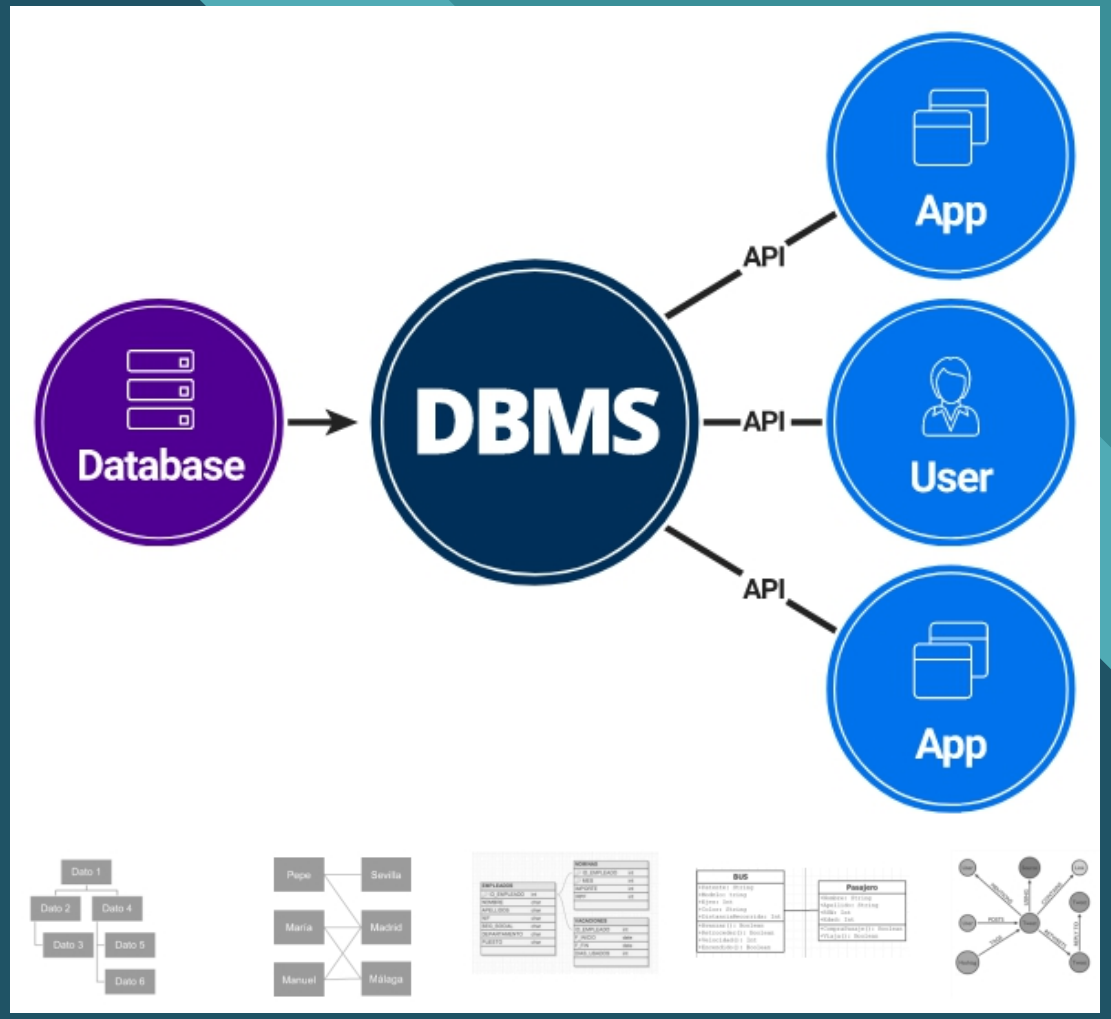
1. Bases de datos Relacionales
2. El Desfase Objeto-Relacional
3. Acceso a Base de Datos Relacionales
4. CRUD con JDBC
5. El problema de Null y el tipo Optional
6. Procesando información en colecciones. API Stream y programación funcional

Bases de Datos Relacionales

La información entre tablas, filas y columnas

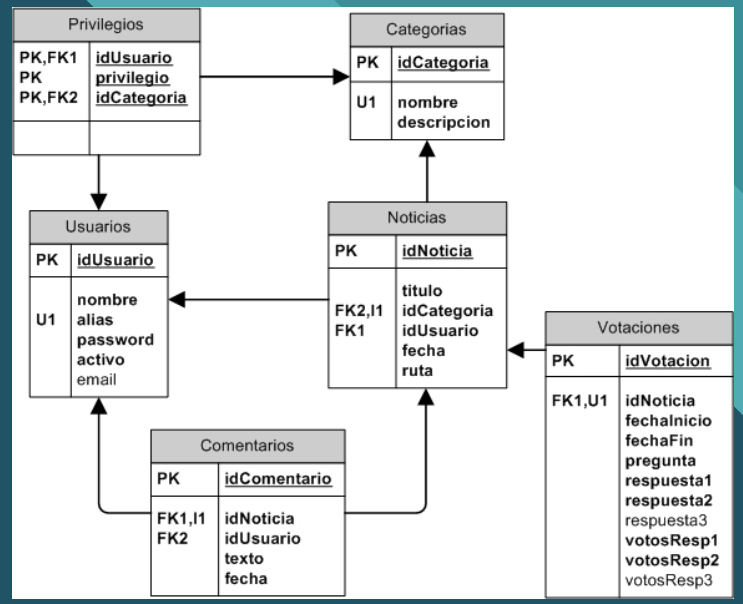
Bases de Datos Relacionales

- Desde que ser humano existe, siempre hemos tenido la necesidad de almacenar la información, ya sea desarrollando la escritura, luego el papel, imprenta, libros, cintas perforadas, disco duro, etc.
- A la misma vez que almacenamos la información debemos organizarla y procesarla ya sea en ficheros de acceso aleatorio, indexados o usando bases de datos.
- Tenemos distintos DBMS según si esquema lógico:
  - Jerárquicos: En árbol, estructura padre/hijo.
  - En Red: Nodos y enlaces. Manejo complejo
  - Relacional. Uso de tablas como única estructura
  - Orientados a objetos: Origen en la POO, para uso concreto.
  - NoSQL, Not Only SQL, Orientado a documentos, no todo es relacional.



Bases de Datos Relacionales

- La base de datos relacional (BDR) es un tipo de base de datos (BD) que cumple con el modelo relacional (el modelo más utilizado actualmente para implementar las BD ya planificadas). Tras ser postuladas sus bases en 1970 por Edgar Frank Codd, de los laboratorios IBM en San José (California), no tardó en consolidarse como un nuevo paradigma en los modelos de base de datos.
- Virtualmente, todos los sistemas de bases de datos relacionales utilizan SQL (Structured Query Language) para consultar y mantener la base de datos.
  - Una base de datos se compone de varias tablas, denominadas relaciones.
  - No pueden existir dos tablas con el mismo nombre ni registro.
  - Cada tabla es a su vez un conjunto de campos (columnas) y registros (filas).
  - La relación entre una tabla padre y un hijo se lleva a cabo por medio de las llaves primarias y llaves foráneas (o ajenas).
  - Las llaves primarias son la clave principal de un registro dentro de una tabla y estas deben cumplir con la integridad de datos.
  - Las llaves ajenas se colocan en la tabla hija, contienen el mismo valor que la llave primaria del registro padre; por medio de estas se hacen las formas relacional

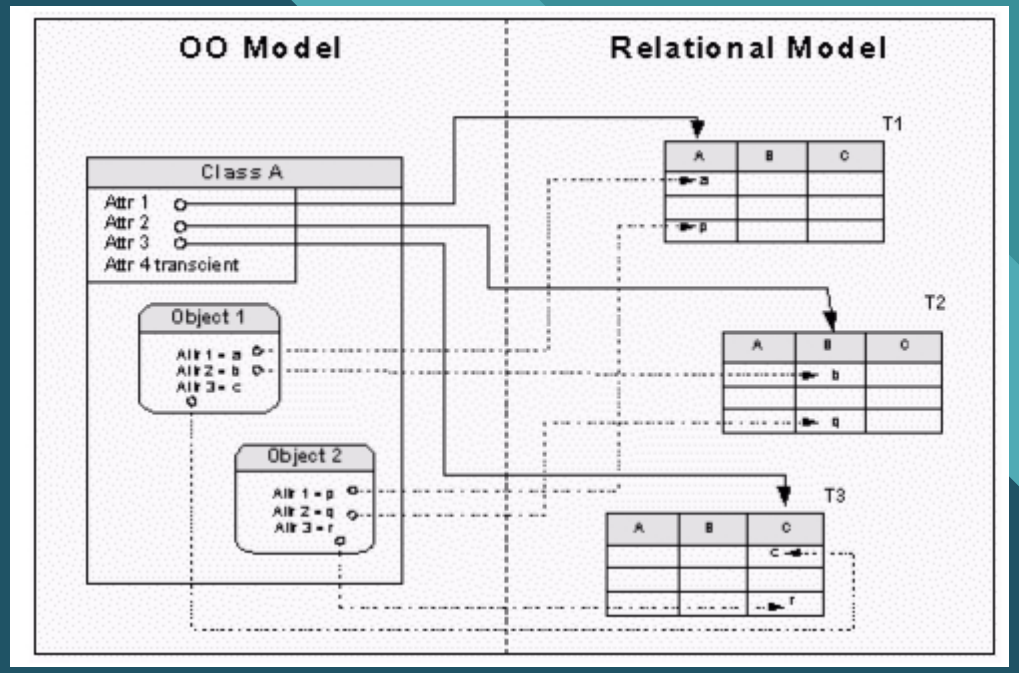


Desfase Objeto-Relacional

De tablas, fuilas y columnas a objetos

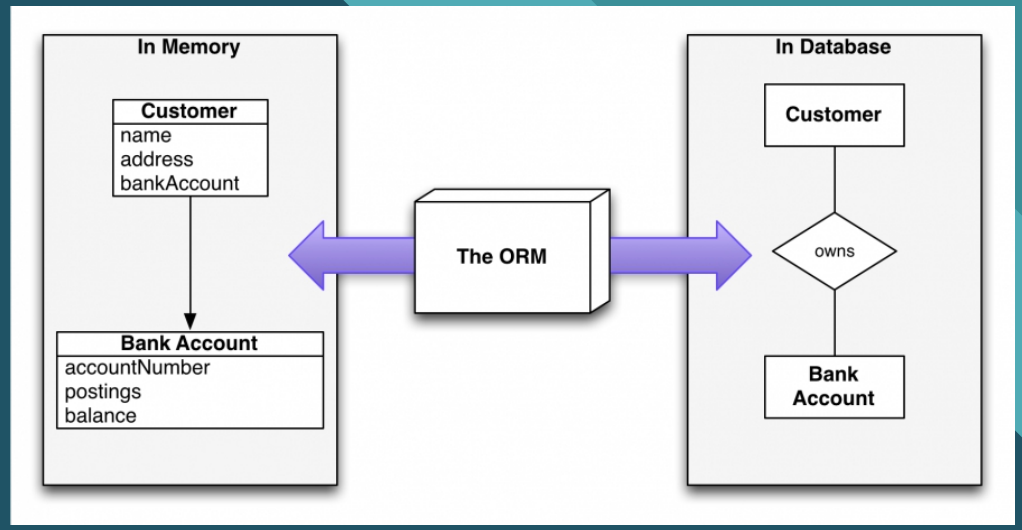
Desfase Objeto-Relacional

- También conocido como **impedancia objeto-relacional,** se trata de la diferencia existente entre la programación orientada a objetos y las bases de datos relacionales. Nos referimos a:
  - Lenguaje: El desarrollador debe conocer dos lenguajes: el de programación y el de la base de datos.
  - Tipos de datos: En la POO el tipo de datos es más complejo.
  - Paradigma de programación: Tiene que haber una traducción del modelo entidad-relación de la base de datos al modelo orientado a objetos del lenguaje de programación.
- Es decir, el modelo relacional trata de relaciones y conjuntos, mientras que POO trata los datos como objetos y asociaciones entre los mismos. Es decir al ahora de programar con BBDD Relacionales, debemos:
  - 1.Abrir una conexión.
  - 2.Crear sentencia SQL
  - 3.Copiar las propiedades del objeto y la sentencia y lanzar la consulta, o objetner de la consulta la información y **mapearla en el objeto.**



Desfase Objeto-Relacional. Mapeo Objeto Relacional

- Al trabajar con lenguajes orientados a objetos y bases de datos relacionales tenemos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos.
- A este conjunto de técnicas que pueden desarrollarse de manera manual por el desarrollador o con apoyo de algún software se conoce como mapeo objeto-relacional.
- Aunque pueda parecer sencillo, es un problema si la base de datos cambia en estructura o tipo de datos, o simplemente el modelo de dominio de nuestro problema muta con el paso del tiempo, lo que puede provocar bastante cambios en cascada en nuestro código.



CRUD con JDBC

Create, Read, Update & Delete

CRUD

CRUD con JDBC. Conexión

- Es necesario **instalar el conector o driver de la base de datos**. En una aplicación JAVA, se instala un conector JDBC, que es el que implementa la funcionalidad de las clases de acceso a datos, y proporciona comunicación entre el API JDBC y el sistema gestor de bases de datos. Traduce los comandos al protocolo nativo del SGBD.
- La versión 3.0 de JDBC proporciona un **pool de conexiones**. Al inicia un servidor JAVA EE, el pool de conexiones crea un número de conexiones físicas iniciales. Cuando un objeto Java del servidor necesita una conexión (método dataSource.getConnection()), la fuente de datos javax.sql.DataSource habla con el pool de conexiones y éste le entrega una conexión lógica java.sql.Connection. Esta conexión es la que recibe el objeto Java.
- Para cerrar una conexión (método connection.close()) la fuente de datos javax.sql.DataSource habla con el pool de conexiones y devuelve la conexión lógica.
- Si hay una alta demanda de conexiones a la base de datos, el pool de conexiones crea más conexiones físicas de objetos tipo Connection.
- El código básico para conectarnos a una base de datos con pool de conexiones a través de JNDI podría ser:
  - javax.naming.Context ctx = new InitialContext();
  - dataSource = (DataSource) ctx.lookup("java:comp/env/jdbc/Basededatos");
- Para realizar una operación, obtenemos una conexión lógica:
  - connection = dataSource.getConnection();
- Y para cerrarla:
  - connection.close();

CRUD con JDBC. Conexión

```java
// Conexión a una BBDD
Connection conexion = null;

try {
  conexion = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/basededatos",
    "usuario", "password");
} catch (ClassNotFoundException cnfe} {
  cnfe.prinStackTrace();
} catch (SQLException sqle) {
  sqle.prinStackTrace();
} catch (InstantiationException ie) {
  ie.prinStackTrace();
} catch (IllegalAccessException iae) {
  iae.prinStackTrace();
}
```

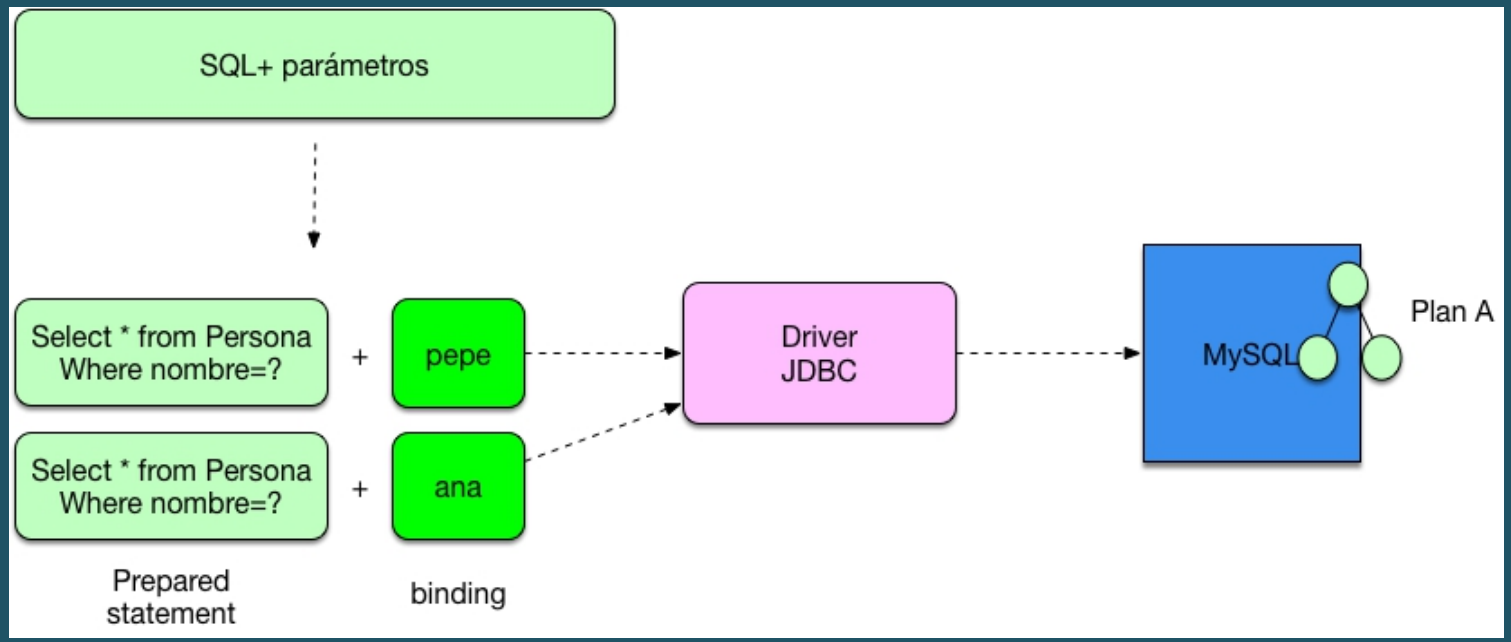
```java
// Desconexión a una BBDD
. . .
try {
  conexion.close();
  conexion = null;
} catch (SQLException sqle) {
  sqle.printStackTrace();
}
. . .
```

CRUD con JDBC. Consultas

- Para operar con una base de datos, nuestra aplicación debe:
  - Cargar el driver necesario.
  - Establecer una conexión con la base de datos.
  - Enviar consultas SQL, procesando el resultado.
  - Liberar los recursos al terminar.
  - Gestionar posibles errores.
- Se pueden usar tres tipos de sentencias:
  - Statement: Sentencias SQL
  - **PreparedStatament**: Consultas preparadas, como las que tienen parámetros. Recomendado
  - CallableStatement: Ejecutar procedimientos almacenados en la base de datos.
- JDBC distingue dos tipos de consultas:
  - Consultas: SELECT
  - Actualizaciones: UPDATE, INSERT, DELETE y sentencias DDL.

CRUD con JDBC. Consultas

- El uso de **JDBC Prepared Statement** es hoy en día prácticamente obligatorio. Son consultas precompiladas por lo que son más eficientes y pueden tener parámetros reutilizando el mismo plan de consulta y evitando ataques por inyección SQL. Se instancia de la siguiente manera (ejemplo con tabla medicamentos):
  - Si hay que emplear parámetros en uan consulta, se puede hacer usando el carácter “?”:
  - Establecemos los parámetros de una consulta utilizando métodos set que dependen del tipo SQL de la columna y el orden donde aparecen.
  - Finalmente ejecutamos la consulta con executeQuery() o executeUpdate(), dependiendo del tipo de consulta que sea.



CRUD con JDBC. Selección

- **Una consulta de selección** sobre una tabla de la base de datos devuelve un conjunto de datos organizados en filas y columnas (registros).
- **Usamos executedQuery**
- En Java almacenamos esa información mediante un objeto de la clase **ResultSet**, el cual está formado por filas y columnas.
- Un ResultSet solo puede recorrerse fila por fila, desde la primera hasta la última en una única dirección deon su método next().
- Además, podemos acceder a cualquier columna de cada fila mediante algunos de sus métodos (getInt(), getString(), getObject(), etc). Las columnas se numeran empezando en el nº 1.
- Obtener las columnas de una consulta Select. Si en algún caso al realizar una consulta no sabemos cuántas columnas nos va a devolver, podemos usar el método getMetaData() que devuelve un objeto del tipo ResultSetMetaData. Se puede obtener antes si se usa PreparedStatement

```
// Consulta tipo Select
. . .
String sentenciaSql = "SELECT nombre, precio FROM productos " +
                      "WHERE precio = ?";
PreparedStatement sentencia = null;
ResultSet resultado = null;

try {
  sentencia = conexion.prepareStatement(sentenciaSql);
  sentencia.setFloat(1, filtroPrecio);
  resultado = sentencia.executeQuery();
  while (resultado.next()) {
    System.out.println("nombre: " + resultado.getString(1));
    System.out.println("precio: " + resultado.getFloat(2));
  }
} catch (SQLException sqle) {
  sqle.printStackTrace();
} finally {
  if (sentencia != null)
    try {
      sentencia.close();
      resultado.close();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
}
. . .
```

CRUD con JDBC. Insercción, Actualización y Borrado

- Selen devolver un entero con el número de registros afectados
- Se usa **executeUpdate()**

```
// Consulta tipo insert
. . .
String sentenciaSql = "INSERT INTO productos (nombre, precio) VALUES (?, ?)";
PreparedStatement sentencia = null;

try {
  sentencia = conexion.prepareStatement(sentenciaSql);
  sentencia.setString(1, nombreProducto);
  sentencia.setFloat(2, precioProducto);
  sentencia.executeUpdate();
} catch (SQLException sqle) {
  sqle.printStackTrace();
} finally {
  if (sentencia != null)
    try {
      sentencia.close();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
}
. . .
```

```
// Consulta tipo update
 . .
String sentenciaSql = "UPDATE productos SET nombre = ?, precio = ? " +
                      "WHERE nombre = ?";
PreparedStatement sentencia = null;

try {
  sentencia = conexion.prepareStatement(sentenciaSql);
  sentencia.setString(1, nuevoNombreProducto);
  sentencia.setFloat(2, precioProducto);
  sentencia.setString(3, nombreProducto);
  sentencia.executeUpdate();
} catch (SQLException sqle) {
  sqle.printStackTrace();
} finally {
  if (sentencia != null)
    try {
      sentencia.close();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
}
. . .
```

```
// Consulta tipo delete
. . .
String sentenciaSql = "DELETE productos WHERE nombre = ?";
PreparedStatement sentencia = null;

try {
  sentencia = conexion.prepareStatement(sentenciaSql);
  sentencia.setString(1, nombreProducto);
  sentencia.executeUpdate();
} catch (SQLException sqle) {
  sqle.printStackTrace();
} finally {
  if (sentencia != null)
    try {
      sentencia.close();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
    }
}
. . .
```

El problema de Null y el tipo Optional

Evitando NullPointer y controlando nuestro código

Null y Optional

- El problema de NullPointerException o lidiar con los null es uno de los problemas más importantes y costosos a la hora de programar. Es por ello que debemos llenar nuestro código con condiciones para controlarlo. Si estas comprobaciones se nos olvidan podemos tener problemas, o la famosa excepción.
- **Desde Java 8, existe el tipo Optional. Este nos permite crear un "envoltorio" obligándonos a usar sus métodos para evitar el null, o simplemente lanzar una excepción si es necesario.**

```
String name = "baeldung";

Optional<String> opt = Optional.of(name);

assertTrue(opt.isPresent());

assertFalse(opt.isEmpty());

Optional.ofNullable(nullName).orElse("john");

Optional.ofNullable(nullName).orElseThrow(...);

Optional<String> opt = Optional.ofNullable(null);

String name = opt.get();
```

Procesando información en colecciones. API Stream y programación funcional

Trabajando de manera fluída y limpia

API Stream

- Antes de nada debemos introducir la **programación funcional en Java**
- **En la programación Estructurada nos centramos en en el QUÉ y el CÓMO, en la funcional nos centramos en el QUÉ**
- **Expresiones Lambda:** Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase. Su sintáxis básica se detalla a continuación: (parámetros) -> { cuerpo-lambda}
- El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.
- Parámetros:
  - Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
  - Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
- Cuerpo de lambda:
  - Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la clausula return en el caso de que deban devolver valores.
  - Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la clausula return en el caso de que la función deba devolver un valor .

API Stream

- **Predicate<T>**: Comprueba si se cumple o no una condición. Se utiliza mucho junto a expresiones lambada a la hora de filtrar:

```
.filter((p) -> p.getEdad() >= 35)
```

- **Consumer<T>**: Sirve para consumir objetos. Uno de los ejemplos más claros es imprimir.

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

  - Adicionalmente, tiene el método andThen, que permite componer consumidores, para encadenar una secuencia de operaciones.

- **Function<T, R>**: Sirve para aplicar una transformación a un objeto. El ejemplo más claro es el mapeo de objetos en otros.

```
.map((p) -> p.getNombre())
```

  - Adicionalmente, tiene otros métodos:
    - andThen, que permite componer funciones.
    - compose, que compone dos funciones, a la inversa de la anterior.
    - identity, una función que siempre devuelve el argumento que recibe

API Stream - Métodos de Búsqueda

- Son un tipo de operaciones terminales sobre un stream, que nos permiten:

- Identificar si hay elementos que cumplen una determinada condición
- Obtener (si el stream contiene alguno) determinados elementos en particular.
- Algunos de los métodos de búsqueda son:

- allMatch(Predicate<T>): verifica si todos los elementos de un stream satisfacen un predicado.
- anyMatch(Predicate<T>): verifica si algún elemento de un stream satisface un predicado.
- noneMatch(Predicate<T>): opuesto de allMatch(…)
- findAny(): devuelve en un Optional<T> un elemento (cualquiera) del stream. Recomendado en streams paralelos.
- findFirst() devuelve en un Optional<T> el primer elemento del stream. NO RECOMENDADO en streams paralelos.

API Stream - Collectors

- **Collectors:** Los collectors nos van a permitir, en una operación terminal, construir una collección mutable, el resultado de las operaciones sobre un stream.

- **Colectores "básicos":** Nos permiten operaciones que recolectan todos los valores en uno solo. Se solapan con alguans operacinoes finales ya estudiadas, pero están presentes porque se pueden combinar con otros colectores más potentes.
  - counting: cuenta el número de elementos.
  - minBy(…), maxBy(…): obtiene el mínimo o máximo según un comparador.
  - summingInt, summingLong, summingDouble: la suma de los elementos (según el tipo).
  - averagingInt, averagingLong, averagingDouble: la media (según el tipo).
  - summarizingInt, summarizingLong, summarizingDouble: los valores anteriores, agrupados en un objeto (según el tipo).
  - joinning: unión de los elementos en una cadena.
- **Colectores "grouping by":** Hacen una función similar a la cláusula GROUP BY de SQL, permitiendo agrupar los elementos de un stream por uno o varios valores. Retornan un Map.

API Stream - Filters

- **filter** es una operación intermedia, que nos permite eliminar del stream aquellos elementos que no cumplen con una determinada condición, marcada por un Predicate<T>.

```
personas.stream().filter(p -> p.getEdad() >= 18 && p.getEdad() <= 65).forEach(persona ->
System.out.printf("%s (%d años)%n", persona.getNombre(), persona.getEdad()));
```

- Es muy combinable con algunos métodos como findAny o findFirst:

```
Persona p1 = personas.stream().filter(p -> p.getNombre().equalsIgnoreCase("Pepe")).findAny().orElse(new
Persona());
```

API Stream - Referencia a métodos

- Las referencias a métodos son una forma de hacer nuestro código aun más conciso:
  - Clase::metodoEstatico: referencia a un método estático.
  - objeto::metodoInstancia: referencia a un método de instancia de un objeto concreto.
  - Tipo::nombreMetodo: referencia a un método de instancia de un objeto arbitrario de un tipo en particular.
  - Clase::new: referencia a un constructor.

API Stream - Similitudes a SQL

SQL	API Stream
from	stream()
select	map()
where	filter() (antes de un collecting)
order by	sorted()
distinct	distinct()
having	filter() (después de un collecting)
join	flatMap()
union	concat().distinct()
offset	skip()
limit	limit()
group by	collect(groupingBy())
count	count()

API Stream - Resumen

- **Stream** Representa un flujo de elementos y podemos aplicar métodos tales como
  - filter para filtrar los elementos. Éste método recibe un predicado como argumento.
  - map para devolver solo el atributo indicado. Es como el select de sql.
  - sorted para ordenar nuestros elementos. Recibe un Comparator.
  - min, max, count que permiten obtener el minimo, máximo y conteo de elementos respectivamente.
  - collect aquí es donde definiremos nuestros funciones de agregado, principalmente agrupados, particiones, etc.
  - Comparator nos proporciona métodos útiles para realizar ordenamientos. Lo usaremos en conjunto con el método sorted
  - Collectors Nos proporciona métodos útiles para agrupar, sumar, promediar, obtener estadísticas. Lo usaremos en conjunto con el método collect

"La programación es una carrera entre los desarrolladores, intentando construir mayores y mejores programas a prueba de idiotas, y el universo, intentanto producir mayores y mejores idiotas. Por ahora va ganando el Universo"

- Rich Cook

Recursos

- Twitter: https://twitter.com/joseluisgonsan
- GitHub: https://github.com/joseluisgs
- Web: https://joseluisgs.github.io
- Discord: https://discord.gg/eFMKxfPY
- Aula Virtual: https://aulavirtual33.educa.madrid.org/ies.luisvives.leganes/course/view.php?id=247

Gracias

José Luis González Sánchez