«Talento Tech»

Back-End

Java

Clase 08

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase 8: Manejo de excepciones y módulos

Índice

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Bienvenida a TechLab

Talento Lab

¡Les damos nuevamente la bienvenida a **TechLab**! En esta ocasión, **Sibelius**, nuestro cliente, se prepara para un importante evento de lanzamientos de productos gourmet, y su plataforma backend está creciendo en complejidad. Imaginá que cada día se agregan nuevos productos —algunos con información incompleta— y se crean módulos especializados para distintas partes del negocio (gestión de pedidos, inventario, promociones, etc.).

El equipo de desarrollo (Matías y Sabrina, junto a vos) ha notado que con cada nueva pieza de código que se integra, surgen situaciones inesperadas: **datos inválidos**, **productos inexistentes** que alguien intenta consultar, **stock negativo** por error, y hasta escenarios donde ciertos archivos de configuración se pierden entre tantos paquetes. Estas "situaciones excepcionales" desatan comportamientos erráticos o silenciosos, y atentan contra la robustez del sistema e-commerce de Sibelius.

Ante tal desorden, **Silvia**, la Product Owner, reclama una solución clara:

1. **Manejar los errores** de manera informativa y segura, evitando que el sistema "truene" sin aviso o arroje resultados incoherentes.
2. **Reestructurar** el proyecto en **paquetes** (productos, pedidos, excepciones, etc.) y **módulos** (ej. `com.techlab.negocio`), de modo que cada parte del negocio se mantenga organizada y con dependencias claras.

Para lograr esto, **el manejo de excepciones y la modularización** son piezas clave. Con ellos, evitaremos que la aplicación falle de forma silenciosa o inestable, y además tendremos un diseño escalable conforme Sibelius siga expandiendo sus líneas de productos. Con **Java** y sus mecanismos de **excepciones**, **paquetes** y **módulos**, TechLab podrá ofrecer un sistema de ecommerce confiable y bien estructurado para triunfar en el evento de lanzamientos de Sibelius.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Objetivos de la clase

- Comprender cómo funcionan las **excepciones** en Java y la diferencia entre excepciones verificadas (checked) y no verificadas (unchecked).
- Aprender a manejar excepciones con **try, catch y finally** asegurando un flujo de ejecución controlado ante errores.
- Conocer la creación de **excepciones personalizadas** para representar errores específicos del dominio de TechLab.
- Entender la **organización del código con paquetes**, cómo importar clases y mantener una estructura coherente del proyecto.
- Introducir el concepto de **módulos en Java (desde Java 9)** y cómo el `module-info.java` puede ayudar a gestionar dependencias entre partes del proyecto.

Manejo de excepciones en Java

Una **excepción** es un evento que ocurre durante la ejecución y que interrumpe el flujo normal del programa. En Java, las excepciones son objetos que extienden `Throwable`. Existen dos grandes categorías:

- **Checked Exceptions:** Heredan de `Exception`. Deben ser declaradas en la firma del método o manejadas con try-catch.
- **Unchecked Exceptions:** Heredan de `RuntimeException`. No requieren declaración explícita. Surgen típicamente por errores de programación (índices fuera de rango, null pointers, etc.).

Uso de try, catch, finally

Para manejar excepciones, usamos el bloque `try` que envuelve el código que podría fallar. Si ocurre una excepción, se transfiere el control a un bloque `catch` que la maneja. Opcionalmente, un bloque `finally` ejecuta código que se corre siempre, haya o no excepción, lo cual es ideal para liberar recursos.

```
try {
    Producto p = buscarProductoPorNombre("Café");
    System.out.println("Producto encontrado: " + p.getNombre());
} catch (ProductoNoEncontradoException e) {
    System.out.println("No se encontró el producto: " + e.getMessage());
} finally {
    System.out.println("Operación de búsqueda finalizada.");
}
```

Creación de excepciones personalizadas

Podemos crear excepciones a la medida del dominio, extendiendo `Exception` o `RuntimeException`. Por ejemplo:

```
public class ProductoNoEncontradoException extends Exception {
    public ProductoNoEncontradoException(String mensaje) {
        super(mensaje);
    }
}
```

Esto permite comunicar con claridad qué tipo de error sucedió.

Problema: manejo de errores al buscar productos inexistentes

Contexto: Silvia se queja de la falencia del sistema al no informar claramente cuando se busca un producto que no existe. Matías y Sabrina deciden lanzar una `ProductoNoEncontradoException` cuando no hallan el producto y manejarla en el `main`.

Solución:

- Definir `ProductoNoEncontradoException`.
- En el método `buscarProductoPorNombre`, si no se encuentra el producto, `throw new ProductoNoEncontradoException("El producto 'X' no existe.")`.
- En `main`, usar **try-catch** para capturar esta excepción y mostrar un mensaje adecuado.

Organización del código con paquetes

A medida que el proyecto crece, conviene agrupar clases en **paquetes** para mayor claridad. Por ejemplo:

- `com.techlab.productos` para clases `Producto`, `Bebida`, `Comida`.
- `com.techlab.clientes` para `Cliente`.
- `com.techlab.pedidos` para `Pedido`.
- `com.techlab.excepciones` para excepciones personalizadas.

Esto facilita la búsqueda y el mantenimiento del código. Para usar una clase de otro paquete, se la `importa`:

```
import com.techlab.productos.Producto;
import com.techlab.excepciones.ProductoNoEncontradoException;
```

Problema: dividir el proyecto en paquetes lógicos

Contexto: Silvia pide agregar más productos y más tipos de clientes. Matías y Sabrina deciden separar las clases por dominio. Ahora, `Producto` y sus subclases van a `com.techlab.productos`, las excepciones a `com.techlab.excepciones`, etc. Esto hace más fácil ubicar y mantener el código a largo plazo.

Módulos en Java (Java 9+)

Los **módulos** en Java permiten una división lógica mayor que los paquetes. Un módulo puede contener varios paquetes y un descriptor `module-info.java` que especifica qué paquetes se exportan y qué dependencias se requieren de otros módulos.

Beneficios:

- Control preciso de qué partes del código son accesibles.
- Gestión de dependencias explícita, evitando colisiones de clases y problemas al escalar el proyecto.

```
// module-info.java
module com.techlab.negocio {
    exports com.techlab.productos;
    exports com.techlab.clientes;
    requires com.techlab.utilidades;
}
```

Problema: creación de un módulo para la lógica de negocio

Contexto: Silvia quiere que la lógica de negocio principal (productos, pedidos, clientes) esté separada en un módulo llamado `com.techlab.negocio`. Otros módulos podrían encargarse de la persistencia o la interfaz de usuario.

Solución:

- Crear `module-info.java` en `com.techlab.negocio`.
- Especificar qué paquetes se exportan para que otros módulos puedan usarlos.
- Declarar dependencias con `requires` para acceder a módulos auxiliares.

Esto permite un mayor control sobre la arquitectura del proyecto y facilita la integración con otros componentes.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Situación inicial en TechLab

Silvia, la Product Owner, ha notado que algunas operaciones del backend podrían fallar. Por ejemplo, si un cliente busca un producto que no existe, el sistema podría simplemente fallar o retornar datos incoherentes. Matías y Sabrina desean manejar estas situaciones informando claramente al usuario o registrando el error para su posterior análisis.

Además, la cantidad de clases y paquetes crece y el equipo ve la necesidad de mantener una estructura ordenada. Esto implica agrupar clases en paquetes según su responsabilidad (productos, clientes, pedidos, utilidades) e incluso considerar la modularización para controlar qué partes del código se exponen y a quién.

Para cumplir con necesidades el equipo de desarrollo (Matías, Sabrina) y vos deberán realizar los siguientes ejercicios prácticos:

Ejercicios prácticos

1. **Manejo de excepciones:**
   - Creá el método `buscarProductoPorNombre(String nombre)` que lance `ProductoNoEncontradoException` si no halla el producto.
   - Manejá la excepción en un `main` con try-catch.
2. **Excepciones personalizadas:**
   - Creá una `StockInsuficienteException` que se lance cuando se intente vender más unidades de las que hay en stock.
   - Manejá la excepción en el código y mostrale al usuario un mensaje adecuado.
3. **Organización en paquetes:**
   - Mové las clases `Producto`, `Bebida`, `Comida` al paquete `com.techlab.productos`.
   - Mové las excepciones a `com.techlab.excepciones`.
   - Ajustá los imports en el `main`.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

4. **Módulos:**
    - Creá un módulo `com.techlab.negocio` con un `module-info.java`.
    - Exportá el paquete `com.techlab.productos`.
    - Simulá que otro módulo `com.techlab.ui` importa y utiliza `Producto`.

Materiales y recursos adicionales

- Documentación Oficial de Java: Excepciones
- Módulos en Java (Java 9+)
- Videos recomendados en YouTube sobre manejo de excepciones, empaquetado del código y modularización.

Preguntas para reflexionar

- ¿Cómo el manejo adecuado de excepciones mejora la robustez y calidad del software en TechLab?
- ¿En qué situaciones utilizarías excepciones personalizadas en lugar de las predefinidas en Java?
- ¿De qué manera organizar el código en paquetes y módulos facilita la evolución y el mantenimiento del proyecto?

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Consignas de la Pre-Entrega (Proyecto Integrador)

Introducción a la Pre-Entrega del Proyecto

¡Hola, desarrollador/a de TechLab! Recordá que es **fundamental** cumplir con la **pre-entrega** de tu proyecto integrador para evidenciar el avance en cada uno de los temas aprendidos: desde lo más básico en Java hasta la organización modular y el manejo de excepciones. Esta instancia te va a permitir:

1. **Validar** que hayas comprendido y aplicado los conceptos clave (POO, colecciones, excepciones, etc.).
2. **Recibir feedback** oportuno antes de la entrega final.
3. **Asegurar** que tu aplicación esté en un estado funcional y coherente con los requerimientos de un sistema de e-commerce inicial.

A continuación, se detallan **las consignas** que debés cumplir en tu pre-entrega. ¡Revisalas atentamente y asegurate de cubrir cada una para demostrar tu dominio de los contenidos!

1. **Ingreso de Productos**
    - Implementar una **funcionalidad** para **agregar productos** con:
        - Nombre (String).
        - Precio (double).
        - Cantidad en Stock (int).
    - Almacenarlos en una **colección dinámica** (ej. `ArrayList<Producto>`).
2. **Visualización de Productos**
    - **Listar** todos los productos, mostrando:
        - ID (autogenerado o posición en la lista).
        - Nombre, Precio, Stock.
    - El sistema debe mostrar un **menú** con la opción de “Listar productos”.
3. **Búsqueda y Actualización**
    - Permitir **buscar** un producto por **nombre o ID**.
    - Si se encuentra, **mostrar su información** (mínimo: nombre, precio, stock).
    - (Opcional) Poder **actualizar** algunos datos (precio o stock), validando que los valores sean coherentes (ej., stock no negativo).

Buenos Aires aprende
Agencia de Habilidades para el Futuro

4. **Eliminación de Productos**
    - Habilitar la **eliminación** de un producto por **ID** o por **posición**.
    - (Opcional) Pedir confirmación antes de borrarlo.
5. **Creación de Pedidos**
    - Implementar la clase `Pedido` (o `Orden`) que incluya:
        - Lista de productos (o estructura `LineaPedido` con producto y cantidad).
    - **Solicitar** al usuario qué productos desea y en qué cantidad.
    - Verificar stock; si no alcanza, lanzar una **excepción** (por ejemplo `StockInsuficienteException`) o mostrar un mensaje de error.
    - Calcular el **costo total** (suma de `precio * cantidad` de cada producto).
    - **Disminuir el stock** de cada producto si el pedido se confirma.
    - (Opcional) Mostrar una lista de pedidos realizados.
6. **Menú Principal Interactivo**
    - Presentar un menú con **opciones** como:
        - Agregar producto
        - Listar productos
        - Buscar/Actualizar producto
        - Eliminar producto
        - Crear un pedido
        - (Opcional) Listar pedidos
        - Salir
    - El **programa** se repite hasta que el usuario elija “Salir”.
7. **POO y Principios de Diseño**
    - Dividir la lógica en **clases** y **métodos**:
        - `Producto`, `Pedido`, `Main` (para el menú), (opcional) `LineaPedido`, etc.
    - Emplear **encapsulamiento** (atributos privados, getters/setters).
    - (Opcional) Añadir **herencia/polimorfismo** si tu proyecto lo requiere (ej. `Bebida`, `Comida` que heredan de `Producto`).

8. **Excepciones**
    - Manejar con **try/catch** los errores de conversión de tipo, al ingresar valores no válidos, etc.
    - Crear al menos **una excepción personalizada** (por ej. `StockInsuficienteException` o `ProductoNoEncontradoException`) y lanzarla en el escenario adecuado.
9. **Organización en Paquetes y Módulos**
    - Separar las clases en **paquetes** lógicos (ej. `com.techlab.productos`, `com.techlab.excepciones`, etc.).
10. **Estructura del Código y Legibilidad**
    - Mantener un **código limpio**, con nombres descriptivos y funciones cortas.
    - Evitar métodos excesivamente largos o mezclar muchas responsabilidades en una sola clase.

Buenos Aires Ciudad