Teo

semana1

▼ 1era clase

Apunte-01.pdf

▼ Introducción en ingles

Backend Development with Java – Summary and Improved Writing

The **Backend Application Development with Java** course aims to provide students with a solid understanding of the concepts, techniques, and tools used in server-side (backend) application development with Java. Throughout the course, students will explore various frameworks, design patterns, best practices, and security measures relevant to backend development. The curriculum also covers deployment strategies and logging to ensure application efficiency and reliability.

Software Development Overview

In the **Software Development** course, students were introduced to real-world application programming and explored the layered software architecture model. This model structures an application into a set of interconnected layers, where each layer interacts with adjacent ones. While not the only approach, it reflects a common real-world model that students have already started implementing at a basic level.

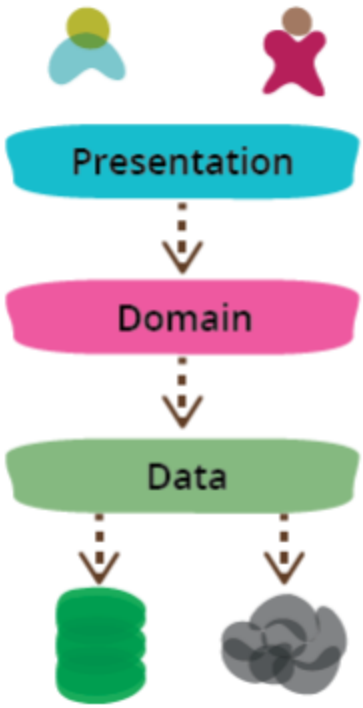
The simplified model includes four key layers: **Client**, **Frontend**, **Backend**, and **Database**. Students implemented a minimal version of each layer to understand their interactions and roles.

Layered Architecture (Stack)

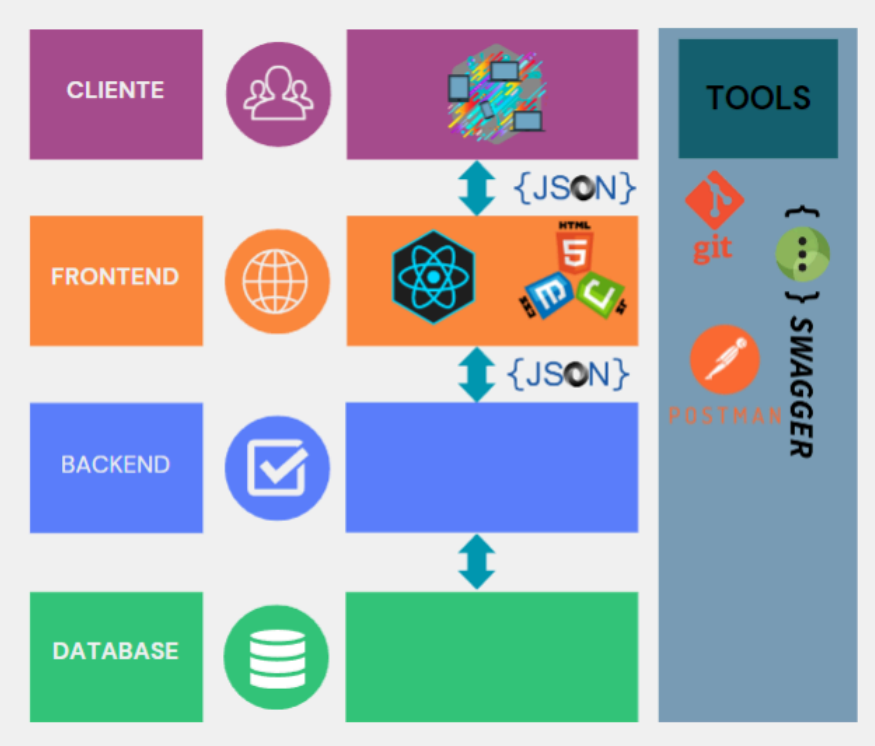
A software stack refers to the set of technologies used to build an application, where each layer is associated with a specific technology. Established stacks widely used in the industry include:

- **LAMP** (Linux, Apache, MySQL, PHP) – Traditionally used for open-source web applications.
- **MERN** (MongoDB, Express, React, Node.js) – A more modern stack for web applications.

In the Software Development course, students worked with a specific stack, represented visually in the course materials. The stack includes key technologies that will be explored in more depth as the course progresses. Technologies that will be replaced in the Backend Development course have been intentionally omitted from the diagram to avoid confusion.



Main Layers Explained



Client

The client refers to any device or application that connects to the backend, such as a web browser or a dedicated app. While the client layer is essential for user interaction, it is beyond the scope of this course, except in cases where mobile apps integrate the client and frontend directly with the backend.

Frontend

The frontend handles user interface (UI) development, data presentation, and user interaction. It manages both the visual styling and the logic needed to meet application requirements. In this course, the frontend will be built using tools for web, desktop, and mobile platforms.

Backend

The backend manages the business logic and rules governing the application. It ensures data consistency, controls access, and optimizes processes to serve multiple users efficiently. This course will focus

specifically on backend development, covering frameworks, security, scalability, and performance.

Database

The database layer handles data persistence and storage. It includes relational databases like MySQL or SQLite and external system integration when needed. Java's database connection methods are consistent across different relational databases.

Tools

The course will also cover essential development tools, including:

- Version control systems
- Testing tools
- Integrated development environments (IDEs)

Conclusion

This course builds on the foundational knowledge from the Software Development course, with a specific focus on backend development using Java. Students will have the opportunity to specialize in backend development or broaden their understanding of the full stack, equipping them with valuable skills for modern software development.

▼ introduccion

Introducción

Desarrollo Backend con Java – Resumen y Mejora de Escritura

El curso de **Desarrollo de Aplicaciones Backend con Java** tiene como objetivo proporcionar a los estudiantes una comprensión sólida de los conceptos, técnicas y herramientas utilizadas en el desarrollo de aplicaciones del lado del servidor (backend) con Java. A lo largo del curso, los estudiantes explorarán diversos frameworks, patrones de

diseño, buenas prácticas y medidas de seguridad relevantes para el desarrollo backend. El programa también cubre estrategias de implementación y registro (logging) para garantizar la eficiencia y confiabilidad de las aplicaciones.

Visión General del Desarrollo de Software

En el curso de **Desarrollo de Software**, los estudiantes fueron introducidos a la programación de aplicaciones en el mundo real y exploraron el modelo de arquitectura de software en capas. Este modelo estructura una aplicación en un conjunto de capas interconectadas, donde cada capa interactúa con las adyacentes. Aunque no es el único enfoque, refleja un modelo común en el mundo real que los estudiantes ya han comenzado a implementar a un nivel básico.

El modelo simplificado incluye cuatro capas clave: **Cliente**, **Frontend**, **Backend** y **Base de Datos**. Los estudiantes implementaron una versión mínima de cada capa para comprender sus interacciones y funciones.

Arquitectura en Capas (Stack)

Un stack de software se refiere al conjunto de tecnologías utilizadas para construir una aplicación, donde cada capa está asociada con una tecnología específica. Algunos stacks establecidos y ampliamente utilizados en la industria incluyen:

- **LAMP** (Linux, Apache, MySQL, PHP) – Tradicionalmente usado para aplicaciones web de código abierto.
- **MERN** (MongoDB, Express, React, Node.js) – Un stack más moderno para aplicaciones web.

En el curso de Desarrollo de Software, los estudiantes trabajaron con un stack específico, representado visualmente en el material del curso. Las tecnologías que serán reemplazadas en el curso de Desarrollo Backend se han omitido intencionalmente para evitar confusión.

Explicación de las Capas Principales

Cliente

El cliente se refiere a cualquier dispositivo o aplicación que se conecta al backend, como un navegador web o una aplicación dedicada. Aunque la capa cliente es esencial para la interacción con el usuario, está fuera del alcance de este curso, excepto en casos donde las aplicaciones móviles integren directamente el cliente y el frontend con el backend.

Frontend

El frontend maneja el desarrollo de la interfaz de usuario (UI), la presentación de datos y la interacción del usuario. Gestiona tanto el diseño visual como la lógica necesaria para cumplir con los requisitos de la aplicación. En este curso, el frontend se construirá utilizando herramientas para plataformas web, de escritorio y móviles.

Backend

El backend administra la lógica empresarial y las reglas que gobiernan la aplicación. Asegura la consistencia de los datos, controla el acceso y optimiza los procesos para servir de manera eficiente a múltiples usuarios. Este curso se centrará específicamente en el desarrollo backend, cubriendo frameworks, seguridad, escalabilidad y rendimiento.

Base de Datos

La capa de base de datos gestiona la persistencia y el almacenamiento de datos. Incluye bases de datos relacionales como MySQL o SQLite y la integración con sistemas externos cuando sea necesario. Los métodos de conexión a bases de datos en Java son consistentes entre diferentes bases de datos relacionales.

Herramientas

El curso también cubrirá herramientas de desarrollo esenciales, entre ellas:

- Sistemas de control de versiones
- Herramientas de prueba

- Entornos de desarrollo integrados (IDEs)

Conclusión

Este curso se basa en los conocimientos fundamentales adquiridos en el curso de Desarrollo de Software, con un enfoque específico en el desarrollo backend con Java. Los estudiantes tendrán la oportunidad de especializarse en el desarrollo backend o ampliar su comprensión del stack completo, adquiriendo habilidades valiosas para el desarrollo de software moderno.

▼ La capa de Backend ingles

Backend Layer – Summary and Improved Writing

The **backend** of an application consists of a set of services and components that work together to support the application's functionality. It serves as the foundation of the system's architecture, ensuring that the application is **scalable**, **secure**, easy to maintain, and capable of handling high traffic volumes.

Developing a backend involves addressing several key challenges, including **business logic**, **scalability**, **security**, **performance**, and **integration with external systems**. Successfully managing these aspects requires strong technical skills and well-defined development methodologies.

Key Challenges in Backend Development

1. Business Logic

Backend development involves implementing the business rules and processes that define how the system operates.

- A clear understanding of business requirements is essential to translate them into functional code.

- Business logic must be organized, modular, and easy to maintain as the application grows and evolves.

2. Scalability

Scalability refers to the ability of the backend to handle increased workload or user volume without sacrificing performance.

- Systems must be designed to scale **horizontally** (adding more servers) or **vertically** (increasing server capacity).
- Architecture and technology choices should allow for seamless scaling without compromising performance or increasing costs significantly.

3. Security

Security is critical for protecting user data and system resources from unauthorized access and malicious attacks.

- Strong security practices include **authentication**, **authorization**, **data encryption**, and **input validation**.
- A secure backend is essential to maintain data integrity and user trust.

4. Performance

Performance measures how quickly the backend can respond to user requests and maintain a smooth user experience.

- Optimizing database queries, reducing resource consumption, and managing workload distribution are key to improving performance.
- Regular performance testing helps identify and resolve bottlenecks.

5. Integration with External Systems

Many applications rely on external systems (e.g., web services, third-party APIs, legacy systems) to extend functionality and access external data.

- Successful integration requires managing compatibility, error handling, and versioning.
- The integration process should be reliable and tolerant to failures.

▼ Backend Internal Organization

To maintain clean and efficient code, the backend is organized into **modules** and **internal sublayers**. Each module handles a specific area of functionality within the application, while sublayers separate different responsibilities.

For example, in an **e-commerce system**, the backend might include modules for:

- **User management**
- **Product catalog**
- **Payment processing**
- **Reporting**

Each module could have sublayers such as:

- **Controllers** – Handle HTTP requests and responses.
- **Services** – Manage business logic.
- **Repositories** – Handle data access and interaction with the database.

Conclusion

Backend development involves tackling complex technical challenges related to business logic, scalability, security, performance, and system integration. This course aims to equip students with the knowledge and tools to address these challenges and make informed decisions when selecting and implementing backend solutions.

▼ Organización de Componentes en el Interior del Backend

Backend Component Organization

There are several ways to organize components within a backend, each with its own advantages and challenges. The choice of architecture impacts the scalability, maintainability, and complexity of the application. The most common backend architectures are:

---

1. Monolithic Architecture

In a monolithic architecture, all components are contained within a single deployable application.

- All services and functionalities share the same codebase and resources, allowing direct communication between components.
- While this approach is simple to develop and deploy, it can become difficult to scale and maintain as the application grows.

✅ **Advantages:**

- Easier to develop and test locally.
- Simplified deployment and monitoring.

❌ **Challenges:**

- Difficult to scale components independently.
- Tight coupling between modules makes updates and maintenance harder.

---

2. Microservices Architecture

Microservices architecture divides an application into small, independent, and specialized services.

- Each microservice handles a specific functionality and can be developed, deployed, and scaled independently.
- Microservices communicate through well-defined interfaces like HTTP APIs or message queues.

✅ **Advantages:**

- Independent scalability and deployment of services.

- Facilitates agile development and evolution of components.
- Greater fault tolerance — failure in one service doesn't affect others.

❌ **Challenges:**

- Increased complexity in development and management due to multiple services.
- Requires proper infrastructure for deployment and communication.
- More complex testing and integration due to the distributed architecture.

---

3. Hexagonal Architecture (Ports & Adapters)

Hexagonal architecture separates business logic (core) from data access and external system interaction using ports (interfaces) and adapters (implementations).

- Business logic is isolated from infrastructure, improving modularity and testability.

✅ **Advantages:**

- Easier to unit test and modify the core without affecting external systems.
- Flexible adaptation to different environments (e.g., databases, external services).

❌ **Challenges:**

- Requires more code and design effort to define interfaces and adapters.
- May be excessive for small or simple applications.

---

4. Service-Oriented Architecture (SOA)

SOA builds applications as a collection of independent services that communicate using standard protocols (e.g., HTTP, XML).

- Services encapsulate business logic and can be implemented using different technologies.
- Unlike microservices, SOA is often implemented as a monolithic application where services act as interoperability mechanisms.

✅ **Advantages:**

- Encourages service reuse and interoperability.
- Facilitates integration with external systems.

❌ **Challenges:**

- Can become complex in large applications due to the number of services.
- Requires a solid infrastructure for communication and coordination.

Conclusion

The backend plays a crucial role in modern software development by providing the logic, functionality, and data access that applications need to work efficiently and reliably. Choosing the right architecture and organizing internal modules effectively is essential for building scalable and maintainable applications.

This course will cover key topics related to microservices development in Java, including frameworks, design patterns, best practices, security, deployment, and logging — all essential for building high-quality backend applications.

▼ La capa de Backend

La capa de Backend

Capa de Backend – Resumen y Mejora de Escritura

El **backend** de una aplicación está compuesto por un conjunto de servicios y componentes que trabajan juntos para soportar la funcionalidad de la aplicación. Sirve como la base de la arquitectura del

sistema, asegurando que la aplicación sea **escalable**, **segura**, fácil de mantener y capaz de manejar grandes volúmenes de tráfico.

El desarrollo backend implica abordar varios desafíos clave, como la **lógica empresarial**, la **escalabilidad**, la **seguridad**, el **rendimiento** y la **integración con sistemas externos**. Gestionar con éxito estos aspectos requiere habilidades técnicas sólidas y metodologías de desarrollo bien definidas.

Desafíos Clave en el Desarrollo Backend

1. Lógica Empresarial

El desarrollo backend implica implementar las reglas y procesos empresariales que definen cómo opera el sistema.

- Comprender claramente los requisitos empresariales es esencial para traducirlos en código funcional.
- La lógica empresarial debe estar organizada, ser modular y fácil de mantener a medida que la aplicación crece y evoluciona.

---

2. Escalabilidad

La escalabilidad se refiere a la capacidad del backend para manejar un aumento en la carga de trabajo o el número de usuarios sin sacrificar el rendimiento.

- Los sistemas deben diseñarse para escalar de manera **horizontal** (agregando más servidores) o **vertical** (aumentando la capacidad del servidor).
- Las decisiones sobre arquitectura y tecnología deben permitir una escalabilidad sin afectar el rendimiento o aumentar los costos significativamente.

---

3. Seguridad

La seguridad es fundamental para proteger los datos de los usuarios y los recursos del sistema contra accesos no autorizados y ataques maliciosos.

- Las buenas prácticas de seguridad incluyen **autenticación**, **autorización**, **cifrado de datos** y **validación de entrada**.
- Un backend seguro es esencial para mantener la integridad de los datos y la confianza de los usuarios.

---

4. Rendimiento

El rendimiento mide la rapidez con la que el backend puede responder a las solicitudes de los usuarios y mantener una experiencia fluida.

- Optimizar las consultas a la base de datos, reducir el consumo de recursos y gestionar la distribución de carga son claves para mejorar el rendimiento.
- Las pruebas de rendimiento periódicas ayudan a identificar y resolver cuellos de botella.

---

5. Integración con Sistemas Externos

Muchas aplicaciones dependen de sistemas externos (por ejemplo, servicios web, APIs de terceros, sistemas heredados) para extender funcionalidades y acceder a datos externos.

- Una integración exitosa requiere gestionar la compatibilidad, el manejo de errores y el control de versiones.
- El proceso de integración debe ser confiable y tolerante a fallos.

Organización Interna del Backend

Para mantener un código limpio y eficiente, el backend se organiza en **módulos** y **sublayers internas**. Cada módulo maneja un área específica de funcionalidad dentro de la aplicación, mientras que las sublayers separan las diferentes responsabilidades.

Por ejemplo, en un sistema de **e-commerce**, el backend podría incluir módulos para:

- **Gestión de usuarios**

- **Catálogo de productos**
- **Procesamiento de pagos**
- **Informes**

Cada módulo podría tener las siguientes sublayers:

- **Controladores** – Manejan las solicitudes y respuestas HTTP.
- **Servicios** – Gestionan la lógica empresarial.
- **Repositorios** – Manejan el acceso a los datos y la interacción con la base de datos.

Conclusión

El desarrollo backend implica enfrentar desafíos técnicos complejos relacionados con la lógica empresarial, la escalabilidad, la seguridad, el rendimiento y la integración de sistemas. Este curso tiene como objetivo equipar a los estudiantes con los conocimientos y herramientas necesarios para abordar estos desafíos y tomar decisiones informadas al seleccionar e implementar soluciones backend.

Organización de Componentes en el Backend

Existen varias formas de organizar los componentes dentro de un backend, cada una con sus propias ventajas y desafíos. La elección de la arquitectura afecta la escalabilidad, mantenibilidad y complejidad de la aplicación. Las arquitecturas backend más comunes son:

---

1. Arquitectura Monolítica

En una arquitectura monolítica, todos los componentes están contenidos dentro de una única aplicación desplegable.

- Todos los servicios y funcionalidades comparten el mismo código base y recursos, lo que permite una comunicación directa entre los componentes.

- Aunque este enfoque es simple de desarrollar y desplegar, puede volverse difícil de escalar y mantener a medida que la aplicación crece.

✅ **Ventajas:**

✔️ Más fácil de desarrollar y probar localmente.

✔️ Despliegue y monitoreo simplificados.

❌ **Desafíos:**

❌ Difícil escalar los componentes de forma independiente.

❌ El fuerte acoplamiento entre módulos dificulta las actualizaciones y el mantenimiento.

---

2. Arquitectura de Microservicios

La arquitectura de microservicios divide una aplicación en pequeños servicios independientes y especializados.

- Cada microservicio maneja una funcionalidad específica y puede desarrollarse, desplegarse y escalarse de forma independiente.
- Los microservicios se comunican mediante interfaces bien definidas, como APIs HTTP o colas de mensajes.

✅ **Ventajas:**

✔️ Escalabilidad y despliegue independientes de los servicios.

✔️ Facilita el desarrollo ágil y la evolución de los componentes.

✔️ Mayor tolerancia a fallos: una falla en un servicio no afecta a los demás.

❌ **Desafíos:**

❌ Aumenta la complejidad en el desarrollo y la gestión debido a la multiplicidad de servicios.

❌ Requiere una infraestructura adecuada para el despliegue y la comunicación.

❌ Pruebas e integración más complejas debido a la arquitectura distribuida.

---

3. Arquitectura Hexagonal (Puertos y Adaptadores)

La arquitectura hexagonal separa la lógica empresarial (núcleo) del acceso a datos y la interacción con sistemas externos mediante puertos (interfaces) y adaptadores (implementaciones).

- La lógica empresarial está aislada de la infraestructura, lo que mejora la modularidad y la capacidad de prueba.

✅ **Ventajas:**

✔️ Facilita las pruebas unitarias y la modificación del núcleo sin afectar a los sistemas externos.

✔️ Adaptación flexible a diferentes entornos (por ejemplo, bases de datos, servicios externos).

❌ **Desafíos:**

❌ Requiere más código y esfuerzo de diseño para definir interfaces y adaptadores.

❌ Puede ser excesivo para aplicaciones pequeñas o simples.

---

4. Arquitectura Orientada a Servicios (SOA)

SOA construye aplicaciones como una colección de servicios independientes que se comunican mediante protocolos estándar (por ejemplo, HTTP, XML).

- Los servicios encapsulan la lógica empresarial y pueden implementarse utilizando diferentes tecnologías.
- A diferencia de los microservicios, SOA suele implementarse como una aplicación monolítica donde los servicios actúan como mecanismos de interoperabilidad.

✅ **Ventajas:**

✔️ Fomenta la reutilización de servicios e interoperabilidad.

✔️ Facilita la integración con sistemas externos.

❌ **Desafíos:**

❌ Puede volverse compleja en aplicaciones grandes debido al número de servicios.

❌ Requiere una infraestructura sólida para la comunicación y coordinación.

Conclusión

El backend juega un papel crucial en el desarrollo de software moderno al proporcionar la lógica, funcionalidad y acceso a datos que las aplicaciones necesitan para funcionar de manera eficiente y confiable.

Elegir la arquitectura adecuada y organizar los módulos internos de manera efectiva es esencial para construir aplicaciones escalables y fáciles de mantener.

Este curso cubrirá temas clave relacionados con el desarrollo de microservicios en Java, incluyendo frameworks, patrones de diseño, buenas prácticas, seguridad, despliegue y logging, todos esenciales para construir aplicaciones backend de alta calidad.

▼ Microservicios: Enfoque Detallado

Resumen e Informe Mejorado sobre Microservicios

Microservicios: Enfoque Detallado

La arquitectura de microservicios es ideal para aplicaciones complejas y de gran escala, ya que permite escalabilidad, despliegue independiente y flexibilidad tecnológica. Cada microservicio es una unidad autónoma que puede desarrollarse y mantenerse de manera independiente, lo que mejora la eficiencia y facilita la actualización y escalabilidad de los servicios.

Características de los Microservicios

1. **Independencia:**

Cada microservicio es autónomo, lo que permite desarrollar, probar, desplegar y escalar cada componente de forma independiente.

2. **Comunicación basada en APIs:**

   Los microservicios interactúan mediante APIs RESTful o colas de mensajes, asegurando interfaces bien definidas y estandarizadas.

3. **Escalabilidad:**

   Solo se escalan los servicios que requieren más recursos, lo que mejora el rendimiento y la eficiencia.

4. **Resiliencia:**

   Si un microservicio falla, los otros continúan operando, lo que mejora la disponibilidad y estabilidad del sistema.

5. **Flexibilidad tecnológica:**

   Cada microservicio puede implementarse con tecnologías distintas, eligiendo la mejor para cada funcionalidad.

Consideraciones en el Desarrollo de Microservicios

- **Gestión de datos:**

  Cada microservicio puede tener su propia base de datos o compartir una base de datos común, dependiendo de la cohesión e independencia de los datos.

- **Comunicación:**

  La comunicación puede ser sincrónica (mediante HTTP) o asincrónica (mediante eventos o colas de mensajes).

- **Gestión de errores y tolerancia a fallas:**

  Implementar estrategias como **reintentos** y **circuit breakers** para mejorar la resistencia.

- **Seguridad:**

  Aplicar autenticación y autorización en cada microservicio y asegurar las comunicaciones mediante HTTPS.

- **Monitorización y registro:**

  Implementar sistemas de logging y monitoreo para detectar y resolver problemas rápidamente.

Microservicios y Endpoints

En una arquitectura de microservicios, la aplicación se divide en múltiples servicios independientes que se comunican mediante **endpoints** bien definidos. Cada microservicio encapsula una funcionalidad específica e interactúa con otros servicios mediante estos endpoints.

Microservicio como Contenedor

- Un microservicio contiene todo lo necesario para su funcionamiento:
  - Lógica de negocio
  - Acceso a datos
  - Configuraciones
  - Dependencias

Ejemplo:

En una aplicación de comercio electrónico, los microservicios podrían dividirse en:

- **Usuarios** → Autenticación y perfiles
- **Catálogo** → Gestión de productos
- **Pagos** → Procesamiento de pagos
- **Notificaciones** → Envío de alertas y confirmaciones

Endpoints como Puntos de Entrada

Un endpoint define las operaciones que pueden realizarse sobre un microservicio, utilizando protocolos estándar como **HTTP/HTTPS**.

Ejemplo de endpoints para un microservicio de usuarios:

- `GET /users` → Obtener todos los usuarios

- `GET /users/{id}` → Obtener detalles de un usuario
- `POST /users` → Crear un nuevo usuario
- `PUT /users/{id}` → Actualizar un usuario
- `DELETE /users/{id}` → Eliminar un usuario

Interacción entre Microservicios

1. **Comunicación sincrónica:**
    - Una solicitud espera una respuesta antes de continuar.
    - Implementación típica mediante **APIs REST** o **llamadas HTTP**.
    - Ejemplo: el microservicio de pagos verifica un pago antes de confirmar una reserva.
2. **Comunicación asincrónica:**
    - El emisor no espera respuesta inmediata.
    - Implementación mediante **colas de mensajes** o **eventos**.
    - Ejemplo: el microservicio de reservas publica un evento cuando se crea una reserva, y el microservicio de notificaciones envía un mensaje al usuario.

Estructura de Endpoints en Microservicios

Definir correctamente los endpoints mejora la coherencia, eficiencia y mantenimiento del sistema.

Principios Básicos

✅ **Simplicidad y consistencia**

✅ **RESTful:** Uso de verbos HTTP estándar (GET, POST, PUT, DELETE)

✅ **Versionado:** Para manejar cambios sin afectar clientes existentes

---

Estrategias para Definir Endpoints

1. **Recursos:**

   Identificar claramente los recursos (usuarios, productos, pedidos).

2. **Acciones:**

   Utilizar métodos HTTP estándar:

   - `GET` → Obtener recursos
   - `POST` → Crear recursos
   - `PUT` / `PATCH` → Actualizar recursos
   - `DELETE` → Eliminar recursos

Ejemplo de Estructura de Endpoints

✅ Usuarios

- `GET /api/v1/users/{id}` → Obtener detalles de un usuario
- `POST /api/v1/users` → Crear un nuevo usuario
- `PUT /api/v1/users/{id}` → Actualizar un usuario
- `DELETE /api/v1/users/{id}` → Eliminar un usuario

✅ Productos

- `GET /api/v1/products` → Obtener productos
- `POST /api/v1/products` → Crear producto

✅ Pedidos

- `GET /api/v1/orders/{id}` → Obtener detalles de un pedido
- `POST /api/v1/orders` → Crear un pedido

Buenas Prácticas para Definir Endpoints

✅ Claridad y Consistencia:

- Utilizar nombres descriptivos y convenciones de nombramiento comunes.
- `✅ https://api.example.com/users`

- ❌ https://api.example.com/getUsers

✅ **Uso de sustantivos:**

- Prefiere sustantivos en plural para representar colecciones.

✅ **Minúsculas en URIs:**

- ✅ https://api.example.com/users
- ❌ https://api.example.com/Users

✅ **Evitar caracteres especiales:**

- ✅ https://api.example.com/users
- ❌ https://api.example.com/user%20details

✅ **Separación por barras (/) y guiones (-):**

- ✅ https://api.example.com/users/1234/first-name
- ❌ https://api.example.com/users-1234-first-name

✅ **Versionado:**

- ✅ https://api.example.com/v1/users
- ❌ https://api.example.com/users

✅ **CamelCase para parámetros:**

- ✅ https://api.example.com/users/{userId}
- ❌ https://api.example.com/users/{userid}

Resultados y Gestión de Datos

1. **Paginación:**

   Implementar paginación para resultados extensos para mejorar el rendimiento.

2. **Filtros y búsquedas:**

   Permitir parámetros para filtrar y buscar datos eficientemente.

3. **Manejo de errores:**

- Definir códigos HTTP claros (`200 OK`, `404 Not Found`, `500 Server Error`).
- Proveer mensajes de error informativos.

Seguridad y Documentación

✅ **Seguridad:**

- Implementar autenticación y autorización.
- Proteger las comunicaciones mediante HTTPS.

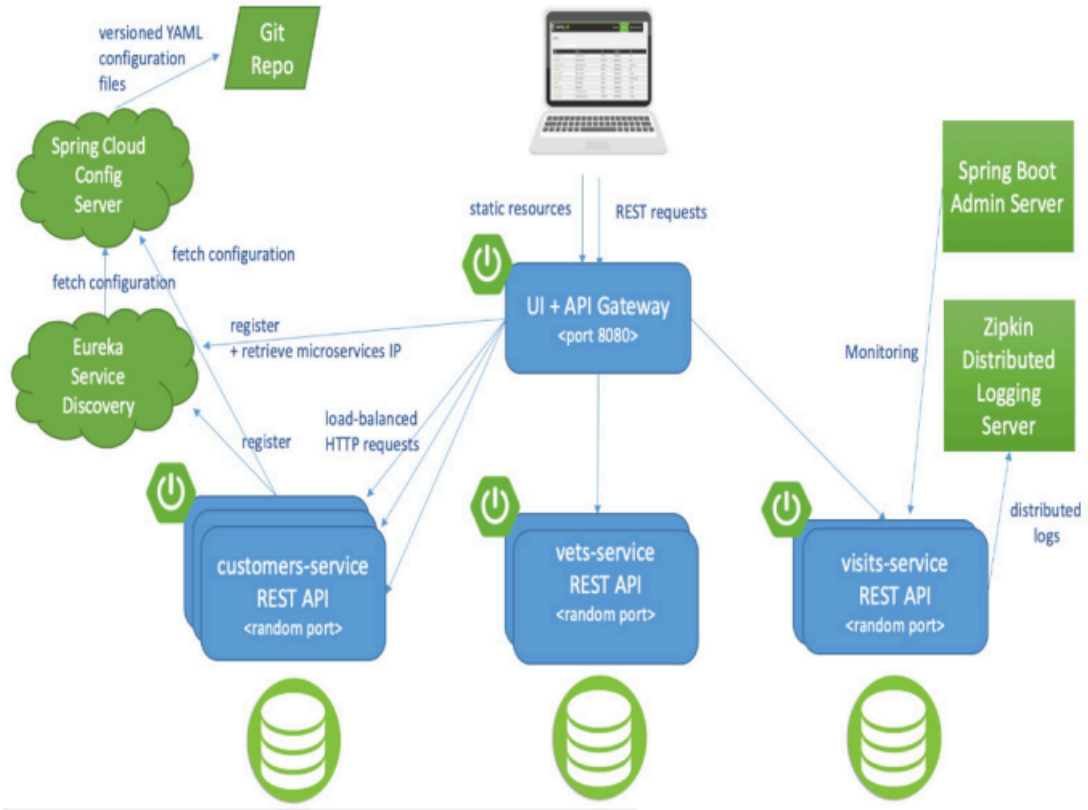
✅ **Documentación:**

- Documentar endpoints, parámetros y respuestas mediante herramientas como **Swagger**.

✅ Conclusión

Una arquitectura de microservicios bien diseñada mejora la flexibilidad, escalabilidad y resiliencia de las aplicaciones. La correcta definición y estructuración de endpoints garantiza una comunicación eficiente y segura entre los servicios.

▼ ejemplo



Resumen y Mejora del Texto

Conclusión - Ejemplo de aplicación: PetClinic

PetClinic es una aplicación de ejemplo desarrollada por **Spring** para mostrar cómo construir aplicaciones web robustas y escalables utilizando el framework Spring. Con el tiempo, se ha convertido en un referente en la comunidad de desarrollo de software, sirviendo como un modelo práctico para implementar diversas tecnologías y arquitecturas.

Evolución de PetClinic

1. **Aplicación Monolítica**
    - **Tecnologías:** Inicialmente, PetClinic era una aplicación monolítica construida con Spring Framework, integrando todas las funcionalidades en una sola aplicación desplegable.

- **Arquitectura:** Basada en capas tradicionales (presentación, negocio y acceso a datos) dentro de una misma aplicación.
- **Desventajas:**
  - Acoplamiento entre módulos.
  - Dificultad para escalar horizontalmente.
  - Complejidad en el despliegue.

2. **Migración a Spring Boot**
   - **Tecnologías:** Con la popularización de Spring Boot, PetClinic fue migrada para aprovechar la configuración automática y simplificar el desarrollo y despliegue.
   - **Arquitectura:** Aunque seguía siendo monolítica, el uso de Spring Boot permitió empaquetar la aplicación en un solo archivo JAR ejecutable y simplificó la integración con otras tecnologías de Spring.
   - **Ventajas:**
     - Desarrollo más rápido y eficiente.
     - Configuración simplificada.
3. **Arquitectura de Microservicios**
   - **Tecnologías:** Con la adopción de arquitecturas de microservicios, PetClinic fue rediseñada para fragmentarse en múltiples servicios especializados.
   - **Arquitectura:**
     - División en servicios independientes (gestión de clientes, veterinarios, visitas).
     - Comunicación mediante API REST.
     - Gestión mediante un API Gateway.
   - **Ventajas:**
     - Escalabilidad independiente de los servicios.

- Mejora de la resiliencia del sistema.
- Integración con prácticas de **DevOps**, **CI/CD** y **cloud computing**.

Importancia de PetClinic en la Comunidad

PetClinic ha servido como un modelo de referencia para aprender y aplicar diversas tecnologías de Spring, incluyendo:

- **Spring Framework:** Patrones como inyección de dependencias, AOP y transacciones.
- **Spring Boot:** Simplificación del desarrollo mediante la autoconfiguración.
- **Microservicios:** Transformación de una aplicación monolítica en una arquitectura escalable y resiliente.
- **DevOps:** Prácticas de CI/CD y automatización en aplicaciones Spring.

PetClinic continúa siendo actualizado para reflejar las mejores prácticas y nuevas tecnologías, consolidándose como una herramienta educativa clave para desarrolladores de todos los niveles.

Arquitectura de Microservicios en PetClinic

La arquitectura de microservicios en PetClinic se basa en Spring Boot y está compuesta por los siguientes componentes:

1. **Spring Cloud Config Server**
   - **Función:** Proporciona configuración centralizada obtenida de un repositorio Git.
   - **Ventaja:** Facilita la actualización y mantenimiento de la configuración desde un único punto.
2. **Eureka Service Discovery**
   - **Función:** Permite a los microservicios registrarse y descubrir otros servicios dinámicamente.

- **Ventaja:** Facilita el escalado horizontal y la detección automática de servicios disponibles.

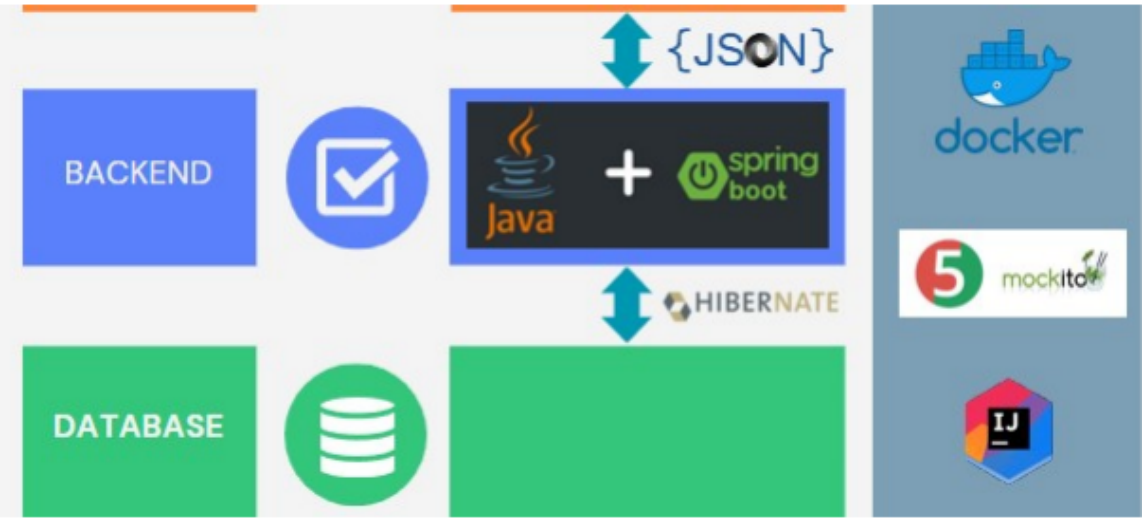
3. **API Gateway**
   - **Función:** Actúa como punto de entrada para las peticiones REST, redirigiendo las solicitudes a los microservicios.
   - **Ventaja:** Centraliza la autenticación, autorización y balanceo de carga.
4. **Microservicios (customers-service, vets-service, visits-service)**
   - **Función:** Cada servicio gestiona una funcionalidad específica (clientes, veterinarios y visitas).
   - **Ventaja:** Desacoplamiento, escalabilidad y mantenimiento independiente.
5. **Spring Boot Admin Server**
   - **Función:** Monitorea el estado de los microservicios registrados.
   - **Ventaja:** Permite revisar logs, métricas y estado general desde una interfaz central.
6. **Zipkin Distributed Logging Server**
   - **Función:** Recopila y muestra logs distribuidos para rastrear y analizar el flujo de solicitudes.
   - **Ventaja:** Mejora la observabilidad y facilita la detección de problemas de rendimiento.

Resumen

La arquitectura de microservicios en PetClinic representa una implementación robusta basada en Spring Boot. La división de la aplicación en servicios independientes mejora la escalabilidad, la resiliencia y la capacidad de respuesta en entornos de producción. Componentes como **Eureka**, **Spring Cloud Config** y el **API Gateway** proporcionan una infraestructura sólida, mientras que herramientas como **Spring Boot Admin** y **Zipkin** permiten una administración y monitoreo efectivos.

Aunque algunos de estos componentes no se utilizarán directamente en la asignatura, este material proporciona una visión general para facilitar el aprendizaje y comprensión de las tecnologías y técnicas asociadas.

▼ Tecnologías a utilizar en el desarrollo de Backend



Resumen y mejora del texto

Tecnologías para el desarrollo de Backend

Para implementar el backend basado en microservicios, utilizaremos un conjunto de tecnologías clave que proporcionarán la infraestructura necesaria para construir una solución robusta y escalable. A continuación, se describen brevemente las principales tecnologías que conforman el stack:

Tecnologías principales del Stack

- **Java**

  Java es un lenguaje de programación orientado a objetos, ampliamente utilizado en el desarrollo de aplicaciones empresariales debido a su portabilidad, facilidad de uso y robustez.

- **JDBC (Java Database Connectivity)**

  JDBC es una API de Java que permite la interacción con bases de datos relacionales mediante consultas y actualizaciones SQL.

- **Spring Framework**

  Framework de desarrollo para aplicaciones Java que proporciona una infraestructura completa para la inyección de dependencias, acceso a datos, seguridad, transacciones y otros servicios.

- **Spring Boot**

  Proyecto de Spring que simplifica la creación de aplicaciones Spring autónomas mediante configuración automática y predeterminada, permitiendo centrarse en el desarrollo de funcionalidades.

- **Spring Data**

  Módulo de Spring que simplifica el acceso y manipulación de datos, proporcionando una abstracción que permite trabajar con diferentes fuentes de datos (SQL y NoSQL) de manera uniforme.

- **Hibernate**

  Framework de mapeo objeto-relacional (ORM) que facilita la interacción con bases de datos relacionales a través de objetos Java, evitando la necesidad de escribir directamente consultas SQL.

- **Spring Security**

  Módulo de Spring que gestiona la seguridad de la aplicación mediante autenticación, autorización y protección contra ataques.

- **Spring Gateway (Spring Cloud Gateway)**

  Proporciona un enrutador/API Gateway para microservicios, facilitando el enrutamiento y filtrado de solicitudes entre servicios.

- **JUnit**

  Framework de pruebas unitarias para Java que permite automatizar la verificación del comportamiento de las clases y métodos de la aplicación.

- **Mockito**

Framework de pruebas para Java que permite crear objetos simulados (mocks) para facilitar la prueba de unidades independientes y simular el comportamiento de servicios externos.

- **IntelliJ IDEA**

  Entorno de Desarrollo Integrado (IDE) para Java que ofrece herramientas avanzadas de desarrollo, como autocompletado, depuración, pruebas y refactorización.

Resumen

El stack de backend incluye **Java** como lenguaje principal y el ecosistema de **Spring** para la infraestructura y gestión de servicios. **Hibernate** facilita el mapeo objeto-relacional, mientras que **Spring Security** garantiza la seguridad de la aplicación. **Spring Gateway** gestiona el enrutamiento entre microservicios, y **JUnit** y **Mockito** permiten realizar pruebas automatizadas. Finalmente, **IntelliJ IDEA** proporciona un entorno de desarrollo eficiente para crear una aplicación sólida, escalable y segura.

▼ Documentación de APIs

Resumen Mejorado: Documentación de APIs y Uso de Swagger

Importancia de la Documentación de APIs

Una documentación clara y completa es esencial para el éxito de cualquier proyecto basado en microservicios, ya que las APIs actúan como un puente entre distintos sistemas o componentes. Una API sin documentación es difícil de entender y usar, lo que limita su efectividad y dificulta la integración con otros sistemas.

Elementos clave de la documentación de una API:

1. **Descripción de Endpoints**: Explicar claramente el propósito y uso de cada endpoint.
2. **Parámetros de Request**: Definir los parámetros necesarios (tipo de dato, requeridos u opcionales) con ejemplos prácticos.
3. **Códigos de Estado y Respuestas**: Detallar las respuestas posibles (códigos HTTP y estructura de respuesta).

4. **Ejemplos de Uso**: Proporcionar ejemplos claros de solicitudes y respuestas para facilitar la implementación.

Una buena documentación mejora la integración, reduce errores y acelera el proceso de desarrollo al proporcionar una referencia clara para los desarrolladores.

Swagger y OpenAPI

Swagger es un conjunto de herramientas de código abierto que facilita el diseño, implementación y documentación de APIs RESTful mediante la especificación **OpenAPI**.

Beneficios de Swagger:

- Generación automática de documentación interactiva.
- Permite probar las llamadas a la API directamente desde la interfaz.
- Asegura consistencia y estandarización en la definición de las APIs.

Herramientas de Swagger:

- **Swagger Editor**: Editor web para definir APIs en formato OpenAPI.
- **Swagger UI**: Interfaz interactiva que permite probar y explorar las APIs.
- **Swagger Codegen**: Generador de código para cliente y servidor a partir de la definición de API.
- **Swagger Hub**: Plataforma para colaborar en el diseño y documentación de APIs.

OpenAPI

OpenAPI es una especificación estándar para describir la estructura y el comportamiento de una API RESTful mediante archivos YAML o JSON. Define endpoints, métodos HTTP, parámetros y respuestas, sirviendo como referencia única para el desarrollo e integración.

Estructura de un archivo OpenAPI:

1. **openapi**: Versión de la especificación (por ejemplo, `3.0.0`).

2. **info**: Metadatos de la API (nombre, descripción y versión).
3. **servers**: URLs de los entornos (desarrollo, prueba, producción).
4. **paths**: Definición de endpoints, métodos HTTP, parámetros y respuestas.
5. **components**: Modelos reutilizables y esquemas de seguridad.
6. **security**: Especificación de los mecanismos de autenticación y autorización.
7. **tags**: Organización de las operaciones mediante etiquetas.
8. **externalDocs**: Enlace a documentación adicional.

Uso de Swagger Editor

El **Swagger Editor** permite crear y validar definiciones OpenAPI en tiempo real:

- Validación en tiempo real para asegurar el cumplimiento de la especificación.
- Generación automática de documentación interactiva mediante Swagger UI.
- Creación de código base para cliente y servidor con Swagger Codegen.

Alternativas de uso:

- **Documentación de APIs existentes**: Documentar APIs ya implementadas para facilitar el mantenimiento y la integración.
- **Diseño de nuevas APIs**: Definir la API antes de implementarla para asegurar consistencia y permitir pruebas tempranas mediante mocks.

✅ Conclusión

Una documentación clara y detallada mejora la comprensión, la integración y el mantenimiento de las APIs. Swagger y OpenAPI proporcionan una

forma estandarizada y eficiente de documentar, probar y generar código para APIs RESTful, facilitando el desarrollo y la colaboración entre equipos.

objetivo: de esta lista de Tópicos de lectura, identificarlos por #heading 1 y ##heading 2 . En estos tópicos intentaremos cubrir tanto tecnología, como patrones de diseño, buenas prácticas y recomendaciones, cada grupo deberá interiorizarse a fondo en una de ellas a elección para generar un material

Temas sobre Java

Manejo de Excepciones en aplicaciones Java Descripción: El manejo de excepciones se refiere a la captura y gestión adecuada de situaciones inesperadas o errores en la aplicación.

Análisis FODA:
Fortalezas: Facilita la identificación y resolución de problemas en la aplicación.
Oportunidades: Mejora la experiencia del usuario al proporcionar mensajes de error claros y útiles.

Debilidades: Un mal manejo de excepciones puede ocultar problemas o generar inestabilidad en la
aplicación.
Amenazas: Excesivo uso de excepciones puede afectar el rendimiento y la legibilidad del código.
Link de donde sacar la informacion:
https://stackify.com/java-exception-handling-best-practices/

Java NIO

Java NIO (New I/O), también conocido como NIO, es una API (Interfaz de Programación de Aplicaciones) introducida en Java 1.4 que ofrece una forma más eficiente de realizar operaciones de entrada/salida (I/O) en comparación con la API clásica de E/S de Java. Java NIO está diseñado para manejar

múltiples conexiones y flujos de datos de manera eficiente, lo que lo hace especialmente útil para aplicaciones de red y servidores.
Fortalezas: Eficiencia, Manejo de Concurrencia y Flexibilidad de buffers.
Oportunidades: Optimización de accesos por red y Manejo de grandes volúmenes de datos.
Debilidades: Complejidad adicional y Posibles huecos de rendimiento.
Amenazas: Errores de concurrencia.
Link de donde sacar la informacion :
https://www.baeldung.com/java-nio

Modelo de hilos de Java y su aplicación tradicional en servidores

El Modelo de Hilos Tradicional de Java se refiere a la forma en que Java maneja la concurrencia y la ejecución de tareas en hilos separados dentro de una aplicación. Java utiliza la API de hilos proporcionada por la plataforma para permitir la ejecución simultánea de múltiples tareas y mejorar la eficiencia en sistemas multicore. Los hilos permiten que una aplicación realice operaciones en paralelo, lo que es esencial para aplicaciones con requerimientos de rendimiento, capacidad de respuesta y tareas asincrónicas.
Fortalezas: Paralelización eficiente, capacidad de respuesta y flexibilidad para tareas asincrónicas.
Oportunidades: Mejora de rendimiento y mejora de la experiencia de usuario.
Debilidades: Complejidad de Programación y Posibles huecos de seguridad.
Amenazas: Problemas de concurrencia y complicación del mantenimiento.
Link de donde sacar la informacion:
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Tema Especial desafío de producción propia

Proponemos realizar una revisión similar a los demás temas sobre el nuevo esquema de Hilos virtuales de Java 21, pero en lugar de revisar sus particularidades específicas, realizar una comparación con el esquema de hilos tradicional de java y sobre todo con el esquema de funcionamiento serializado de Node JS.

Temas sobre Microservicios

Spring Framework

Descripción: Spring es un framework de código abierto para el desarrollo de aplicaciones Java. Proporciona soporte integral para el desarrollo de aplicaciones empresariales, incluyendo la gestión de dependencias, la inyección de dependencias y la creación de servicios RESTful.

Análisis FODA:

Fortalezas: Amplia comunidad de usuarios, gran cantidad de módulos y extensibilidad.
Oportunidades: Facilita el desarrollo rápido y la integración con otros frameworks y librerías.
Debilidades: Curva de aprendizaje para principiantes.
Amenazas: Competencia de otros frameworks similares.
Link de donde sacar la informacion:
https://spring.io/

Spring Boot

Descripción: Spring Boot es un proyecto de Spring que simplifica la configuración y el despliegue de
aplicaciones Java. Permite crear aplicaciones autocontenidas con todas las dependencias empaquetadas.
Análisis FODA:
Fortalezas: Facilita el inicio rápido del desarrollo y el despliegue.
Oportunidades: Mejora la productividad del equipo y la eficiencia en el desarrollo.
Debilidades: Puede generar aplicaciones más pesadas debido a la inclusión de todas las dependencias.
Amenazas: Uso innecesario de características de Spring Boot puede aumentar la complejidad.
Link de donde sacar la informacion:
https://spring.io/projects/spring-boot

Spring Data JPA

Descripción: Spring Data JPA es un subproyecto de Spring que simplifica el acceso a datos mediante el uso de interfaces y convenciones de nomenclatura para las consultas. Análisis FODA:
Fortalezas: Facilita la interacción con la base de datos y reduce la cantidad de código necesario.
Oportunidades: Mejora la productividad y la mantenibilidad del acceso a datos.
Debilidades: Puede generar consultas SQL complejas en ciertos casos.
Amenazas: Requiere una adecuada gestión de la seguridad y la optimización de las consultas.
Link de donde sacar la informacion:
https://spring.io/projects/spring-data-jpa

Hibernate en profundidad

Descripción: Hibernate es un framework de mapeo objeto-relacional (ORM) que permite a los desarrolladores interactuar con una base de datos relacional utilizando objetos Java.
Análisis FODA:
Fortalezas: Facilita el acceso y manipulación de datos, evita código SQL manual.
Oportunidades: Mejora el rendimiento y la mantenibilidad de la capa de datos.
Debilidades: Puede generar consultas SQL complejas en ciertos casos.
Amenazas: Puede generar sobrecarga en la base de datos si no se usa adecuadamente.
Link de donde sacar la informacion:
https://hibernate.org/

Spring Security

Descripción: Spring Security es un módulo de seguridad de Spring que proporciona herramientas para implementar autenticación, autorización y protección contra ataques comunes en aplicaciones Java. Análisis FODA:
Fortalezas: Facilita la implementación de medidas de seguridad robustas.
Oportunidades: Mejora la confianza y la credibilidad del sistema.

Debilidades: Requiere una configuración adecuada y un buen entendimiento del funcionamiento.
Amenazas: Vulnerabilidades en la configuración pueden exponer el sistema a ataques.
Link de donde sacar la informacion:
https://spring.io/projects/spring-security

Spring Cloud

Descripción: Spring Cloud es un conjunto de herramientas para facilitar la construcción de sistemas
distribuidos basados en microservicios.
Análisis FODA:
Fortalezas: Facilita la gestión de configuración, registro de servicios y balanceo de carga.
Oportunidades: Mejora la escalabilidad y la resiliencia de los microservicios.
Debilidades: Requiere una curva de aprendizaje adicional para dominar todas las herramientas.
Amenazas: Uso innecesario de Spring Cloud puede agregar complejidad innecesaria al sistema.
Link de donde sacar la informacion:
https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop

Spring Cloud Config

Descripción: Spring Cloud Config es una herramienta de Spring Cloud que proporciona un servidor
centralizado para la gestión de configuraciones de microservicios.
Análisis FODA:
Fortalezas: Centraliza y simplifica la gestión de configuraciones en un sistema distribuido.
Oportunidades: Facilita la actualización y el monitoreo de configuraciones en tiempo real.
Debilidades: Requiere una infraestructura adecuada y una gestión adecuada de la seguridad.

Amenazas: Configuraciones incorrectas pueden afectar la estabilidad del sistema y exponer datos
sensibles.
Link de donde sacar la informacion:
https://docs.spring.io/spring-cloud-config/docs/current/reference/html/

Temas sobre Patrones y buenas prácticas

RESTful API

Descripción: Una API RESTful (Representational State Transfer) es una interfaz de programación de
aplicaciones que sigue los principios de REST. Permite a las aplicaciones comunicarse y transferir datos utilizando métodos HTTP estándar como GET, POST, PUT y DELETE.
Análisis FODA:
Fortalezas: Interoperabilidad, escalabilidad y simplicidad en el diseño.
Oportunidades: Facilita la integración con diferentes tecnologías y plataformas.
Debilidades: Puede resultar menos eficiente para transferencias grandes de datos.

Amenazas: Requiere una buena gestión de la seguridad y la autenticación.
Link de donde sacar la informacion:
https://spring.io/guides/gs/rest-service/

Patrón de diseño MVC

Descripción: El patrón Modelo-Vista-Controlador (MVC) es un patrón de diseño que separa la lógica de la aplicación en tres componentes: Modelo (representación de los datos), Vista (interfaz de usuario) y Controlador (manejo de la interacción con el usuario).
Análisis FODA:
Fortalezas: Facilita la separación de responsabilidades, mejora el mantenimiento.

Oportunidades: Permite el desarrollo concurrente de diferentes capas del sistema.
Debilidades: Puede generar complejidad si no se aplica correctamente.
Amenazas: Mal uso o sobrediseño del patrón puede llevar a una excesiva fragmentación del código.
Link de donde sacar la informacion:
https://www.oracle.com/technical-resources/articles/javase/mvc.html

Seguridad en Backend

Descripción: La seguridad en backend se refiere a la implementación de medidas y prácticas para proteger la aplicación de posibles vulnerabilidades y ataques maliciosos.
Análisis FODA:
Fortalezas: Protege la aplicación y los datos sensibles de posibles ataques.
Oportunidades: Asegura la confidencialidad, integridad y disponibilidad de la aplicación.
Debilidades: Puede generar sobrecarga en el rendimiento de la aplicación.
Amenazas: Falta de seguridad puede poner en riesgo la información y la reputación de la empresa.
Link de donde sacar la informacion:
https://www.oracle.com/java/technologies/javase/seccodeguide.html

Patrón de diseño DAO

Descripción: El patrón Data Access Object (DAO) se utiliza para separar la lógica de acceso a datos de la lógica de negocio. Proporciona una capa de abstracción entre la aplicación y la base de datos. Análisis FODA:
Fortalezas: Mejora la mantenibilidad y reutilización de la lógica de acceso a datos.
Oportunidades: Permite cambiar la base de datos subyacente sin afectar la lógica de negocio.
Debilidades: Puede generar código repetitivo y aumentar la complejidad.
Amenazas: Una mala implementación puede afectar el rendimiento de la aplicación.

Link de donde sacar la informacion:
https://www.baeldung.com/java-dao-pattern

Patrón de diseño Repository

El Patrón Repositorio es un patrón de diseño de software que se utiliza en el desarrollo de aplicaciones para separar la lógica de acceso a datos de la lógica de negocio. Proporciona una abstracción de la fuente de datos subyacente, como una base de datos, archivos o servicios web, a través de interfaces y clases de repositorio. Esto permite a los desarrolladores interactuar con los datos de manera coherente y centralizada,independientemente de cómo se almacenen o recuperen. El Patrón Repositorio mejora la modularidad, la reutilización del código y la flexibilidad al cambiar la fuente de datos sin afectar la lógica de la aplicación.
Fortalezas: Separación de responsabilidades, abstracción del origen de datos y reutilización de código.
Oportunidades: Mejora en las pruebas y Flexibilidad en la capa de datos.
Debilidades: Complejidad adicional, sobrecarga en proyectos pequeños y curva de aprendizaje.
Amenazas: Sbreingeniería y menejo de la eficiencia.
Link de donde sacar la informacion :
https://platzi.com/blog/patron-repositorio/

Patrón de diseño Singleton

Descripción: El patrón de diseño Singleton garantiza que una clase tenga una única instancia en todo el sistema, proporcionando un punto global de acceso a esta instancia. Análisis FODA:
Fortalezas: Mejora la eficiencia y el rendimiento al evitar instancias innecesarias.
Oportunidades: Útil en escenarios donde se necesita una única instancia de un objeto compartido.
Debilidades: Puede generar problemas de concurrencia y dificultar las pruebas unitarias.
Amenazas: Abuso del patrón puede acoplar de manera innecesaria

componentes del sistema.
Link de donde sacar la informacion:
https://www.geeksforgeeks.org/singleton-design-pattern/

Auditoría y Generación de Logs

Descripción: La auditoría y la generación de logs son prácticas importantes para rastrear acciones y eventosen la aplicación, lo que facilita la identificación y solución de problemas. Análisis FODA:
Fortalezas: Facilita la trazabilidad y la resolución de incidencias.
Oportunidades: Mejora la seguridad y la confiabilidad del sistema.
Debilidades: Puede generar un mayor consumo de recursos si no se gestiona adecuadamente.
Amenazas: Exceso de logs puede afectar la privacidad y la seguridad de los datos.
Link de donde sacar la informacion:
https://www.baeldung.com/java-logging-intro

Patrón de diseño Factory

Descripción: El patrón Factory se utiliza para crear objetos sin exponer la lógica de creación. Proporciona una interfaz común para la creación de objetos de diferentes tipos.
Análisis FODA:
Fortalezas: Simplifica la creación de objetos y mejora la modularidad.
Oportunidades: Facilita la adición de nuevos tipos de objetos sin afectar el código existente.
Debilidades: Aumenta la complejidad del código en aplicaciones pequeñas y sencillas.
Amenazas: Un uso excesivo del patrón puede afectar el rendimiento y la legibilidad del código.
Link de donde sacar la informacion:
https://www.baeldung.com/java-factory-method

Testing en Backend

Descripción: Las pruebas en backend son esenciales para garantizar el correcto funcionamiento de los servicios y componentes. Pueden ser pruebas unitarias o de integración.
Análisis FODA:
Fortalezas: Mejora la calidad del software y permite detectar errores tempranamente.
Oportunidades: Facilita el mantenimiento y la refactorización del código.
Debilidades: Requiere recursos y tiempo adicionales para implementar las pruebas.
Amenazas: Pruebas insuficientes pueden resultar en la entrega de código defectuoso.
Link de donde sacar la informacion:
https://www.baeldung.com/spring-boot-testing

Patrón de diseño Observer

Descripción: El patrón Observer permite a objetos interesados suscribirse y recibir notificaciones cuando un objeto observado cambia su estado.
Análisis FODA:
Fortalezas: Mejora la comunicación y la interacción entre objetos en tiempo de ejecución.
Oportunidades: Facilita la implementación de comportamientos reactivos en el sistema.
Debilidades: Puede aumentar la complejidad si no se aplica de manera adecuada.
Amenazas: Un mal uso del patrón puede generar un alto acoplamiento entre objetos.
Link:
https://www.baeldung.com/java-observer-pattern

Manejo de Caché

Descripción: El manejo de caché consiste en almacenar datos en memoria para evitar acceder repetidamente a la fuente original, mejorando así el rendimiento.
Análisis FODA:
Fortalezas: Mejora la velocidad de acceso a datos y reduce la carga en la base de datos.
Oportunidades: Aumenta la eficiencia en aplicaciones con datos estáticos o de acceso frecuente.
Debilidades: Puede generar problemas de consistencia y requerir una gestión adecuada del caché.
Amenazas: Datos desactualizados en caché pueden afectar la integridad del sistema.
Link de donde sacar la informacion:
https://www.oracle.com/technical-resources/articles/java/caching.html

Patrón de diseño Gateway

Descripción: El patrón Gateway se utiliza para centralizar la lógica de acceso y seguridad en un punto de entrada único para los microservicios.
Análisis FODA:
Fortalezas: Simplifica el acceso y la autenticación a los servicios, mejora la seguridad.
Oportunidades: Facilita la gestión de políticas de acceso y registro de actividad.
Debilidades: Introduce un único punto de fallo si no se implementa correctamente.
Amenazas: Mal diseño o implementación pueden afectar la latencia y el rendimiento.
Link de donde sacar la informacion:
https://sourcemaking.com/design_patterns/gateway

semana2

▼ 2da clase notio tablet

maeven es un gestor de paquetes

pom.xml → par agg dependencias

scanner, → para lee por tecladoo primero creabamos la clase y luego utilizamos el metodo

ejercicos que hico estan en semana 2 materales adicionales y actividades java y a maven

▼ 2da clase

Apunte 02

Maeven

3er pedf

▼ Introducción a la Plataforma Java

▼ Breve referencia histórica de la Plataforma Java

Aquí tienes un resumen de la historia de la Plataforma Java:

Breve referencia histórica de la Plataforma Java

A inicios de los años 90, el lenguaje C++ era el preferido por los desarrolladores. En 1991, un equipo de Sun Microsystems, liderado por James Gosling y Patrick Naughton, inició *The Green Project* para desarrollar un lenguaje adaptable a diversos dispositivos. Inicialmente llamado *7* y luego *Oak*, terminó nombrándose **Java**, supuestamente inspirado en el café de una cafetería.

El diseño de Java priorizó la **portabilidad**, permitiendo ejecutar programas en distintos sistemas sin recompilar. Aunque inicialmente se

pensó para electrodomésticos, el auge de Internet en 1991 cambió su destino. Java se popularizó con los **applets**, que permitían crear páginas web interactivas, superando las limitaciones del HTML de la época.

Con el tiempo, otros lenguajes y tecnologías como JavaScript y Flash reemplazaron a los applets, pero Java se consolidó en el desarrollo empresarial. Actualmente, es un pilar fundamental para el desarrollo de **backends** en aplicaciones empresariales, destacándose por su independencia de plataforma y su integración con diversos sistemas y tecnologías.

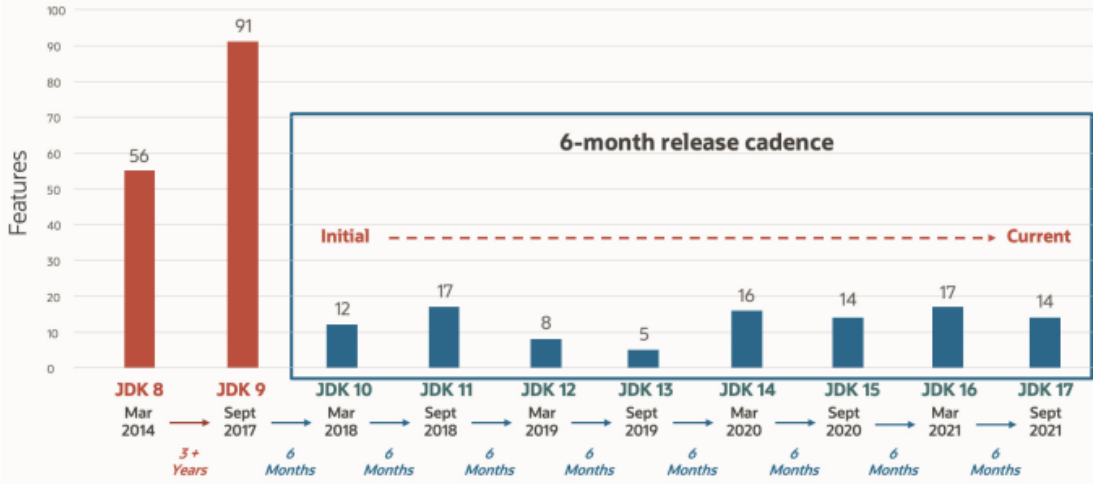
▼ Versiones de Java

Desde su lanzamiento en **1996**, Java ha evolucionado con múltiples versiones, incorporando mejoras y nuevas APIs. Inicialmente, las versiones llevaban números como **Java 1.0** (1996) y **Java 1.2** (1998), que introdujo **Swing** y consolidó a Java en el desarrollo empresarial.

En **2004**, con **Java 5.0**, se introdujeron cambios significativos en el lenguaje. Luego, en **2006**, **Java 6** eliminó la denominación **J2SE** en favor de **Java SE**.

Tras la adquisición de **Sun por Oracle (2010)**, se estableció un esquema de versiones con **soporte a largo plazo (LTS)**. Desde **Java 11**, Oracle introdujo licencias comerciales, impulsando el uso de **OpenJDK** como alternativa gratuita.

Actualmente, la versión **Java SE 21** es LTS (soporte hasta **2031**), mientras que **Java SE 25**, la próxima LTS, está prevista para **septiembre de 2025**.



▼ Ediciones de Java

Ediciones de Java (Resumen breve)

Java se divide en varias ediciones, cada una orientada a distintos usos:

- **Java SE (Standard Edition)**: Versión principal, incluye herramientas y APIs esenciales como colecciones, E/S, bases de datos y concurrencia.
- **Java EE → Jakarta EE (Enterprise Edition)**: Enfocada en aplicaciones empresariales, ahora gestionada por la Fundación Eclipse. Se divide en:
  - **Web Profile**: Tecnologías web (Servlets, JSF, JPA, CDI).
  - **Full Platform Profile**: Incluye lo anterior más EJB, JMS y JTA.
  - **MicroProfile**: Optimizado para microservicios con métricas, tolerancia a fallos y seguridad.
- **Java ME (Micro Edition)**: Orientada a dispositivos IoT.
- **Java Card**: Para tarjetas inteligentes y dispositivos seguros.
- **Java FX**: Plataforma moderna para crear interfaces gráficas avanzadas, ahora parte de OpenJFX.

▼ El lenguaje de Programación Java

Resumen: El Lenguaje de Programación Java

Java es el lenguaje central de **Java SE** y se usa en todas sus ediciones. Es un lenguaje único con similitudes a **C#**, pero con características propias que lo hacen destacar.

Características principales:

- **De alto nivel**: Más cercano al lenguaje humano y con amplias librerías predefinidas.
- **Orientado a objetos**: Todo en Java son objetos, a diferencia de lenguajes como C++, que se adaptaron a la POO.
- **Independiente de la plataforma**: Gracias a la **JVM**, el mismo código funciona en distintos sistemas.
- **Compilado e interpretado**: Se compila en **bytecode**, ejecutado por la JVM sin necesidad de recompilarlo para cada sistema.
- **Gestión automática de memoria**: Usa un **garbage collector** para liberar memoria de objetos no utilizados.
- **Multihilos**: Permite la ejecución paralela de procesos para mejorar el rendimiento.

▼ Hola Mundo en Java

Aquí tienes un resumen de los puntos clave:

El Lenguaje de Programación Java

- Java es un lenguaje de alto nivel, orientado a objetos y multiplataforma.
- Se compila e interpreta, permitiendo independencia del sistema operativo gracias a la **JVM** (Máquina Virtual de Java).
- Gestiona automáticamente la memoria con un recolector de basura.
- Soporta **multihilos** para ejecución en paralelo.

Hola Mundo en Java

- Se escribe en un archivo `.java`, cuyo nombre debe coincidir con la clase principal.

- Es un lenguaje **case sensitive**, usa **llaves** `{}` para delimitar bloques y requiere `;` al final de cada sentencia.
- Se compila con `javac` y se ejecuta con `java <NombreClase>`.

Compilación y Bytecode

- **Compilación**: El código fuente `.java` se transforma en **bytecode** (.class) con `javac`.
- **Bytecode**: Código intermedio independiente del sistema operativo, ejecutado por la JVM.
- **Ventajas**: Portabilidad, seguridad y eficiencia.

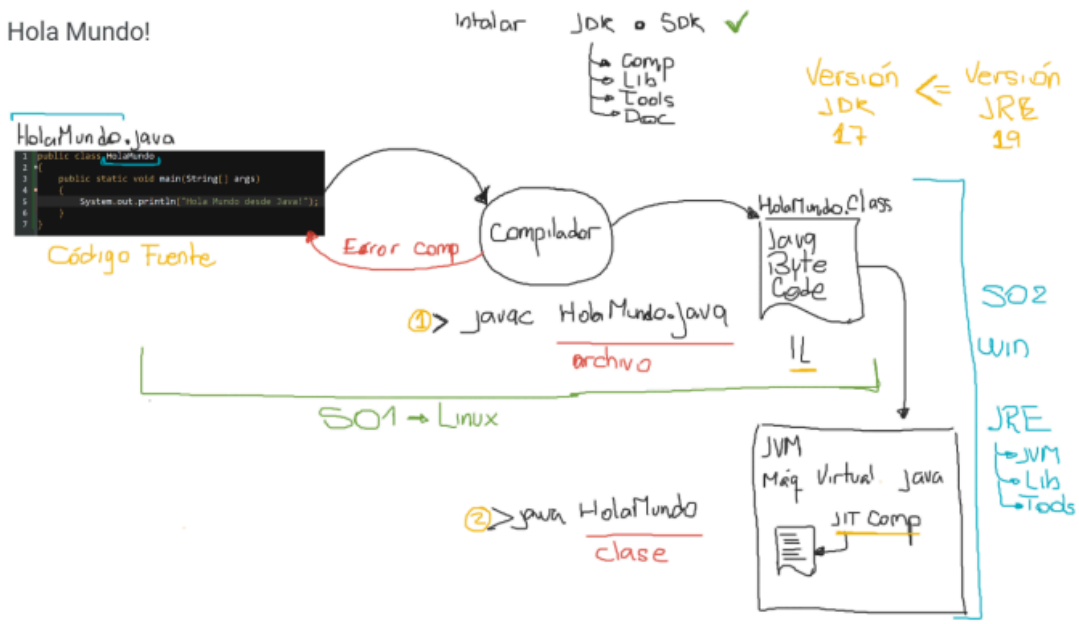
Ejecución en Diferentes Plataformas

- **JDK** (Java Development Kit): Incluye herramientas para programar, compilar y ejecutar código.
- **JRE** (Java Runtime Environment): Solo permite ejecutar programas ya compilados.
- La JVM permite ejecutar programas Java en cualquier sistema operativo sin necesidad de recompilar.

Multiplataforma

- Un programa Java se compila una vez y se ejecuta en cualquier plataforma con una JVM instalada.
- Esto permite que Java sea utilizado en múltiples dispositivos sin modificaciones al código fuente.

Este resumen sintetiza los conceptos esenciales sobre la programación en Java y su ejecución en distintos entornos.



▼ Breve debate sobre las diferentes JVMs existentes

Resumen: Breve debate sobre las diferentes JVMs existentes

Existen diversas máquinas virtuales de Java (JVM), cada una con características específicas:

1. **HotSpot VM (Oracle/OpenJDK)**
   - **Ventajas:** Alto rendimiento, optimización Just-In-Time (JIT) y recolección de basura adaptativa.
   - **Desventajas:** Mayor consumo de memoria y latencia de inicio.
2. **OpenJ9 (Eclipse Adoptium, antes IBM J9)**
   - **Ventajas:** Bajo consumo de memoria, ideal para entornos con recursos limitados.
   - **Desventajas:** Puede requerir ajustes en algunas aplicaciones para compatibilidad.
3. **GraalVM**

- **Ventajas:** Soporta múltiples lenguajes, permite compilar aplicaciones Java en imágenes nativas.
- **Desventajas:** Configuración más compleja en comparación con JVMs tradicionales.

4. **Azul Zing**
   - **Ventajas:** Optimizado para baja latencia y rendimiento en tiempo real.
   - **Desventajas:** Uso más específico y costos elevados.
5. **SAP Machine**
   - **Ventajas:** Estabilidad y optimización para aplicaciones SAP.
   - **Desventajas:** Enfocado en un nicho específico.

La elección de la JVM dependerá de factores como rendimiento, consumo de recursos y necesidades del proyecto.

▼ MAVEN

▼ Introduccion

Resumen: Apunte de clase 3 - Maven

Maven es una herramienta de gestión y construcción de proyectos que automatiza tareas comunes en el desarrollo de aplicaciones Java. Su propósito es estructurar proyectos, administrar dependencias y orquestar su ciclo de vida.

1. **Generador de estructuras de proyecto**
   - Utiliza *archetypes* (plantillas) para crear proyectos con una estructura estándar.
   - Define la organización de archivos y el archivo de configuración *pom.xml*.
2. **Administrador de dependencias**
   - Permite declarar dependencias en *pom.xml* para su descarga automática.

- Garantiza el uso coherente de versiones mediante repositorios remotos como Maven Central.

3. **Orquestador del ciclo de vida del proyecto**
   - Coordina desde la compilación hasta la generación de artefactos (*JAR*, *WAR*).
   - Usa comandos como `mvn compile`, `mvn test` y `mvn package` para automatizar tareas.

Maven es una herramienta clave en el desarrollo Java y en entornos DevOps, mejorando la eficiencia y estandarización del proceso de desarrollo.

▼ Un poco más de detalle sobre las Funciones de Maven

Resumen de las Funciones de Maven

1. Generador de Estructuras de Proyecto

- **Arquetipos**: Plantillas para crear proyectos Java con estructuras predefinidas (web, consola, bibliotecas).
- **Artefacto**: Producto generado por Maven (JAR, WAR), definido por:
  - `GroupId`: Identificador del grupo (ej. dominio al revés).
  - `ArtifactId`: Nombre del proyecto.
  - `Version`: Número de versión (ej. `1.0.0-SNAPSHOT`).

2. Administrador de Dependencias

- **Dependencias**: Bibliotecas externas utilizadas en el proyecto.
- **Problema del "DLL Hell"**: Conflictos de versiones resueltos con Maven.
- **Gestión de Dependencias en Maven**:
  - Se declaran en el `pom.xml`.
  - **Repositorios**: Remotos (ej. Maven Central) y locales (`.m2`).

- **Dependencias transitivas**: Descargadas automáticamente.

3. Orquestación del Ciclo de Construcción

- **Fases clave**:
  - `clean`: Limpia archivos generados.
  - `validate`: Verifica dependencias.
  - `compile`: Compila el código.
  - `test`: Ejecuta pruebas.
  - `package`: Empaqueta (JAR/WAR).
  - `install`: Instala el artefacto en el repositorio local.
  - `deploy`: Publica el artefacto en un repositorio remoto.

4. Archivo `pom.xml`

- **Estructura de directorios estándar**:

```
hola-mundo/
├── src/main/java/
├── src/main/resources/
├── src/test/java/
├── src/test/resources/
└── pom.xml
```

- **Elementos clave**:
  - `<groupId>`, `<artifactId>`, `<version>`, `<dependencies>`, `<build>`, `<plugins>`.

5. Instalación de Maven

- **Windows**:
  1. Descargar y extraer Maven.
  2. Configurar variables de entorno (`M2_HOME`, `Path`).
- **Linux/macOS**:

1. Descargar y extraer con `wget` y `tar`.
2. Configurar variables en `.bashrc` o `.zshrc`.

- **Verificación**: `mvn --version`.

6. Directorio `.m2`

- Ubicado en `C:\Users\miUsuario\.m2` (Windows) o `/home/miUsuario/.m2` (Linux/macOS).
- Contiene `settings.xml` para configuraciones personalizadas (opcional).

👉 **Con Maven correctamente instalado, se pueden gestionar proyectos Java de manera eficiente.**

▼ Hola mundo con Maven

Aquí tienes un resumen del proceso de creación y ejecución de un proyecto "Hola Mundo" en Java usando Maven:

1. **Creación del Proyecto**:
   - Se utiliza el arquetipo `maven-archetype-simple` con el comando:

     ```
     mvn archetype:generate "-DgroupId=ar.edu.utnfc.backend" "-DartifactId=hola-mundo" "-DarchetypeGroupId=org.apache.maven.archetypes" "-DarchetypeArtifactId=maven-archetype-simple" "-DinteractiveMode=false"
     ```

   - En PowerShell, los argumentos deben ir entre comillas dobles.
2. **Estructura del Proyecto**:
   - Se genera un directorio con el nombre del artefacto (`hola-mundo`).
   - Contiene el archivo `pom.xml` y las carpetas estándar:
     - `src/main/java`: Código fuente (incluye `App.java` con el `Hola Mundo`).
     - `src/test/java`: Código de pruebas unitarias.
     - `target`: Se genera tras la compilación.
3. **Compilación del Proyecto**:

- Se ejecuta el comando:

```
mvn compile
```

- Genera la carpeta `target` con los archivos `.class` compilados.

4. **Empaquetado del Proyecto**:
   - Se modifica `pom.xml` para indicar la clase principal en el `maven-jar-plugin`:

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
    <archive>
      <manifest>
        <mainClass>ar.edu.utnfc.backend.App</mainClas
s>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

   - Se genera el `.jar` ejecutando:

```
mvn package
```

   - El archivo resultante `hola-mundo-1.0-SNAPSHOT.jar` se encuentra en `target/`.

5. **Ejecución del Programa**:
   - Para ejecutar el `.jar`, se usa:

```
java -jar hola-mundo-1.0-SNAPSHOT.jar
```

   - Esto ejecuta el programa y muestra el mensaje "Hola Mundo".

▼ Repositorios de dependencias

Resumen sobre Repositorios en Maven

Maven utiliza distintos tipos de repositorios para gestionar dependencias y artefactos a lo largo del ciclo de vida del proyecto. Los principales son:

1. **Repositorio Central**:
    - Es el repositorio global mantenido por la comunidad Maven.
    - Contiene una gran cantidad de bibliotecas y artefactos Java de uso común.
    - Maven descarga automáticamente las dependencias declaradas en `pom.xml` desde aquí.
2. **Repositorio Remoto**:
    - Es un servidor externo donde se almacenan dependencias y artefactos.
    - Se usa para acceder a bibliotecas específicas o trabajar en entornos con restricciones de red.
    - Puede configurarse en `pom.xml` (para un proyecto) o en `settings.xml` (para todos los proyectos en la PC).
3. **Repositorio Local o Caché (`.m2`)**:
    - Es un directorio en el sistema local donde Maven almacena las dependencias descargadas.
    - Ubicado en la carpeta `.m2` dentro del directorio del usuario.
    - Permite reutilizar dependencias sin necesidad de descargarlas nuevamente, facilitando el trabajo sin conexión.

Comparación entre el Repositorio Local y la Caché de Dependencias Remotas

- **Repositorio Local (`.m2`)**
    - Almacena todas las dependencias descargadas en el sistema.

- Evita descargas repetidas y mejora la eficiencia.
- Permite trabajar en proyectos sin conexión a Internet.
- Facilita la colaboración entre miembros de un equipo.

- **Caché de Dependencias Remotas**
  - Guarda temporalmente dependencias obtenidas de repositorios remotos.
  - Maven primero busca en la caché antes de descargar nuevamente.
  - Optimiza el rendimiento al reducir las solicitudes a servidores remotos.

Conclusión

Los repositorios en Maven (central, remoto y local) junto con la carpeta `.m2` son fundamentales para la gestión eficiente de dependencias en proyectos Java. Su correcto uso garantiza que los proyectos sean portables, optimizados y funcionales en distintos entornos de desarrollo.

▼ Sintaxis

Claro, acá tenés un resumen del **Apunte de Clase 4 - Sintaxis (Java)** con los conceptos principales y algunos detalles clave:

📌 Resumen - Apunte de Clase 4: Sintaxis en Java

📦 Fundamentos del Lenguaje Java:

- **Basado en objetos**
- **Case sensitive** (distingue entre mayúsculas y minúsculas)
- **Usa llaves** `{}` para bloques de código
- **Finaliza sentencias con** `;`

🔢 Tipos de datos, Variables y Asignaciones

✅ Java es un lenguaje tipado:

- Toda variable debe tener un tipo **especificado**
- El tipo no puede cambiar luego

🧠 Declaración de variables:

```
int x;
float y, z;
char c;
String nombre;
```

📊 Tipos de datos:

Tipo	Tamaño	Ejemplo literal	Rango (aproximado)
byte	1 byte	`-128` a `127`	[-128, 127]
short	2 bytes		[-32.768, 32.767]
int	4 bytes	`100` , `-4`	[-2^{31}, 2^{31} - 1]
long	8 bytes		Muy amplio
float	4 bytes	`3.5f`	~6-7 decimales
double	8 bytes	`0.123456789`	~14-15 decimales
char	1-2 b.	`'A'`	Depende codificación
boolean	1 byte	`true` , `false`	
String		`"Hola"`	Objeto (Clase)

📝 Asignaciones y Conversión de Tipos

- Si los **tipos son iguales**, se asigna sin problema.
- Si se asigna de un tipo **más chico a uno más grande**, Java realiza **promoción automática**.
- Si se asigna de un tipo **más grande a uno más chico**, requiere **casting explícito**:

```
int a = 123456;
short b = (short) a; // Puede generar pérdida de precisión
```

Ejemplo de pérdida de precisión:

```
int a = 123456;
short b = (short) a; // b termina valiendo -7616
```

🏷️ Nombres de Variables (Identificadores)

🔒 Restricciones:

- Letras, dígitos y guión bajo (`_`)
- No puede **comenzar con un número**
- No puede usar **palabras reservadas**

💡 Convención:

- **camelCase**: empieza en minúscula, palabras nuevas con mayúscula

  Ejemplo: `nombreCliente`, `edadUsuario`

🧠 Inferencia de tipos y literales

- Por defecto:
  - **Enteros** → `int`
  - **Decimales** → `double`
- Si usás `float`, agregá `f` al final:

```
float x = 3.5f;
```

⚡ Casting entre tipos

```
int x = 100;
short y = (short) x; // Ok, si no hay pérdida
```

🆕 Uso de `var` (Java 10+)

- Permite **inferencia automática** del tipo
- Solo para **variables locales**

```
var edad = 25;       // tipo int
var nombre = "Juan";  // tipo String
```

Pros:

✅ Código más limpio y conciso

✅ Menos repetición

Contras:

⚠️ Puede dificultar la lectura si el tipo no es obvio

⚠️ No puede usarse con valores ambiguos (como `null`)

¿Querés que te haga también un resumen visual como una tabla o esquema de este apunte?

Aquí tienes un resumen del contenido:

Operadores Aritméticos en Java

En Java, se pueden asignar valores a variables y también el resultado de expresiones aritméticas, que combinan operadores y operandos.

Principales Operadores Aritméticos

Operador	Significado	Ejemplo
`+`	Suma	`a = b + c;`

`-`	Resta	`a = b - c;`
`*`	Multiplicación	`a = b * c;`
`/`	División	`a = b / c;`
`%`	Módulo (resto)	`a = b % c;`

🔷 **División entera**: Si ambos operandos son `int`, el resultado es un entero (sin decimales).

🔷 **Inferencia de tipo**: El tipo de resultado sigue el orden: `int < long < float < double`.

Entrada y Salida en Java

Mostrar datos en pantalla

- `System.out.print()` muestra texto en la misma línea.
- `System.out.println()` agrega un salto de línea al final.
- Se pueden combinar texto y variables:

```
System.out.println("El resultado es: " + b);
```

Leer datos desde teclado con Scanner

La clase `Scanner` permite capturar datos del usuario:

```
import java.util.Scanner;
public class App {
  public static void main(String[] args) {
    Scanner miEscaner = new Scanner(System.in);
    System.out.print("Ingrese el valor de a: ");
    int a = miEscaner.nextInt();
    System.out.print("Ingrese el valor de b: ");
    int b = miEscaner.nextInt();
    System.out.println("La suma es: " + (a + b));
```

```
  }
}
```

📌 Métodos principales de `Scanner`

Método	Retorno	Descripción
`nextInt()`	`int`	Captura un número entero.
`nextDouble()`	`double`	Captura un número decimal.
`nextLine()`	`String`	Captura una línea de texto.

Estructuras Condicionales

Permiten tomar decisiones según una condición lógica.

Estructura `if-else`

```
if (condición) {
  // Código si la condición es verdadera
} else {
  // Código si la condición es falsa
}
```

Ejemplo:

```
if (edad >= 18) {
  System.out.println("Eres mayor de edad.");
} else {
  System.out.println("Eres menor de edad.");
}
```

Expresiones Lógicas y Operadores Relacionales

Los operadores relacionales comparan valores y devuelven `true` o `false`.

Operador	Significado	Ejemplo

`==`	Igualdad	`a == b`
`!=`	Distinto	`a != b`
`>`	Mayor que	`a > b`
`<`	Menor que	`a < b`
`>=`	Mayor o igual	`a >= b`
`<=`	Menor o igual	`a <= b`

Operadores Lógicos

Permiten combinar múltiples condiciones lógicas.

Operador	Significado	Ejemplo
`&&`	AND (y)	`(a > 5 && b < 10)`
`		`
`!`	NOT (negación)	`!(edad >= 18)`

📌 Reglas importantes

- En Java, las condiciones deben devolver un valor `boolean` (`true` o `false`).
- No se pueden usar valores numéricos como `0` o `1` en lugar de `true` o `false`.

Este resumen cubre los conceptos clave sobre operadores aritméticos, entrada/salida, estructuras condicionales y operadores lógicos en Java. ¿Necesitas que amplíe o aclare algún punto? 😊

Aquí tienes un resumen del contenido:

Variantes de la expresión condicional en Java

Condicional Simple

Permite ejecutar instrucciones solo si la condición es verdadera. Si hay una sola instrucción, las llaves `{}` pueden omitirse.

```
if (condición)
    instrucción;
```

Condicional Doble

Incluye una rama `else` para ejecutar código en caso de que la condición sea falsa.

```
if (condición) {
    // Rama verdadera
} else {
    // Rama falsa
}
```

Condicional Múltiple (`switch`)

Utilizado para evaluar múltiples valores posibles de una expresión. Funciona en cascada si se omiten los `break`.

```
switch (expresión) {
    case valor1:
        instrucciones;
        break;
    case valor2:
        instrucciones;
        break;
    default:
        instrucciones;
}
```

Operador Ternario (`? :`)

Es una alternativa compacta a `if-else` cuando se asignan valores a una variable.

```
String mensaje = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
```

Ventajas: Código más conciso y legible.

Desventajas: No recomendable para lógica compleja.

Operadores Resumidos en Java

Se usan para simplificar operaciones sobre una variable:

```
a += 1;  // Equivale a a = a + 1
b -= 1;  // Equivale a b = b - 1
c *= 3;  // Equivale a c = c * 3
d /= 4;  // Equivale a d = d / 4
```

Operadores Unarios (Incremento/Decremento Pre/Post)

```
a++;  // Post-incremento (usa el valor y luego incrementa)
++a;  // Pre-incremento (incrementa y luego usa el valor)
b--;  // Post-decremento
--b;  // Pre-decremento
```

Ejemplo de diferencia:

```
int a = 5;
int c = ++a;  // a = 6, c = 6
int d = a++;  // a = 7, d = 6
```

Estructuras Repetitivas en Java

Ciclo `for` (0-N repeticiones conocidas)

```
for (int i = 1; i <= 10; i++) {
    // Instrucciones
}
```

Ciclo `while` (0-N repeticiones, condición evaluada antes de entrar al ciclo)

```
while (condición) {
    // Instrucciones
}
```

Ciclo `do-while` (1-N repeticiones, garantiza al menos una ejecución)

```
do {
    // Instrucciones
} while (condición);
```

Este resumen cubre los conceptos clave de las estructuras condicionales y repetitivas en Java.

semana 3

▼ 3era Clase POO

▼ Introducción a POO

📌 Resumen: Introducción a la Programación Orientada a Objetos (POO)

- **Objetivo del curso**: No se busca enseñar Java ni POO en profundidad, ya que se asume que esos temas fueron abordados en materias previas o paralelas. El enfoque está en revisar lo necesario para implementar el backend de aplicaciones, con algunas buenas prácticas.
- **Justificación de la POO**: Se destaca su utilidad para **manejar la complejidad** en problemas que parecen similares, pero no lo son.
- **Ejemplo comparativo**:
    - Calcular el promedio de 10 números enteros → **problema trivial**.

    - Calcular el promedio de 10 fracciones → **problema más complejo**, requiere:
        - Operaciones más elaboradas (suma, división).
        - Simplificación de fracciones para evitar números enormes.
- **Solución propuesta**: Crear una **clase** `Fracción` que encapsule operaciones como suma y simplificación, permitiendo tratar fracciones como objetos y reduciendo la complejidad del código.
- **Conclusión**: Aunque el objetivo es implementar backend, este ejemplo ilustra cómo la POO ayuda a **abstraer y simplificar problemas complejos**, haciendo el código más manejable y reutilizable.

▼ Primeros pasos

Claro, acá te dejo el **resumen del apartado "Primeros pasos"**:

🧩 Resumen: Primeros pasos en POO

- En Java, los **objetos se describen con clases**, que actúan como moldes para crear múltiples instancias.
- Una clase define:
    - **Atributos**: lo que el objeto *es* o *tiene*.
    - **Métodos**: lo que el objeto *hace*.
- La **clase** `main()` lanza la ejecución de la app. En este curso, se usa una clase `App` por convención, pero no es obligatorio.
- Se puede tener múltiples clases, incluso cada una con su propio `main()`.
- **Ejemplo**: la clase `Fracción` tendrá atributos como numerador y denominador, y métodos para operar con fracciones.
- Se introduce el **Principio de Ocultamiento (Encapsulamiento)**:
    - Los atributos no deben ser accesibles directamente.
    - Se usan métodos *getters* y *setters* para acceder o modificar atributos.

- Esto **protege los datos internos** y **evita errores**, como dividir por cero.

¿Querés que siga con los siguientes apartados también?

▼ Modificadores de Acceso

Perfecto, acá tenés el **resumen completo actualizado**, combinando todo lo anterior con lo nuevo que agregaste:

🔐 Resumen: Modificadores de Acceso y Constructores

✅ Modificadores de acceso en Java

Los modificadores controlan la visibilidad de los atributos y métodos desde otras clases:

Modificador	Acceso permitido desde...
`public`	Desde cualquier clase.
`private`	Solo desde dentro de la misma clase.
`protected`	Desde subclases y clases del mismo paquete.
(default)	Solo desde clases del mismo paquete. *(se aplica si no se especifica ningún modificador)*

→ Por eso, **los atributos de la clase `Fracción` se declaran como `private`**, siguiendo el **Principio de Ocultamiento**, que protege los datos internos de accesos directos desde fuera.

🛠️ Constructores y sobrecarga

- Un **constructor** inicializa un objeto cuando se lo crea con `new`.
- Tiene el **mismo nombre que la clase**, **no tiene tipo de retorno**, y puede tener **parámetros**.
- **Sobrecarga de constructores**: permite tener varios constructores con distinta cantidad o tipo de parámetros.

Ejemplos:

1. **Constructor con dos parámetros (`int`)**

   Inicializa numerador y denominador.

   → Se realiza una verificación del denominador (revisada en el siguiente apartado).

2. **Constructor con un solo parámetro (`int`)**

   Interpreta el entero como una fracción impropia (`entero/1`).

   → Usa la palabra reservada `this` para llamar a otro constructor (esto debe hacerse siempre en la **primera línea** del constructor).

3. **Constructor copia**

   Recibe otro objeto `Fracción` como parámetro.

   → Crea una nueva fracción con los mismos valores que otra.

   → Esto **no es lo mismo** que asignar referencias: copia los datos, no la referencia. La clonación real se verá más adelante.

➕ Próximos pasos en la clase `Fracción`

Hasta este punto, la clase puede **guardar datos**, pero no tiene comportamientos reales. Se agregarán dos métodos fundamentales:

1. **Un método que devuelve el valor real de la fracción** (como `double`).
2. **Un método que devuelve una representación textual** de la fracción (`toString()`), útil para mostrarla como cadena.

¿Querés que también resuma esos dos métodos cuando los veamos en el próximo apartado?

▼ Referencias y Creación de Objetos

Claro, acá tenés el **resumen** del apartado **"Referencias y Creación de Objetos"**:

📦 Resumen: Referencias y Creación de Objetos

Una clase puede usarse para **crear objetos** usando el operador `new`, que:

- **Crea una nueva instancia** de la clase especificada.
- **Invoca automáticamente un constructor** según los **parámetros proporcionados**.

🧪 Ejemplo:

```
Fraccion f1 = new Fraccion(2, 3);
```

Este código:

- Usa `new` para crear un objeto de la clase `Fraccion`.
- Llama al **constructor con dos enteros**.
- Guarda la referencia al objeto en la variable `f1`.
- El objeto `f1` tendrá `numerador = 2` y `denominador = 3`.

📡 Invocación de métodos

Una vez creado un objeto, se pueden **invocar sus métodos** usando la sintaxis:

```
nombreObjeto.nombreMetodo();
```

Por ejemplo:

```
f1.valorReal();
```

→ Se dice que el **objeto "recibe un mensaje"** para ejecutar ese método.

→ En este caso, `f1` invoca el método `valorReal()` para obtener el valor real de la fracción.

¿Querés que siga con el resumen de los próximos apartados cuando los tengas?

▼ Métodos de acceso de lectura y escritura

Claro, acá tenés el **resumen** del último apartado que pasaste, titulado:

🔒 Resumen: Métodos de Acceso, toString() y Complejidad por Capas

🔐 Privacidad y Acceso a Atributos

- En Java, los **atributos privados** no pueden accederse desde fuera de la clase directamente.
- Para hacerlo de forma controlada, se usan los **métodos de acceso**:
  - `get`: para consultar valores.
  - `set`: para modificarlos (si es conveniente).
- Ejemplo: para un atributo `clave` existen `getClave()` y `setClave()`.

> ⚠️ En la clase Fraccion, no se permiten modificaciones independientes del numerador o denominador ya que, conceptualmente, eso implicaría una nueva fracción (nuevo objeto).

📦 Principio de Ocultamiento

- Se recomienda declarar **atributos como** `private` y **métodos como** `public`.
- Algunos equipos definen todos los `set` aunque no sean necesarios, por **coherencia de diseño**.

🧬 Herencia desde `Object` y `toString()`

- Todas las clases en Java **heredan de la clase** `Object`, que provee métodos básicos.
- El método `toString()` sirve para **representar objetos como cadenas de texto**.
  - Si no se redefine, muestra el nombre de la clase y la dirección de memoria.

- Al redefinirlo, se puede personalizar la salida textual del objeto.
- Se invoca automáticamente en contextos como `System.out.println(objeto)`.

➕ Comportamientos adicionales de la clase `Fraccion`

- Se agregan métodos para que la fracción pueda:
  - **Sumarse, restarse, multiplicarse y dividirse** con otra fracción.
  - **Simplificarse**.
- Esto permite que la clase esté lista para usarse en **operaciones más complejas**, como por ejemplo calcular el **promedio de fracciones**.

🧰 Objetos en base a Objetos

- Además del ejemplo de `Fraccion`, se incluye uno donde **objetos se crean a partir de otros más simples**.
- Este enfoque refleja el diseño **orientado a objetos por capas**, permitiendo **manejar la complejidad de forma modular**.

¿Querés que junte todos los resúmenes en un solo documento más adelante o los vas copiando aparte?

▼ Arreglos en Java

Resumen sobre Arreglos en Java:

- **Arreglos unidimensionales (vectores):**

  Son estructuras que almacenan múltiples elementos del mismo tipo, accedidos por **índices** que comienzan en **0**. Por ejemplo, `v[2]` accede al tercer elemento del arreglo `v`.

- **Declaración y creación:**

  Primero se declara la referencia al arreglo:

  ```
  int[] v;
  ```

Luego se crea con `new`:

```
v = new int[6]; // arreglo de 6 enteros, inicializados en 0
```

- **Atributo `length`:**

  Permite saber el tamaño del arreglo:

  ```
  v.length
  ```

- **Procesamiento típico:**

  Se usa un ciclo `for` para recorrer los elementos:

  ```
  for (int i = 0; i < v.length; i++) {
      System.out.println(v[i]);
  }
  ```

Matrices (arreglos bidimensionales):

- Se estructuran como una **tabla de filas y columnas**, y requieren **dos índices**: uno para la fila, otro para la columna.

  ```
  int[][] a;
  a = new int[6][4]; // 6 filas y 4 columnas
  a[2][3] = 5; // accede a fila 2, columna 3
  ```

- **Recorridos:**

  Se usan **dos ciclos `for` anidados**:

  - **Fila creciente:**

    ```
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a[i].length; j++) {
            // procesar a[i][j]
    ```

```
    }
}
```

- **Fila decreciente:**

```
for (int i = a.length - 1; i >= 0; i--) {
    for (int j = 0; j < a[i].length; j++) {
        // procesar a[i][j]
    }
}
```

- **Columna creciente o decreciente:** invertir el orden de los ciclos:

```
for (int j = 0; j < a[0].length; j++) {
    for (int i = 0; i < a.length; i++) {
        // procesar a[i][j]
    }
}
```

- **Importante:**

  El **primer índice** siempre indica la **fila**, y el **segundo la columna**, sin importar el orden de los bucles.

¿Querés que arme ejemplos de código completos con esto?

▼ Manejo de Excepciones en Java

Resumen: Manejo de Excepciones en Java

En Java, los errores en tiempo de ejecución son representados como **objetos de excepción**, que derivan de la clase base `Throwable`. Estos errores pueden ser tratados por el programador para evitar que el programa termine abruptamente.

Jerarquía de Excepciones

- `Throwable`
  - `Error`: Errores graves que la JVM maneja automáticamente.

- `Exception`: Errores comunes de programación.
  - **Checked (Chequeadas)**: El compilador obliga a manejarlas (ej: `IOException`).
  - **Unchecked (No chequeadas)**: No es obligatorio manejarlas (ej: `RuntimeException` y sus derivadas).

Excepciones comunes

- **De `RuntimeException`**: `NullPointerException`, `ArithmeticException`, `IndexOutOfBoundsException`.
- **De `IOException`**: `EOFException`, `FileNotFoundException`, `InterruptedIOException`.

Manejo de Excepciones

Bloque `try-catch-finally`

- `try`: Código que puede lanzar una excepción.
- `catch`: Captura y maneja la excepción.
- `finally`: Se ejecuta siempre, haya o no excepción (útil para cerrar archivos, liberar recursos, etc.).

```
try {
  // Código que puede lanzar excepción
} catch (IOException e) {
  // Manejo del error
} finally {
  // Siempre se ejecuta
}
```

- Pueden usarse múltiples `catch`, de más específicos a más generales.
- Se debe evitar capturar todo con `catch(Exception e)` de forma genérica.

`try-with-resources`

Desde Java 7, permite declarar recursos (como archivos) dentro del `try`, los cuales se cierran automáticamente al finalizar:

```
try (RandomAccessFile raf = new RandomAccessFile("archivo.txt",
"r")) {
  // Operaciones con raf
} catch (IOException e) {
  // Manejo del error
}
```

Este enfoque evita tener que usar `finally` solo para cerrar recursos.

`throws`: Declarar posibles excepciones

Si un método puede lanzar una excepción *checked* y no la maneja, debe declararla con `throws`:

```
void leer() throws IOException {
  // Código que puede lanzar IOException
}
```

- Si puede lanzar varias excepciones, se listan separadas por comas.
- No se necesita declarar `RuntimeException` ni `Error`.

Conclusión: Java provee un sistema robusto para el manejo de errores en tiempo de ejecución mediante excepciones. Es responsabilidad del programador decidir cómo manejar estos errores, ya sea capturándolos o propagándolos, para asegurar la estabilidad de la aplicación.

semana 4

▼ 4ta clase colecciones

3K5 - Iteradores y Colecciones.pdf

▼ Colecciones

● Las colecciones en Java son estructuras de datos que permiten almacenar grupos de objetos.

● Existen varios tipos de colecciones: List, Set, Queue y Map.

● Una lista es una colección ordenada que puede contener elementos duplicados, mientras que un Set no permite duplicados.

```
List<String> miLista = new ArrayList<>();
miLista.add("Elemento 1");
miLista.add("Elemento 2");
```

▼ lista vs array

● Las listas y arrays se diferencian en que las listas pueden crecer dinámicamente mientras que los arrays son de tamaño fijo

● Los arrays en Java no permiten redimensionar automáticamente su tamaño. Cuando el array está lleno, es necesario crear un nuevo array de mayor tamaño.

● Una lista sobre un array en Java simula el comportamiento dinámico de una lista
usando un array.

● Arrays:
○ Tamaño fijo: Una vez definido el tamaño del array, no puede cambiar.
○ Tipo homogéneo: Todos los elementos deben ser del mismo tipo.
○ Acceso rápido por índice.

● Colecciones (Listas, Set, Map, etc.):

○ Tamaño dinámico: Las colecciones como ArrayList pueden crecer y decrecer según sea necesario.
○ Tipo genérico: Puedes almacenar cualquier tipo de objeto utilizando genéricos.
○ Métodos avanzados: Soporte para búsquedas, ordenamientos e iteraciones eficientes.

▼ Ventajas del uso de Colecciones en Java

● Flexibilidad: Las colecciones permiten almacenar y gestionar grandes cantidades de datos de forma eficiente.

● Dinamismo: A diferencia de los arrays, las colecciones como las listas pueden cambiar de tamaño automáticamente a medida que se añaden o eliminan elementos.

● Reutilización: Al utilizar interfaces y clases genéricas, las colecciones se pueden aplicar a distintos tipos de objetos sin cambiar el código base.

● API: La API de colecciones de Java ofrece una amplia variedad de métodos y estructuras (listas, sets, mapas, etc.) para resolver diversos problemas.

▼ JERARQUIA DE COLECCIONES

● La jerarquía de colecciones en Java está basada en dos interfaces principales: Collection y Map.

○ Collection: Raíz de la jerarquía para colecciones como List, Set, y Queue.
Define las operaciones básicas como add(), remove(), y size().

○ Map: No extiende Collection, pero permite asociar claves con valores (Ejemplo: HashMap).

Ejemplo de jerarquía:

- **List** -> `ArrayList`, `LinkedList`
- **Set** -> `HashSet`, `TreeSet`
- **Map** -> `HashMap`, `TreeMap`

▼ METODOS COMUNES

- `add(E e)`: Añade un elemento a la colección.
- `remove(Object o)`: Elimina un elemento de la colección.
- `size()`: Retorna el número de elementos en la colección.
- `isEmpty()`: Verifica si la colección está vacía.
- `contains(Object o)`: Verifica si un objeto está presente en la colección.
- `iterator()`: Devuelve un iterador para recorrer la colección.

```java
List<String> lista = new ArrayList<>();
lista.add("Elemento");
System.out.println(lista.size());
```

▼ DIFF entre set y list

List:

- Permite elementos duplicados.
- Mantiene el orden de inserción.
- Ejemplos: `ArrayList`, `LinkedList`.

Set:

- No permite duplicados.
- No garantiza el orden de los elementos (excepto `TreeSet`).
- Ejemplos: `HashSet`, `TreeSet`.

```java
Set<String> set = new HashSet<>();
set.add("A");
set.add("A");  // No se permite duplicado.
System.out.println(set.size());  // Resultado: 1
```

▼ Ejemplo práctico con ArrayList y HashMap

● ArrayList permite almacenar elementos de forma ordenada y con acceso por índice.
● HashMap es una estructura clave-valor donde cada clave está asociada a un valor específico.

```java
List<String> lista = new ArrayList<>();
lista.add("Elemento 1");
lista.add("Elemento 2");

Map<String, Integer> map = new HashMap<>();
map.put("Clave1", 10);
map.put("Clave2", 20);
System.out.println(map.get("Clave1"));  // Salida: 10
```

▼ Patron Iterador

● El patrón Iterador es un patrón de diseño que permite recorrer los elementos de una colección sin exponer su estructura interna.

● Un iterador en Java implementa la interfaz Iterator, que define los métodos hasNext(), next() y remove().

● Este patrón facilita el acceso secuencial a los elementos de una colección,
independientemente de cómo esté implementada la colección

▼ Clases internas

● Las clases internas en Java permiten acceder a los métodos y atributos de la clase contenedora.

● Son útiles para encapsular y mejorar la organización cuando una clase sólo tiene sentido en el contexto de otra.

● Una clase interna puede ser usada para manejar iteraciones o comportamientos específicos, como en el caso del patrón Iterador.

▼ Clases anonimas

● Las clases anónimas son clases internas que no tienen un nombre específico y se definen en el momento de ser utilizadas.

● Son útiles cuando solo necesitas una instancia de una clase para sobrescribir métodos o realizar pequeñas tareas

▼ GENERICS JAVA

● Generics en Java permiten que las clases y métodos operen con tipos de datos
especificados por el usuario, asegurando la seguridad de tipos y eliminando la
necesidad de casting.

💡 de parcial:

lectura de csv. tenian datos de alumnos y necesitaba sacar la edad promedio por curso
es util para agrupar valores sacar un valor

JPA ORM JAVA

hacer ej de la semana 4

💡 para construir un listado de clientes para una determinada ciudad, aca usamos un map para hacer que no se repitan los clientes

💡 para lo ordenes podemos usar una lista porque no deberian repetirse en el csv

▼ Herencia en Java

Claro, acá tenés un **resumen** en un solo párrafo que sintetiza todo el texto que compartiste:

En Java, la herencia es un mecanismo de la programación orientada a objetos que permite crear nuevas clases (subclases) basadas en clases ya existentes (superclases), heredando sus atributos y métodos sin necesidad de redefinirlos. Este mecanismo promueve la reutilización de código y la organización jerárquica de clases. Java solo permite **herencia simple**, es decir, una clase solo puede heredar directamente de una única superclase, a diferencia de lenguajes como C++ que permiten herencia múltiple. Las relaciones de herencia permiten modelar comportamientos compartidos y se representan en UML con una flecha que va desde la subclase hacia su superclase. Un buen criterio para decidir si debe haber herencia entre dos clases es preguntarse si una clase "es un tipo de" la otra. Por ejemplo, un `Cliente` **es una** `Persona`, por lo que puede heredar de ella, pero un `Cliente` **usa una** `Cuenta`, por lo tanto no debería heredar de `Cuenta` sino tenerla como atributo. Java resuelve la falta de herencia múltiple con el uso de interfaces para representar comportamientos compartidos entre distintas clases.

¿Querés que también lo pasemos a formato de diapositiva, o con eso te alcanza?

▼ Forma de declaración y uso de constructores en una jerarquía

Aquí tenés un **resumen claro y ordenado** sobre la *declaración y uso de constructores en una jerarquía en Java*:

🧬 Herencia en Java

- Se usa la palabra clave `extends` para declarar que una clase hereda de otra.
- Java **no permite herencia múltiple**, solo puede haber una superclase directa.

```
public class Inversion extends Cuenta { ... }
```

🏗️ Relación con la clase Object

- Si una clase no indica de quién hereda, **hereda implícitamente de** `Object`.
- Esto explica por qué métodos como `toString()` están presentes en cualquier clase: se heredan desde `Object`.

🧳 Constructores en jerarquías

- Si **no se define un constructor**, Java genera automáticamente un **constructor por defecto** (sin parámetros).
- Si **se define al menos un constructor con parámetros**, Java **no genera** el constructor por defecto automáticamente.

🔁 Constructores y palabra clave `super`

- Un constructor de clase derivada debe **invocar a un constructor de la superclase**, usando `super(...)`.
- Si no se incluye explícitamente, Java inserta una llamada a `super()` (constructor sin parámetros) por defecto.

```
public Corriente(int num, float sal, boolean des) {
    super(num, sal); // llama al constructor de Cuenta
    descubierto = des;
}
```

🧬 Herencia de métodos

- Una clase derivada **puede redefinir** (sobrescribir) métodos heredados para adaptarlos.

- También puede **usar directamente** métodos heredados si no necesita modificarlos.

🛠️ Redefinición vs. Sobrecarga

- **Redefinir**: volver a declarar un método heredado con la misma firma para modificar su comportamiento.
- **Sobrecargar**: crear múltiples versiones de un método con el mismo nombre pero diferentes parámetros.

🔒 Acceso a atributos heredados

- Si un atributo en la superclase es:
  - `private`: **no accesible directamente** por la subclase (se accede mediante getters/setters).
  - `protected`: **accesible directamente** por la subclase, pero no por clases externas.
  - `public`: accesible desde cualquier clase.

📦 Reglas de uso de métodos en la jerarquía

- Un objeto puede invocar cualquier método definido **en su clase o en sus superclases**.
- **No puede** invocar métodos definidos exclusivamente en subclases (más abajo en la jerarquía).

¿Querés que te prepare también un cuadro comparativo o un diagrama UML para visualizar esto?

▼ Polimorfismo en Java

Resumen: Polimorfismo en Java

- **Definición:** El polimorfismo permite que una **referencia a una clase base** (por ejemplo, `Cuenta`) pueda apuntar a objetos de cualquier clase derivada de ella (`Inversion`, `Corriente`, etc.).

Ejemplo:

```
Cuenta x = new Inversion();
```

- **Referencias polimórficas:** Se refiere a variables como `Cuenta x;` que pueden contener objetos de diferentes clases de una misma jerarquía. La inversa no es válida: una variable `Inversion y;` no puede apuntar a un objeto `Cuenta`.
- **Invocación de métodos:**
  - Si el método está definido en la **clase base** (`Cuenta`), Java lo resuelve correctamente en tiempo de ejecución según el tipo real del objeto.
  - Si el método **no está en la clase base**, no se puede invocar desde una referencia polimórfica. Es necesario un **casting explícito**:

    ```
    Cuenta b = new Inversion(...);
    ((Inversion) b).actualizar();  // solo funciona tras el casteo
    ```

- **Ejemplo de uso de polimorfismo:**

  ```
  Cuenta a = new Cuenta(1, 1000);
  Cuenta b = new Inversion(2, 2000, 2.31f);
  Cuenta c = new Corriente(3, 1500, true);

  System.out.println(a.toString()); // usa toString de Cuenta
  System.out.println(b.toString()); // usa toString de Inversion
  System.out.println(c.toString()); // usa toString de Corriente
  ```

- **Detección de tipo real del objeto:**
  - Con `instanceof`:

    ```
    if (a instanceof Corriente) { ... }
    ```

  - Con `getClass()`:

```
if (a.getClass() == b.getClass()) { ... }
```

Conclusión: El polimorfismo permite manejar objetos de diferentes clases usando una misma interfaz de referencia (la clase base), siempre que se respeten ciertas reglas. Para usar métodos específicos de las subclases, es necesario hacer **casting**. Además, se pueden usar herramientas como `instanceof` y `getClass()` para identificar el tipo real del objeto.

▼ Arreglos de objetos de clases diferentes

Resumen: Arreglos de objetos de clases diferentes (Polimorfismo en Java)

- **Máximo polimorfismo con `Object`:**

  En Java, todas las clases derivan de `Object`, por lo tanto:

  ```
  Object x = new Inversion();
  Object y = "casa"; // equivalente a new String("casa")
  ```

  Las llamadas a métodos como `toString()` se resuelven correctamente en tiempo de ejecución según el tipo real del objeto.

- **Arreglos polimórficos:**

  Se pueden crear arreglos de una clase base (por ejemplo, `Cuenta`), y llenarlos con objetos de sus subclases (`Inversion`, `Corriente`, etc.):

  ```
  Cuenta[] v = new Cuenta[4];
  v[0] = new Inversion(...);
  v[1] = new Corriente(...);
  v[2] = new Cuenta(...);
  v[3] = new Inversion(...);
  ```

- **Procesamiento genérico:**

Al recorrer el arreglo, se pueden invocar métodos definidos en la clase base (`Cuenta`), y la JVM se encargará de llamar a la versión correspondiente según el objeto:

```
for (int i = 0; i < 4; i++) {
   v[i].retirar(1000); // funciona para todos los objetos
}
```

- **Acceso a métodos específicos de subclases:**

  Para acceder a métodos definidos solo en subclases (como `actualizar()` en `Inversion`), se usa `instanceof` y **casting explícito**:

```
for (int i = 0; i < 4; i++) {
   if (v[i] instanceof Inversion) {
      Inversion inv = (Inversion) v[i];
      inv.actualizar();
   }
}
```

- **Filtrar por clase real del objeto:**

  Usando `getClass()`, se puede comparar el tipo real de los objetos:

```
Cuenta este = v[0];
for (int i = 0; i < 4; i++) {
   if (v[i].getClass() == este.getClass()) {
      System.out.println("v[" + i + "]: " + v[i]);
   }
}
```

Conclusión:

El uso de arreglos con referencias polimórficas permite almacenar y procesar objetos de diferentes clases derivadas con facilidad. Sin embargo, para acceder a comportamientos específicos de subclases, es necesario verificar el tipo real con `instanceof` y aplicar casting.

▼ Referencias y tipos primitivos

Resumen: Referencias y tipos primitivos en Java

- **Wrapper classes (Clases de envoltorio):**

  Java ofrece clases especiales que permiten tratar **valores primitivos** (`int`, `float`, `char`, etc.) como **objetos**. Estas clases permiten:

  - Usar tipos primitivos en estructuras que requieren objetos (como colecciones de `java.util`).
  - Participar en jerarquías y polimorfismo.
  - Convertir entre `String` y valores primitivos, y viceversa.

- **Clases envoltorio por tipo primitivo:**

  Algunas de ellas son:

  - `Integer` para `int`
  - `Float` para `float`
  - `Character` para `char`

- **Métodos de conversión:**

  Cada wrapper incluye métodos como:

  ```
  intValue(), floatValue(), charValue(), etc.
  ```

  Ejemplo clásico (previo a Java 5):

  ```
  Integer i1 = new Integer(23);
  int i2 = i1.intValue();
  ```

- **Auto-boxing y auto-unboxing (desde Java 5):**

  - **Auto-boxing:** Asignar un valor primitivo directamente a una variable de tipo wrapper.
  - **Auto-unboxing:** Extraer automáticamente el valor primitivo de un wrapper.

Ejemplo moderno:

```
Integer i1 = 23;      // auto-boxing
int i2 = i1;          // auto-unboxing

Float f1 = 2.34f;
float f2 = f1;

Character c1 = '$';
char c2 = c1;
```

Ya no es necesario usar `new` ni los métodos `.intValue()` o similares.

- **Ventaja del auto-boxing:**

  Hace más sencillo y limpio el código. El compilador se encarga de hacer las conversiones necesarias de forma automática, similar a cómo gestiona las cadenas con `String`.

Conclusión:

Gracias a las **wrapper classes** y al **auto-boxing/unboxing**, Java permite trabajar cómodamente con tipos primitivos como si fueran objetos, lo que resulta esencial para usar estructuras genéricas y aprovechar el polimorfismo.

▼ Modificadores static, final y abstract

Aquí tenés un resumen claro y conciso sobre los modificadores `static`, `final` y `abstract` en Java, enfocándonos en lo desarrollado en el texto:

🔷 Modificador `static`

- Se aplica a **atributos** y **métodos** para que pertenezcan **a la clase** en lugar de a una instancia específica.
- **Atributos** `static`:
  - Solo hay una **copia compartida** para todas las instancias.
  - Se inicializan al cargarse la clase, antes de crear instancias.

    - Útiles, por ejemplo, para llevar un contador de instancias creadas.
- **Métodos** `static`:
    - Se pueden invocar **sin crear un objeto**, usando el nombre de la clase.
    - **No pueden acceder** a atributos ni métodos que **no sean también** `static`.
    - Ejemplo: `Persona.getInstanceCounter()` retorna el número de objetos creados.

🔷 Miembros de clase vs. miembros de instancia

- **Miembros de instancia**: cada objeto tiene su **propia copia** del atributo o método.
- **Miembros de clase (`static`)**: hay **una única copia** compartida por todos los objetos.

🔷 Modificador `abstract`

- Se aplica a **clases** y **métodos** que están **incompletos** o **no tienen implementación concreta**.
- **Clases abstractas**:
    - **No se pueden instanciar** directamente.
    - Se usan como base para clases derivadas.
    - Ejemplo: `Cuenta` es abstracta, pero `Inversion` puede heredarla e instanciarse.
- **Métodos abstractos**:
    - **No tienen cuerpo**, solo la firma.
    - Obligan a las clases hijas **a implementar** ese método.
    - Si una clase tiene al menos un método abstracto, **debe ser abstracta**.

🟡 Ejemplos clave:

```
// Atributo y método static
public class Persona {
    private String nombre;
    private int edad;
    private static int contador = 0;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        contador++;
    }

    public static int getInstanceCounter() {
        return contador;
    }
}

// Clase y método abstracto
public abstract class Cuenta {
    private float saldo;

    public abstract void retirar(float imp);
}
```

¿Querés que también te resuma `final` o lo estás dejando para otro momento?

Claro, acá tenés el **resumen** del texto sobre el modificador `final` en Java:

Resumen: Modificador `final` en Java

- El modificador `final` es lo opuesto a `abstract`:
    - **Clase** `final`: no puede ser heredada.

    - **Método** `final`: no puede ser sobreescrito.
- También puede aplicarse a **atributos**, **parámetros** y **variables locales**:
    - Un **atributo** `final` se comporta como una **constante**: solo se puede asignar una vez.
- Si se combina `final` con `static`:
    - Se crea una **constante global** compartida por todas las instancias.
    - Suele declararse como `public` para facilitar su acceso.
- **Reglas según el tipo de combinación**:
    1. `static` solo: puede asignarse en cualquier lugar (declaración, constructor, método) y su valor puede cambiar.
    2. `final` solo: puede asignarse una vez, en la declaración **o** en el constructor, no en ambos.
    3. `static final`: debe asignarse **al declararse**, no en el constructor ni en métodos.
- Si `final` se aplica a un **parámetro** o **variable local**, su valor no puede cambiar una vez asignado.

¿Querés un ejemplo en código para cada caso?

▼ Polimorfismo Aplicado - Interfaces o clases de Interfaz en Java

Claro, acá tenés un **resumen claro y conciso** del texto sobre **polimorfismo aplicado con interfaces en Java**:

Resumen: Polimorfismo Aplicado - Interfaces en Java

- **Polimorfismo** permite manejar objetos de distintas clases usando una referencia común, por ejemplo `Object`, lo cual permite máxima generalidad.
- Un **arreglo de tipo** `Object[]` puede almacenar objetos de cualquier clase, pero solo se pueden usar métodos definidos en `Object`, como `toString()`.

Para otros métodos, se necesita *downcasting*.

Problema al comparar objetos

- Si se quiere comparar objetos (por ejemplo para ordenarlos), deberían tener un método como `compareTo(Object o)`.
- Este método **no existe en** `Object` porque no todas las clases pueden o deben ser comparadas (como los números complejos).
- No se puede modificar `Object` ni usar herencia múltiple para forzar clases a tener ese método.

Solución: Interfaces

- Las **interfaces** permiten definir un conjunto de métodos (sin implementación) que las clases pueden implementar.
- La interfaz `Comparable` ya existe en Java (en `java.lang`) y declara el método:

```
int compareTo(Object x);
```

- Las clases que implementan `Comparable` deben definir este método.

Ventajas de las interfaces

- Se pueden implementar múltiples interfaces (a diferencia de la herencia única).
- Permiten polimorfismo entre clases que no comparten herencia directa, pero implementan la misma interfaz.
- Se pueden usar referencias del tipo de la interfaz, por ejemplo:

```
Comparable c;
c = new Cliente();  // válido si Cliente implementa Comparable
```

Aplicación práctica

- Si se quiere un arreglo con capacidad de comparar sus elementos:

```
Comparable[] v = new Comparable[10];
```

- Cada objeto en el arreglo debe implementar `Comparable`, permitiendo así comparar elementos sin necesidad de *casting*.

¿Querés que te haga también un ejemplo práctico de código que ilustre esto?

▼ Clases String y StringBuilder

Claro, acá tenés el **resumen** del contenido sobre `String` y `StringBuilder` en **Java**:

Resumen: Clases `String`, `StringBuilder` y `StringBuffer`

Clase `String`

- Permite manejar cadenas de caracteres en Java.
- No es un tipo primitivo, sino un objeto más complejo (colección de `char`s con propiedades).
- **Inmutable**: una vez creada, su contenido no puede modificarse.
  - Se pueden cambiar las **referencias**, pero no los caracteres internos.
- Comparación de cadenas:
  - `compareTo()`: compara lexicográficamente (como en un diccionario).
  - `equals()`: compara si dos cadenas son exactamente iguales.
- **Concatenar `String`s reiteradamente es ineficiente**, ya que se crean muchos objetos nuevos.

Clase `StringBuilder`

- Alternativa a `String` para modificar cadenas de forma **eficiente**.
- Permite **agregar, modificar y concatenar** sin crear nuevos objetos.
- Métodos importantes:

- `append()`: concatena cadenas.
- `toString()`: convierte el contenido a un objeto `String`.

Clase `StringBuffer`

- Similar a `StringBuilder`, pero sus métodos están **sincronizados**.
- Ideal para entornos con múltiples hilos (**multithreading**).

¿Querés un mini ejemplo con código para mostrar la diferencia entre `String` y `StringBuilder`?

▼ Iteradores y Colecciones en Java

Claro, acá tenés un **resumen** del contenido del **Apunte 08 - Iteradores y Colecciones en Java**:

Resumen: Iteradores y Colecciones en Java

Objetivo General

- Entender cómo funcionan las **colecciones nativas de Java** y cómo aprovecharlas.
- Sin profundizar en la implementación de estructuras de datos no nativas (tema de otra materia).

Estructuras de Datos

- La idea principal es **encapsular** el almacenamiento real de los datos dentro de una clase.
- El usuario puede trabajar con la estructura sin preocuparse por su implementación interna.

Arrays vs Listas

- **Arrays (vectores)**:
  - Tamaño **fijo**.
  - Solo pueden contener **un mismo tipo de dato**.

- **Listas**:
  - Estructura **dinámica**, pueden crecer en tamaño.
  - Mantienen el **orden de inserción**.
  - Representan una **estructura lineal**: cada elemento tiene 0 o 1 sucesor/predecesor.

Implementación en Java

- Aunque el concepto de listas se asocia naturalmente con **listas enlazadas**, en esta asignatura se trabajará:
  - Con el **concepto de lista**, pero implementado usando **arrays**.
  - Para enfocarse en la lógica de las colecciones sin entrar en los detalles complejos de listas vinculadas.

¿Querés que te resuma también el uso de `Iterator` en Java cuando aparezca en este mismo apunte?

▼ Lista sobre vectores o Arrays

Claro, acá tenés un **resumen claro y ordenado** de esta segunda parte del apunte:

Resumen: Lista sobre Vectores (Arrays) en Java

Objetivo

- Implementar una **lista dinámica** usando un **vector (array)** como almacenamiento interno.
- Ocultar las limitaciones de los vectores al usuario (tamaño fijo, tipo único).

Características principales

- Se usa `Object[]` como vector interno para permitir **polimorfismo** (cualquier tipo de dato).
- Se definen tres atributos clave:

1. **Vector de almacenamiento** (`Object[] elementos`).
2. **Capacidad inicial**.
3. **Cantidad de elementos actuales** (`count`).

Constructores

- Uno **sin parámetros** (usa una capacidad por defecto).
- Otro que recibe la **capacidad inicial** para optimizar uso de memoria y minimizar redimensionamientos.

Método `agregar` (equivalente a `append` en Python)

- Verifica que el objeto no sea `null`.
- Si hay espacio: lo agrega en la posición `count`.
- Si no hay espacio: llama a `asegurarCapacidad()` para redimensionar el vector.

Método `asegurarCapacidad()`

- Crea un **nuevo vector más grande**.
- Copia los elementos del vector viejo al nuevo.
- Reemplaza el vector subyacente con el nuevo.
- Deja espacio extra para evitar redimensionar cada vez.

Uso y recorrido de la lista

Clase `PruebaListaArray`

- Contiene código de prueba para verificar los métodos de la lista.
- Por ejemplo, agregar nombres y recorrer la lista.

Sobre recorridos y estructura interna

- Se puede recorrer con `for` usando `get(i)` y `size()`, pero:

- Esto asume conocimiento de la estructura interna (basada en vector).
- No es recomendable si luego se cambia el tipo de almacenamiento.

Necesidad de un Iterador

- Se busca un patrón de recorrido **sin conocer la estructura interna** de la lista.
- Se propone implementar el **patrón iterador**, con 4 funciones básicas:
  1. **Reiniciar** el recorrido desde el inicio.
  2. **Verificar** si hay un siguiente elemento.
  3. **Obtener** el elemento actual.
  4. **Avanzar** al siguiente elemento.
- Esto permite recorrer cualquier tipo de lista (incluso si está implementada sobre una estructura **no lineal**) de forma simple y uniforme.

¿Querés que te resuma también la parte donde se implementa el iterador más formalmente con la interfaz `Iterator` de Java?

▼ Primera versión trivial del iterador

Perfecto, acá va un **resumen claro y breve** de esta parte sobre la primera versión del iterador:

Resumen: Primera versión trivial del iterador

Funcionamiento

- Se implementa el iterador **dentro de la misma clase** `ListaArray`.
- Se usa un nuevo atributo llamado `actual` que guarda el **índice del elemento actual** del recorrido.

Métodos del iterador interno

1. `iniciarIterador()` → pone `actual = 0`, iniciando el recorrido.
2. `getActual()` → devuelve el elemento en la posición `actual`.
3. `hayMas()` → indica si quedan más elementos por recorrer.
4. `siguiente()` → avanza el índice al siguiente elemento.

Limitación

- **Sólo se puede hacer un recorrido a la vez**.
- Si se reinicia el recorrido en medio de otro (por ejemplo, para buscar algo), se **rompe el iterador actual**.
- Esto vuelve la solución **inviable en muchos casos reales**.

Solución propuesta

- Crear un **nuevo objeto** que maneje la iteración.
- Pero para evitar romper el **encapsulamiento** (ya que ese nuevo objeto necesitaría acceso interno a `ListaArray`), se sugiere usar una **clase interna**.

¿Querés que te siga con el resumen de la implementación usando una clase interna y la interfaz `Iterator` de Java?

▼ Clases internas en Java

Claro, acá tenés el **resumen** de esta parte sobre **clases internas e iteradores mejorados** en Java:

Resumen: Clases internas e iterador mejorado

¿Qué es una clase interna?

- Es una clase **definida dentro de otra clase**.
- Puede acceder **directamente** a todos los atributos y métodos de la clase contenedora, incluso si son `private`.
- Esto **no rompe el encapsulamiento**, ya que se considera parte de la clase externa.

Aplicación en `ListaArrayMejorada`

- Se crea una **clase interna llamada** `IteradorLineal` dentro de `ListaArrayMejorada`.
- Esta clase maneja la lógica de iteración.
- Permite tener **múltiples iteradores independientes al mismo tiempo**, superando la limitación de la versión trivial.

Ventajas

- **Acceso directo a los atributos privados** de la lista.
- Se puede crear un nuevo iterador en cualquier momento sin afectar otros recorridos.
- Mayor flexibilidad y respeto por el encapsulamiento.

Uso del iterador

- La clase `ListaArrayMejorada` implementa un método que devuelve una instancia de `IteradorLineal`.
- Ahora, el recorrido se hace a través del objeto iterador:

```
IteradorLineal it = lista.getIterador();
while (it.hayMas()) {
    System.out.println(it.getActual());
    it.siguiente();
}
```

¿Querés que siga con el resumen de cómo se integra esto con la interfaz `Iterator` de Java?

▼ Clases anónimas en java - clases inline

Claro, acá tenés el **resumen sobre clases anónimas (o clases inline) en Java**:

Resumen: Clases Anónimas en Java

¿Qué son?

- Son **clases internas sin nombre**, usadas para crear un **único objeto** con una implementación específica.
- Se utilizan cuando solo se necesita implementar **un método o conjunto muy limitado de métodos**.
- Se crean **en el momento**, se **compilan y cargan en memoria solo una vez**, evitando sobrecarga innecesaria.

¿Cuándo se usan?

- Cuando se quiere **implementar rápidamente una interfaz o extender una clase abstracta**, solo para una tarea puntual.
- Son ideales para **casos simples y temporales**, evitando crear una clase con nombre para una sola instancia.

Sintaxis (Ejemplo):

```
interface Procesador {
    void procesar();
}

Procesador miProcesador = new Procesador() {
    @Override
    public void procesar() {
        System.out.println("Procesamiento en clase anónima.");
    }
};
```

Ventajas:

- **Código más limpio y compacto.**
- Se evita la creación de múltiples archivos de clases innecesarios.

- **Perfectas para lógica rápida**, como callbacks, listeners, etc.

Limitaciones:

- No se pueden reutilizar.
- No pueden tener constructores.
- Si el cuerpo es complejo, puede dificultar la lectura del código.

¿Querés que siga con el resumen de expresiones lambda o ya te quedó claro este bloque?

▼ Clases Parametrizadas en Java - Generics

Claro, acá te va el **resumen sobre Clases Parametrizadas (Generics) en Java**:

Resumen: Clases Parametrizadas (Generics) en Java

Problema Inicial

- En versiones previas a Java 5, se usaban listas tipo `ArrayList` con objetos `Object`, lo que:
  - Permitía insertar objetos de cualquier tipo.
  - Requería **casting** al recuperar elementos.
  - No permitía asegurar homogeneidad de tipos en tiempo de compilación.

Solución: Generics (desde Java 5)

- Permiten **parametrizar clases con tipos**.
- Garantizan que una colección contenga **solo objetos de un tipo específico**.
- El **compilador controla los tipos** y evita errores por inserciones incorrectas.

Ejemplo:

```
ArrayList<String> lista = new ArrayList<>();
lista.add("hola");          // ok
lista.add(5);               // error de compilación
```

Aplicación en Clases Propias

- También podemos crear nuestras clases parametrizadas:

```
public class ListaArray<E> {
    private Object[] datos;

    public void agregar(E elem) { ... }
    public E obtener(int i) {
        return (E) datos[i]; // se requiere casting
    }
}
```

Limitaciones

- No se puede crear directamente un array de tipo `E[]`, por eso se usa `Object[]` internamente.
- Las operaciones sobre los elementos se limitan a los métodos de `Object`, salvo que:

Restricción con `extends`

- Para operaciones más específicas (como comparar), se puede restringir el tipo:

```
public class ListaArray<E extends Comparable> {
    ...
}
```

- Aunque `Comparable` es una interfaz, se usa `extends` como palabra clave para indicar la restricción.

¿Querés que continúe con el resumen de expresiones lambda o pasamos a otra parte?

▼ Framework de Colecciones de Java

Aquí tenés un resumen del contenido:

Resumen: Framework de Colecciones y Paquetes en Java

API de Java

- Es el conjunto de clases que permite a los programadores desarrollar aplicaciones en Java.
- Está organizada en **paquetes (packages)**, que agrupan clases con funcionalidades comunes.
- Java es un lenguaje **orientado a objetos**, y la API ofrece clases utilitarias para múltiples tareas.

Paquetes de clases

- Un **package** agrupa clases con objetivos similares (ej. `java.awt` para interfaces gráficas, `java.io` para entrada/salida).
- No es necesario que las clases de un package tengan relación de herencia.
- Usar packages ayuda a evitar conflictos de nombres y mejora la organización del proyecto.
- Físicamente, un package es un **directorio** que contiene archivos `.class` compilados.
- El nombre completo de un package refleja su ruta de carpetas (ej. `java.awt` = carpeta `java` → subcarpeta `awt`).

Importación y visibilidad

- Para acceder a clases de otro package se usa `import`.
  - Ej: `import java.awt.*;` o `import java.awt.Button;`.
- También se puede usar el **nombre completamente calificado** (ej. `java.awt.Button b1;`).
- El único package que se importa automáticamente es `java.lang` (contiene clases básicas como `String`, `Math`, etc.).

Creación de packages propios

- Se declara con la instrucción `package` al comienzo del archivo fuente.
- Cada clase debe estar dentro de un package (excepto si se ubica en el *default package*, algo no recomendado).
- Los IDEs (como BlueJ, NetBeans, Eclipse) ayudan a gestionar paquetes fácilmente.

Distribución de software: archivos .jar

- Para distribuir aplicaciones o bibliotecas, se utiliza el comando `jar` que crea un **archivo `.jar` (Java Archive)**.
- Los archivos `.jar`:
  - Son como archivos `.zip` que contienen clases compiladas.
  - Pueden ser ejecutables si contienen un `main()`.
  - Se pueden generar desde la línea de comandos o desde el IDE.
  - NetBeans, por ejemplo, genera el `.jar` dentro de la carpeta `dist`.

¿Querés que te lo resuma aún más o en formato de esquema visual?

▼ Api de colecciones

Aquí tenés un **resumen claro y ordenado** del texto sobre la **API de Colecciones en Java**:

📦 Colecciones en Java (`java.util`)

Java ofrece un sistema de clases e interfaces (en el paquete `java.util`) para manejar **estructuras de datos dinámicas** llamadas **Colecciones (Collections)**.

Estas **no almacenan tipos primitivos**, pero pueden hacerlo usando **wrapper classes** como `Integer`, `Double`, etc., con ayuda del **autoboxing**.

🧩 Jerarquías principales

🔷 `Collection` (raíz de varias estructuras)

- Representa un **grupo de elementos** (pueden repetirse o no, ordenados o no).
- No se implementa directamente. Se usan sus subinterfaces:

Subinterfaz	Características
Set	No permite duplicados. Simula conjuntos matemáticos.
List	Ordenada (mantiene el orden de inserción), puede tener duplicados. Permite acceso por índice.
Queue	Manejo FIFO (first-in, first-out), aunque puede variar. Se usa para procesar elementos en orden.
Deque	Puede actuar como FIFO o LIFO. Permite inserción y eliminación por ambos extremos.

🔷 `Map` (otra jerarquía paralela)

- Asocia **claves únicas con valores**. No permite claves duplicadas.
- Implementación común: **HashMap**.

🔷 Versiones ordenadas

Interfaz	Descripción
`SortedSet`	Versión ordenada de `Set`. Orden ascendente.
`SortedMap`	Versión ordenada de `Map`. Ordena por clave.

📄 Listas: una estructura clave

- Son **lineales y adaptativas** (crecen/disminuyen según necesidad).
- A diferencia de arreglos, **permiten inserciones en cualquier posición**.

Tipos de implementación:

1. **Lista ligada**:
    - Hecha de **nodos con enlaces** (sucesor y/o antecesor).
    - Acceso **secuencial**, tiempo de búsqueda **O(n)**.
    - Ejemplo: `LinkedList`.
2. **Lista sobre arreglo**:
    - Usa un **arreglo de soporte contiguo**.
    - Acceso por índice en **tiempo constante O(1)**.
    - Ejemplo: `ArrayList`.

🧰 Clases abstractas para facilitar implementación

`AbstractCollection`, `AbstractList`, `AbstractSequentialList`

Ayudan a implementar nuevas listas con menos código.

- `AbstractList`: Para listas con acceso directo (`ArrayList`).
    - Requiere sobrescribir `get(int)` y `size()`.
    - Para listas modificables: también `set(int, E)`, `add(int, E)` y `remove(int)`.
    - Maneja iteradores internamente. Usa `modCount` para detectar cambios durante iteraciones (fail-fast).
- `AbstractSequentialList`: Para listas ligadas (`LinkedList`).
    - Se debe sobrescribir `listIterator()` y `size()`.
    - Es necesario crear una clase `ListIterator`.

✅ Conclusión

Java provee una poderosa API de colecciones basada en interfaces y clases abstractas. Esta estructura:

- Fomenta el **polimorfismo**.
- Facilita la implementación de nuevas estructuras.
- Distingue entre **acceso secuencial** y **acceso directo**, eligiendo la clase base adecuada.

¿Querés que te resuma también las convenciones para implementar listas propias desde `AbstractSequentialList` o `AbstractList`?

▼ El patrón Iterador en Java

Aquí tenés un **resumen** del texto sobre el **patrón Iterador en Java**:

Java implementa el **patrón Iterador** mediante las interfaces `Iterable<>` e `Iterator<>`.

- `Iterable` obliga a implementar el método `iterator()`, que debe devolver una instancia de un iterador.
- Este enfoque **desacopla** la lógica de iteración del tipo concreto de colección (por ejemplo, `ArrayList`, `LinkedList`, etc.), lo que **flexibiliza el modelo**.

Para ilustrar esta implementación, se presenta la clase `ListaArrayIterable`, que:

- Implementa `Iterable`.
- Incluye una clase interna `ListaArrayIterator` que implementa `Iterator`.

Esta implementación incluye:

- Un constructor que inicializa el iterador.
- El método `hasNext()` para verificar si hay más elementos.
- El método `next()` que:
  - Retorna el elemento actual.
  - Avanza al siguiente.

El método `remove()`:

- Elimina el último elemento retornado por `next()`.
- Ajusta el índice `current` para que apunte al elemento anterior.
- Lanza una excepción si `next()` no fue llamado antes.
- Llama a `ListaArrayIterable.this.remove(current)` para acceder al método de la clase contenedora desde la clase interna.
- Finalmente, desactiva la bandera `next_ok` para evitar que `remove()` sea llamado dos veces seguidas sin una nueva llamada a `next()`.

¿Querés que te resuma también el ejemplo con código o te sirve esto así?

> 💡 otro ej de parcial es crear objetos a partir de un csv y con el jpa guardarlo en la base de datos

semana 5

▼ Programación funcional y API de Streams

▼ Introducción a la programación funcional

La programación funcional es un paradigma de desarrollo de software que se suma a los ya conocidos paradigmas estructurado y orientado a objetos. Aunque tiene una base matemática compleja y existen pocos lenguajes puramente funcionales, hoy en día lenguajes como Java, C#, Python y JavaScript incorporan elementos de este enfoque.

La característica distintiva de la programación funcional es el uso de funciones como un tipo de dato. A diferencia de los paradigmas tradicionales, donde las variables almacenan valores u objetos, en el paradigma funcional las variables pueden almacenar **código fuente**

(funciones). Esto permite ejecutar y reasignar código de forma dinámica, ya que el contenido de estas variables puede cambiar y, por lo tanto, su comportamiento también.

En resumen, la programación funcional se basa en tratar funciones como datos, lo que permite una mayor flexibilidad y un enfoque diferente para resolver problemas.

▼ Funciones de orden superior

📌 Resumen: Funciones de orden superior

En la programación funcional, una **función de orden superior** es aquella que puede recibir una función como parámetro o devolver una función como resultado. Esta capacidad permite desacoplar el comportamiento de un método del código que lo ejecuta, lo cual es útil y flexible, especialmente en lenguajes como Java que utilizan interfaces para este propósito.

Este concepto, con fuerte base matemática, permite escribir código más reutilizable y adaptable, siendo una herramienta clave en muchos escenarios de programación moderna.

▼ Interfaces funcionales

📌 Resumen: Interfaces funcionales y funciones lambda en Java

En Java, un lenguaje orientado a objetos y fuertemente tipado, **no se puede usar directamente una función como tipo de dato**. Para permitir la programación funcional, se introducen las **interfaces funcionales**, que son interfaces con un solo método. Estas pueden marcarse con la anotación `@FunctionalInterface`, lo que ayuda a evitar errores al asegurar que no se agreguen más métodos.

✅ Ventajas y uso de interfaces funcionales:

- Permiten trabajar con funciones como datos.
- Se usan mucho con expresiones lambda y programación funcional.
- Ejemplo: una interfaz `Operacion` con un método `calcular(float a, float b)`.

🔧 Funciones Lambda

Las **expresiones lambda** permiten implementar de forma concisa una interfaz funcional **sin necesidad de crear una clase anónima o concreta**. Esto reduce el código y mejora la claridad.

📌 Sintaxis básica:

```
(parámetros) -> { cuerpo }
```

📉 Ejemplo con expresiones lambda:

```
Operacion suma = (float a, float b) -> a + b;
Operacion resta = (float a, float b) -> a - b;
Operacion multiplicacion = (float a, float b) -> a * b;
Operacion division = (float a, float b) -> {
    if (b != 0) return a / b;
    else throw new ArithmeticException("No se puede dividir entre cer
o.");
};
```

🧠 Ventaja clave:

Permiten evitar la creación de múltiples clases para implementar un solo método, logrando una **calculadora funcional más simple y flexible**.

¿Querés un esquema visual o ejemplo completo con `Scanner` también usando lambdas?

▼ Interfaces funcionales provistas

📌 Resumen: Interfaces funcionales provistas por Java

Java proporciona en su API (`java.util.function`) un conjunto de **interfaces funcionales estándar** para evitar que los desarrolladores tengan que definir nuevas interfaces para cada caso. **Se recomienda usar estas**

interfaces predefinidas en lugar de crear otras nuevas si ya existe una que se ajusta al problema.

🔑 Principales interfaces funcionales y su uso:

Interfaz	Parámetros	Retorno	Uso común
`Supplier<T>`	Ninguno	`T`	Generar valores (`Stream.generate`)
`Consumer<T>`	`T`	`void`	Consumir valores (`forEach`)
`Predicate<T>`	`T`	`boolean`	Filtros (`filter`)
`Function<T, R>`	`T`	`R`	Transformar (`map`)
`UnaryOperator<T>`	`T`	`T`	Transformar sin cambiar el tipo
`BinaryOperator<T>`	`T, T`	`T`	Operaciones binarias del mismo tipo
`DoubleBinaryOperator`	`double, double`	`double`	Igual que `BinaryOperator`, pero con `double`

🧮 Aplicación a la calculadora:

En lugar de definir una interfaz propia (`Operacion`), se puede usar directamente `DoubleBinaryOperator`:

```
DoubleBinaryOperator suma = (a, b) -> a + b;
DoubleBinaryOperator resta = (a, b) -> a - b;
DoubleBinaryOperator multiplicacion = (a, b) -> a * b;
DoubleBinaryOperator division = (a, b) -> {
    if (b != 0) return a / b;
    else throw new ArithmeticException("No se puede dividir entre cer
o.");
};
```

También se puede simplificar aún más:

```
DoubleBinaryOperator[] operaciones = {
    (a, b) -> a + b,
    (a, b) -> a - b,
    (a, b) -> a * b,
    (a, b) -> {
        if (b != 0) return a / b;
        else throw new ArithmeticException("No se puede dividir entre ce
ro.");
    }
};
```

Usar estas interfaces funcionales estándar **facilita el desarrollo, mejora la legibilidad y aprovecha mejor las herramientas del lenguaje**.

¿Querés que te prepare una tabla de equivalencias entre interfaz personalizada y provista por Java?

▼ Stream API

📌 Resumen: Stream API en Java

La **Stream API** de Java permite **procesar conjuntos de datos como flujos** en vez de manejar estructuras completas en memoria, como listas o arrays. La principal ventaja es que permite aplicar operaciones sobre cada elemento **sin tener que recorrerlos explícitamente**, facilitando un estilo declarativo y funcional de programación.

🔄 ¿Qué es un Stream?

Un **stream** es una **secuencia de datos** (como una cascada de elementos) sobre la cual se pueden aplicar múltiples operaciones encadenadas. Estas operaciones pueden **transformar, filtrar, limitar o recolectar** los datos.

📥 Cómo obtener un Stream

Origen	Método usado

Desde colecciones	`collection.stream()`
Desde arreglos	`Arrays.stream(array)`
Desde valores fijos	`Stream.of(val1, val2, ...)`
Flujos infinitos (secuencia)	`Stream.iterate(seed, unaryOperator)`
Flujos infinitos (aleatorio)	`Stream.generate(supplier)`
Desde fuentes específicas	Ej: `Files.lines(path)`, `getResultStream()`

> Los flujos infinitos deben cortarse con .limit(n) para evitar ciclos infinitos.

⚙️ Operaciones intermedias (retornan nuevos streams)

Estas transforman o filtran los datos **sin consumirlos aún**:

Operación	Descripción
`filter(Predicate)`	Filtra elementos según una condición booleana.
`distinct()`	Elimina elementos duplicados.
`map(Function)`	Transforma cada elemento a otro (puede ser de otro tipo).
`limit(n)`	Toma solo los primeros `n` elementos.
`sorted()`	Ordena los elementos (por `Comparable` o `Comparator`).

🧪 Ejemplos comunes

```
// Filtrar pares
Stream<Integer> pares = numeros.stream().filter(x -> x % 2 == 0);

// Calcular cuadrados
Stream<Integer> cuadrados = numeros.stream().map(x -> x * x);

// Generar 100 números aleatorios
Stream<Integer> aleatorios = Stream.generate(() -> new Random().ne
xtInt()).limit(100);
```

```
// Ordenar números
Stream<Integer> ordenados = numeros.stream().sorted();
```

✅ Conclusión

La Stream API permite trabajar con datos de forma más limpia, legible y eficiente, utilizando operaciones encadenadas y expresiones lambda para transformar, filtrar y recolectar información de manera fluida.

¿Querés que te prepare un resumen visual tipo esquema o tabla comparativa entre métodos?

▼ Operaciones terminales

✅ Resumen: Operaciones Terminales en Streams (Java)

Las **operaciones terminales** son el paso final en el procesamiento de un stream. **Consumen el flujo** y **devuelven un resultado o efecto**, sin generar un nuevo stream.

> ⚠️ Las operaciones intermedias no se ejecutan hasta que se invoca una terminal.

🔚 ¿Qué hacen las operaciones terminales?

- Ejecutan el procesamiento final del stream.
- Después de aplicarse, **el flujo queda consumido y no puede reutilizarse**.
- Pueden devolver:
  - Un valor (ej: `count`, `reduce`, `toList`)
  - Un efecto (ej: `forEach`)

📌 Operaciones Terminales Comunes

Método	Retorno	Descripción

`count()`	`long`	Cuenta la cantidad de elementos del stream.
`forEach()`	`void`	Ejecuta una acción (Consumer) por cada elemento.
`toList()`	`List<T>`	Recolecta todos los elementos en una lista.
`reduce()`	`Optional<T>`	Reduce todos los elementos a uno solo aplicando un `BinaryOperator`.
`min()` / `max()`	`Optional<T>`	Retornan el menor o mayor elemento del stream según un comparador.
`anyMatch()`	`boolean`	Retorna `true` si **algún** elemento cumple con un predicado.
`allMatch()`	`boolean`	Retorna `true` si **todos** los elementos cumplen con un predicado.
`noneMatch()`	`boolean`	Retorna `true` si **ningún** elemento cumple con un predicado.

🧪 Ejemplos de uso

```
// Imprimir cada número del stream
pares.forEach(x → System.out.println(x));

// Contar cuántos hay
long cantidad = pares.count();

// Sumar los cuadrados
int suma = cuadrados.reduce((a, b) → a + b).orElse(0);

// Verificar condiciones
if (aleatorios.anyMatch(x → x > 100)) {
    System.out.println("Hay algún número mayor a 100");
    if (aleatorios.allMatch(x → x > 100)) {
        System.out.println("Y todos son mayores a 100");
    }
}
```

🧠 Conclusión

Las operaciones terminales **activan** el stream y determinan cómo se va a recolectar o utilizar la información procesada. Sin una operación terminal, **nada se ejecuta**.

¿Querés que arme un esquema visual con todas las operaciones intermedias y terminales juntas para repaso?

▼ Referencias a métodos

✅ Resumen: Referencias a Métodos en Java

Las **referencias a métodos** (`method references`) son una forma concisa de usar métodos ya existentes como si fueran funciones. Se utilizan como alternativa a expresiones lambda cuando el método que se quiere usar **coincide en firma** con la función esperada.

🔷 Ventajas

- Más legibles y concisas que expresiones lambda.
- Reutilizan métodos ya definidos.
- No ejecutan el método, solo lo **referencian**.

📌 Tipos de Referencias a Métodos

Tipo	Sintaxis	Descripción
Método estático	`Clase::metodoEstatico`	Referencia a un método `static` de una clase.
Método de instancia de un objeto específico	`objeto::metodoDeInstancia`	Método no estático de un objeto existente.
Método de instancia de un tipo arbitrario	`Clase::metodoDeInstancia`	Método no estático, aplicado a cualquier objeto del tipo.
Constructor	`Clase::new`	Referencia al constructor de una clase.

🧪 Ejemplos

🔷 Impresión con `forEach`

```
numeros.stream().forEach(System.out::println);
```

> Equivale a: .forEach(x → System.out.println(x));

🔷 Leer archivo de números

```
List<Integer> numeros = Files.lines(Paths.get("numeros.txt"))
                        .map(Integer::valueOf)
                        .toList();
```

🔷 Leer archivo de personas

Constructor de la clase `Persona`:

```
public Persona(String linea) {
    String[] valores = linea.split(";");
    this.documento = Integer.valueOf(valores[0]);
    this.nombre = valores[1];
    this.apellido = valores[2];
    this.edad = Integer.valueOf(valores[3]);
}
```

Lectura del archivo CSV:

```
List<Persona> plantel = Files.lines(Paths.get("personas.csv"))
                        .map(Persona::new)
                        .toList();
```

🧠 Conclusión

Las referencias a métodos permiten escribir código más limpio y directo, especialmente útil en programación funcional con streams.

¿Querés un cuadro resumen visual con estos cuatro tipos de referencias?

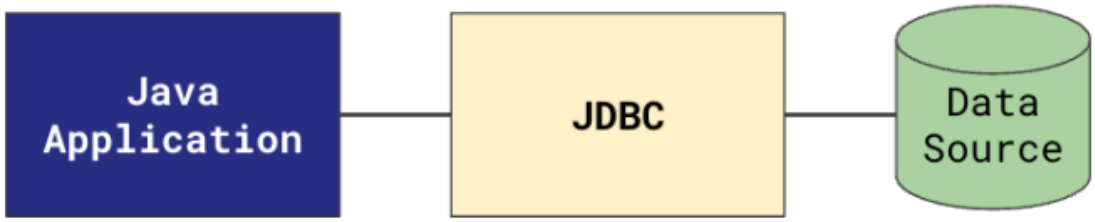
▼ JDBC (Java Database Connectivity)

▼ JDBC: Conectando Java con Bases de Datos

JDBC (Java Database Connectivity) conecta una aplicación Java a una fuente de datos como puede ser una DB.
Es una API que es usada para:

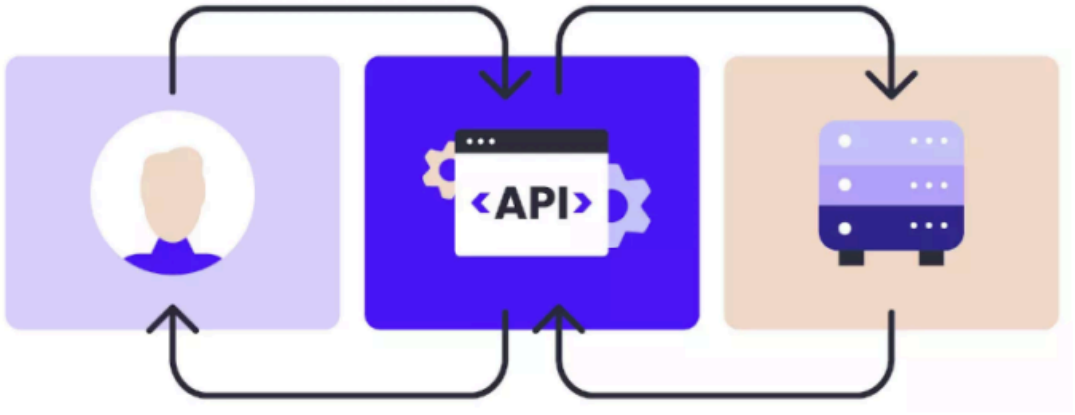
- Conectar a una fuente de datos
- Enviar consultas y actualizaciones
- Recupera y procesa el resultado

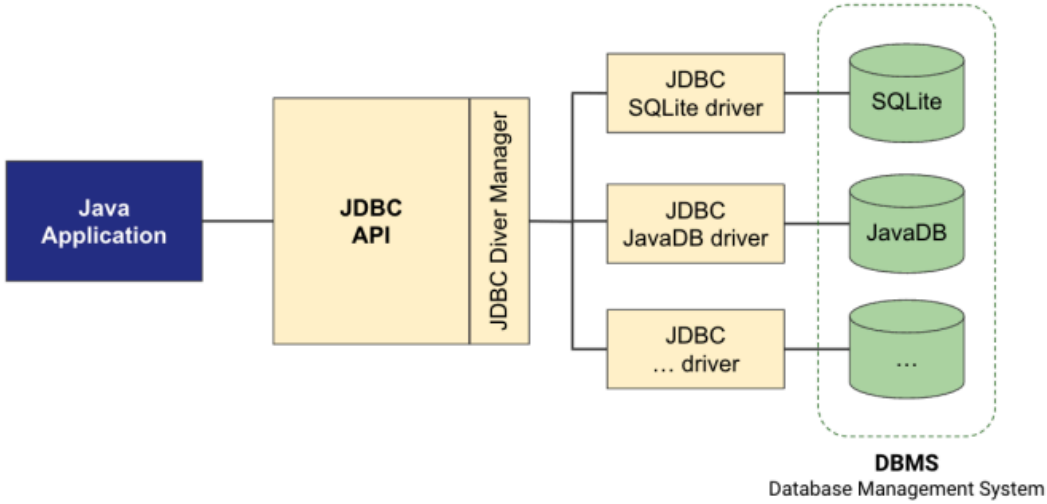
JDBC utiliza drivers para conectarse a la DB.



▼ JDBC - API

API (Application Programming Interfaces)
Es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación a través de un conjunto de reglas.





▼ JDBC - Implementación

✅ Resumen: JDBC Drivers y Conexión a Bases de Datos

JDBC (Java Database Connectivity) permite a las aplicaciones Java interactuar con bases de datos mediante controladores llamados **JDBC Drivers**.

🔷 Tipos de JDBC Drivers

Tipo	Nombre	Descripción	Ejemplo
Tipo 1	Bridge	Usa ODBC como intermediario. Requiere configuración externa.	JDBC-ODBC Bridge

Tipo 2	Native	Combina Java con código nativo (usa librerías del cliente).	Oracle OCI
Tipo 3	Network	Cliente Java se comunica con un servidor middleware que traduce hacia la base de datos.	Middleware personalizado
Tipo 4	Thin o Pure Java	100% Java. Se conecta directamente a la base sin componentes externos.	Oracle Thin Driver

> 🔶 Tipo 4 es el más usado hoy en día por ser portable y eficiente.

🔁 Pasos para interactuar con una Base de Datos

▼ **1. Cargar el driver** adecuado.

Antes de poder conectarse a la base de datos es necesario cargar el driver JDBC. Sólo hay que hacerlo una
única vez al comienzo de la aplicación:

```
Class.forName("com.mysql.jdbc.Driver");
```

El nombre del driver debe venir especificado en la documentación de la base de datos.

Se puede elevar la excepción ClassNotFoundException si hay un error en el nombre del driver o si el fichero .jar no está correctamente en el CLASSPATH o en el proyecto

```java
import java.math.BigDecimal;
import java.sql.*;

public class Inicio
{
    public static void main(String[] args)
            throws ClassNotFoundException, SQLException
    {
        Class.forName("oracle.jdbc.OracleDriver");

        Connection conn = DriverManager.getConnection(
                        "jdbc:oracle:thin:@localhost:1521:xe", "hr", "hrAdmin");

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOBS");
        System.out.println("Listado de Jobs");
        while (rs.next())
        {
            String id = rs.getString("JOB_ID");
            String title = rs.getString(2);
            int minSal = rs.getInt(3);
            int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
            System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
        }

        rs.close();
        stmt.close();
        conn.close();
    }
}
```

Registrar driver

Otra alternativa para lograr el mismo objetivo es utilizar el mecanismo que internamente usa la clase Driver, esto es invocar el método registerDriver de la clase DriverManager.

3 / 11

README.md 9/1/2023

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

En este caso evitamos la excepción ClassNotFoundException porque de no estar la clase no hubiera compilado pero; es necesario contar con el driver (librería.jar) para compilar la clase.

▼ **2.Establecer conexión** con la base (usualmente por red).(crear objeto de conexion)

Las bases de datos actúan como servidores y las aplicaciones como clientes que se comunican a través de la red. Un objeto Connection representa una conexión física entre el cliente y el servidor. Para crear una conexión se usa la clase DriverManager donde se especifica la URL, el nombre y la contraseña:

```
Connection conn = DriverManager.getConnection("jdbc:mysql://local
"usr", "pass");
```

El formato de la URL debe especificarse en el manual de la base de datos.
Ejemplo de Mysql con el driver JDBC:

```
jdbc:mysql://\<host>:\<puerto>:\<instancia>
```

Por ejemplo:

```
jdbc:mysql://localhost:3306/miBD
```

El nombre de usuario y la contraseña dependen también de la base de datos.


```java
import java.math.BigDecimal;
import java.sql.*;

public class Inicio
{
    public static void main(String[] args)
            throws ClassNotFoundException, SQLException
    {
        Class.forName("oracle.jdbc.OracleDriver");

        Connection conn = DriverManager.getConnection(
                        "jdbc:oracle:thin:@localhost:1521:xe", "hr", "hrAdmin");

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOBS");
        System.out.println("Listado de Jobs");
        while (rs.next())
        {
            String id = rs.getString("JOB_ID");
            String title = rs.getString(2);
            int minSal = rs.getInt(3);
            int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
            System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
        }

        rs.close();
        stmt.close();
        conn.close();
    }
}
```

Cada objeto Connection representa una conexión física con la base de datos.

Se pueden especificar más propiedades además del usuario y la password al crear una conexión. Estaspropiedades se pueden especificar usando métodos getConnection(...) sobrecargados de la clase

▼ Alternativas para crear Conexiones a la base de datos

Alternativas para crear Conexiones a la base de datos

```
String url = "jdbc:mysql://localhost:3306/sample";
String name = "root";
String password = "pass" ;
Connection c = DriverManager.getConnection(url, user, password);
```

```
String url =
   "jdbc:mysql://localhost:3306/sample?user=root&password=pass";
Connection c = DriverManager.getConnection(url);
```

```
String url = "jdbc:mysql://localhost:3306/sample";
Properties prop = new Properties();
prop.setProperty("user", "root");
prop.setProperty("password", "pass");
Connection c = DriverManager.getConnection(url, prop);
```

▼ **3.Enviar consultas SQL** y procesar los resultados. (Crear la sentencia)

Esta sentencia es la responsable de ejecutar las consultas a la DB. Una vez que tienes una conexión puedes ejecutar sentencias SQL:

Primero se crea el objeto Statement desde la conexión

Posteriormente se ejecuta la consulta y su resultado se devuelve como un ResultSet.

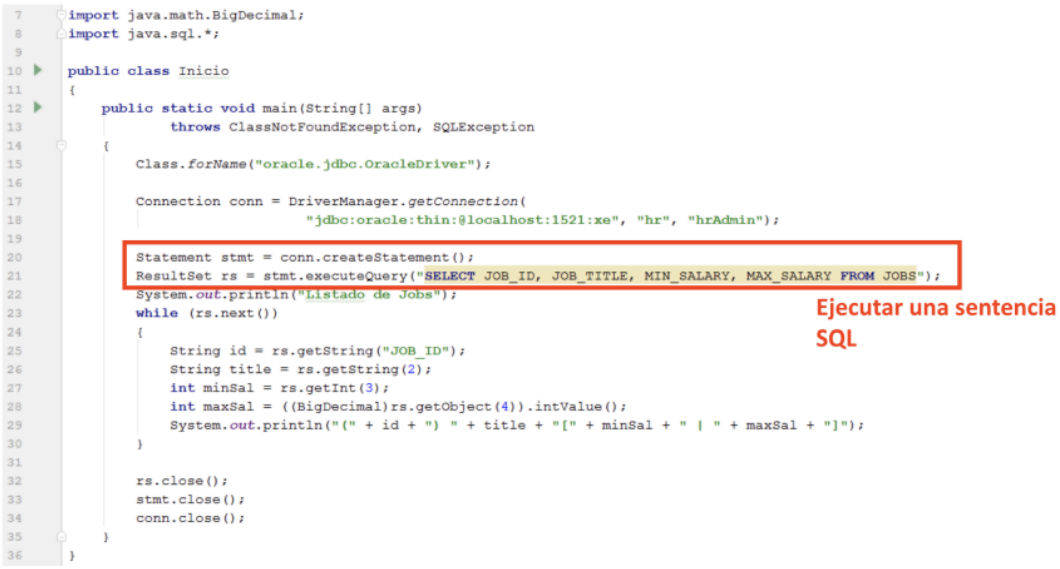
```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT ... FROM ... ");
```

Uso de Statement
Tiene diferentes métodos para ejecutar una sentencia

executeQuery(...): Se usa para sentencias SELECT. Devuelve un ResultSet.

executeUpdate(...): Se usa para sentencias INSERT, UPDATE, DELETE o sentencias DDL. Devuelve el número de filas afectadas por la sentencia.

execute(...): Método genérico de ejecución de consultas. Puede devolver uno o más ResulSet y uno o más contadores de filas afectadas.




```java
import java.math.BigDecimal;
import java.sql.*;

public class Inicio
{
    public static void main(String[] args)
            throws ClassNotFoundException, SQLException
    {
        Class.forName("oracle.jdbc.OracleDriver");

        Connection conn = DriverManager.getConnection(
                          "jdbc:oracle:thin:@localhost:1521:xe", "hr", "hrAdmin");

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOBS");
        System.out.println("Listado de Jobs");
        while (rs.next())
        {
            String id = rs.getString("JOB_ID");
            String title = rs.getString(2);
            int minSal = rs.getInt(3);
            int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
            System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
        }

        rs.close();
        stmt.close();
        conn.close();
    }
}
```

Acceso al conjunto de resultados
El ResultSet es el objeto que representa el resultado. No carga toda la información en memoria, internamente tiene un cursor que apunta a un fila concreta del resultado en la base de datos.

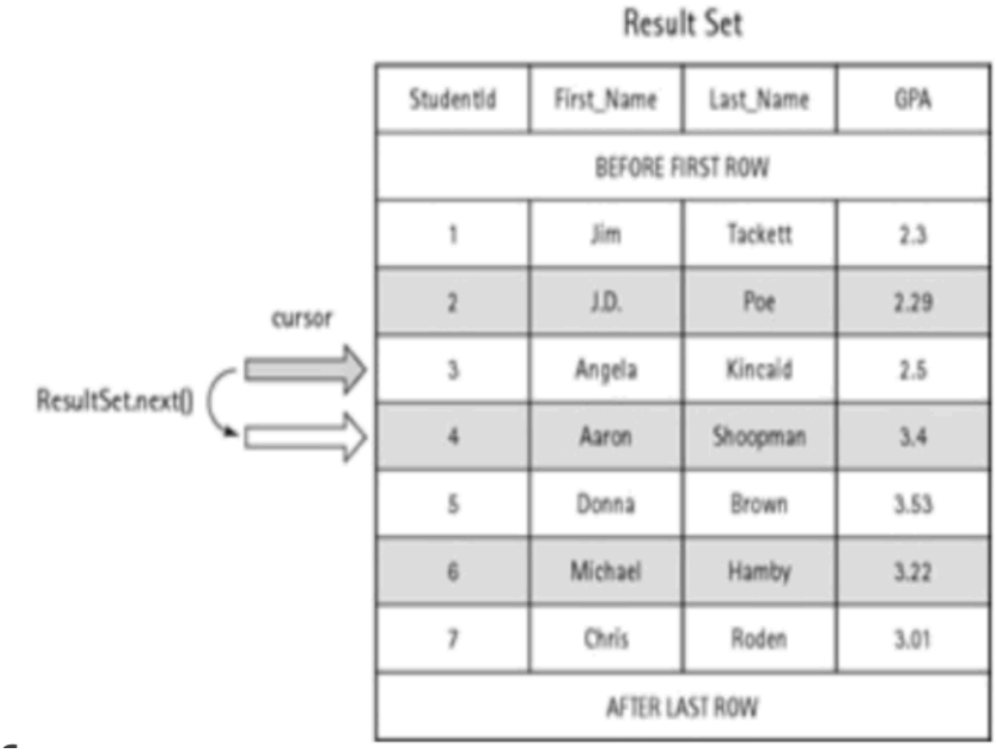
Hay que posicionar el cursor en cada fila y obtener la información de la misma.

```java
import java.math.BigDecimal;
import java.sql.*;

public class Inicio
{
    public static void main(String[] args)
            throws ClassNotFoundException, SQLException
    {
        Class.forName("oracle.jdbc.OracleDriver");

        Connection conn = DriverManager.getConnection(
                           "jdbc:oracle:thin:@localhost:1521:xe", "hr", "hrAdmin");

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOBS");
        System.out.println("Listado de Jobs");
        while (rs.next())
        {
            String id = rs.getString("JOB_ID");
            String title = rs.getString(2);
            int minSal = rs.getInt(3);
            int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
            System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
        }

        rs.close();
        stmt.close();
        conn.close();
```

Acceso al conjunto de resultados

▼ Posicionamiento del cursor

- El cursor puede estar en una fila concreta.
- También puede estar en dos filas especiales.
  - Antes de la primera fila (Before the First Row, BFR)
  - Después de la última fila (After the Last Row, ALR)
- Inicialmente el `ResultSet` está en BFR.
- `next()` mueve el cursor hacia delante,
  - devuelve `true` si se encuentra en una fila concreta
  - y `false` si alcanza el ALR.



▼ Obtención de los datos de la fila

Cuando el `ResultSet` se encuentra en una fila concreta se pueden usar los métodos de acceso a las columnas:

- String getString(String columnLabel)
- String getString(int columnIndex)
- int getInt(String columnLabel)
- int getInt(int columnIndex)
- ... (existen dos métodos por cada tipo)

```
while (rs.next())
{
    String id = rs.getString("JOB_ID");
    String title = rs.getString(2);
    int minSal = rs.getInt(3);
    int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
    System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
}
```

▼ **4.Cerrar recursos** cuando ya no se usan.

Cuando se termina de usar una `Connection`, un `Statement` o un `ResultSet` es necesario liberar los recursos que necesitan.

README.md 9/1/2023

Puesto que la información de un ResultSet no se carga en memoria, existen conexiones de red abiertas.

Métodos close():

- `ResultSet.close()` – Libera los recursos del ResultSet. Se cierra automáticamente al cerrar el Statement que lo creó o al reejecutar el Statement.
- `Statement.close()` – Libera los recursos del Statement.
- `Connection.close()` – Finaliza la conexión con la base de datos

```java
import java.math.BigDecimal;
import java.sql.*;

public class Inicio
{
    public static void main(String[] args)
            throws ClassNotFoundException, SQLException
    {
        Class.forName("oracle.jdbc.OracleDriver");

        Connection conn = DriverManager.getConnection(
                           "jdbc:oracle:thin:@localhost:1521:xe", "hr", "hrAdmin");

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOBS");
        System.out.println("Listado de Jobs");
        while (rs.next())
        {
            String id = rs.getString("JOB_ID");
            String title = rs.getString(2);
            int minSal = rs.getInt(3);
            int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
            System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
        }

        rs.close();
        stmt.close();
        conn.close();
    }
}
```

Liberar Recursos

▼ **5.Manejar errores** adecuadamente (ej: excepciones).

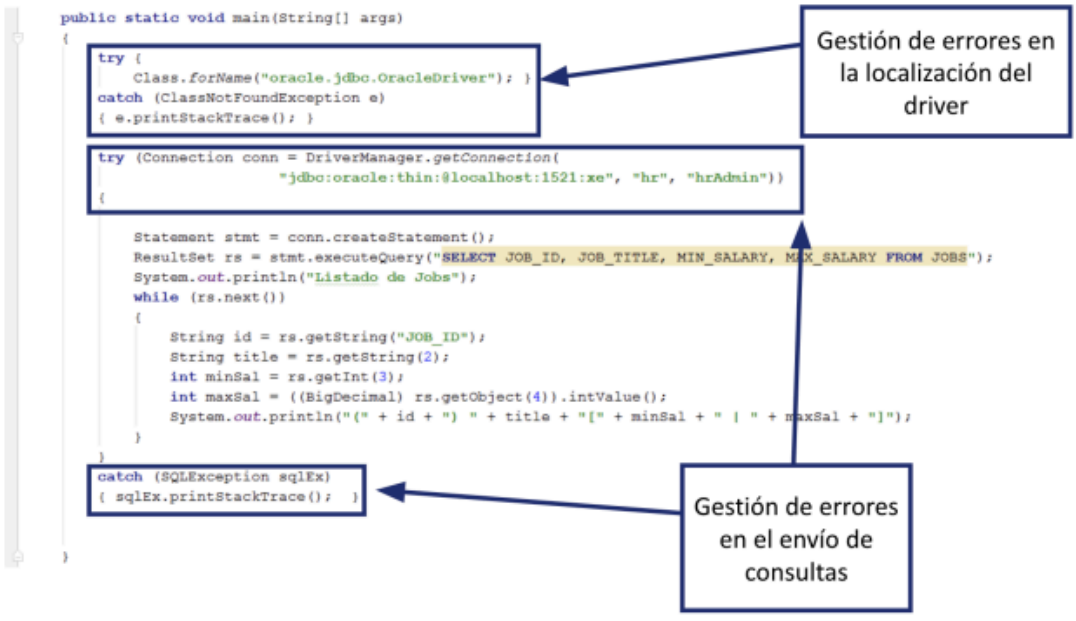
5. Manejar los errores

Hay que gestionar los errores apropiadamente:

- Se pueden producir excepciones `ClassNotFoundException` si no se encuentra el driver.
- Se pueden producir excepciones `SQLException` al interactuar con la base de datos
    - SQL mal formado
    - Conexión de red rota
    - Problemas de integridad al insertar datos (claves duplicadas)

```java
import java.math.BigDecimal;
import java.sql.*;

public class Inicio
{
    public static void main(String[] args)
            throws ClassNotFoundException, SQLException
    {
        Class.forName("oracle.jdbc.OracleDriver");

        Connection conn = DriverManager.getConnection(
                           "jdbc:oracle:thin:@localhost:1521:xe", "hr", "hrAdmin");

        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY FROM JOBS");
        System.out.println("Listado de Jobs");
        while (rs.next())
        {
            String id = rs.getString("JOB_ID");
            String title = rs.getString(2);
            int minSal = rs.getInt(3);
            int maxSal = ((BigDecimal)rs.getObject(4)).intValue();
            System.out.println("(" + id + ") " + title + "[" + minSal + " | " + maxSal + "]");
        }

        rs.close();
        stmt.close();
        conn.close();
    }
}
```

Manejar los errores



▼ Sentencias SQL

Con JDBC se pueden usar diferentes tipos de Statement:

- `Statement`: SQL estático en tiempo de ejecución, no acepta parámetros.

```
Statement stmt = conn.createStatement();
```

- `PreparedStatement`: Para ejecutar la mismas sentencia muchas veces (la "prepara"). Acepta parámetros


```
PreparedStatement ps = conn.prepareStatement(...);
```

- `CallableStatement`: Llamadas a procedimientos almacenados

```
CallableStatement s = conn.prepareCall(...);
```

▼ Ejemplo JDBC

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JPASample {
   public static void main(String[] args) {
   // Crear EntityManagerFactory
   EntityManagerFactory emf = Persistence.createEntityManagerFactory(
   // Crear EntityManager
   EntityManager em = emf.createEntityManager();
   // Iniciar transacción
   em.getTransaction().begin();
```

```
    // Operación CRUD
    MyEntity entity = new MyEntity();
    entity.setName("Sample");
    em.persist(entity);
    // Commit
    em.getTransaction().commit();
    // Cerrar recursos
    em.close();
    emf.close();
    }
}
```

▼ Resumen

- **Maven y Herramientas en Java**:
  - **Maven** es un gestor de dependencias en Java. A través del archivo `pom.xml`, se gestionan las dependencias y se facilita el manejo de bibliotecas externas. Además, se cubren aspectos básicos del uso del **Scanner** para leer entradas de usuarios, una parte importante para la interacción con el sistema.
- **Colecciones en Java**:
  - Las **colecciones** en Java proporcionan estructuras de datos como **List**, **Set**, **Map**, y **Queue**. Se discuten sus diferencias con los **arrays**, las **ventajas** de las colecciones (flexibilidad y dinamismo), y se presentan métodos comunes para su manipulación. Se menciona también el uso de **iteradores** y el patrón **Iterator** para recorrer colecciones de manera eficiente.

📊 Comparación rápida:

Característica	ArrayList	HashSet	HashMap

Permite duplicados	Sí	No	No (en claves)
Mantiene orden	Sí (en general)	No (excepto `LinkedHashSet`)	Depende de la implementación
Acceso por índice	Sí	No	No (acceso por clave)
Relación clave-valor	No	No	Sí

▼ stream

🔄 ¿Qué es un Stream?

Un **stream** es una **secuencia de datos** (como una cascada de elementos) sobre la cual se pueden aplicar múltiples operaciones encadenadas. Estas operaciones pueden **transformar, filtrar, limitar o recolectar** los datos.

▼ **JPA (Java Persistence API)** es una especificación de Java para la gestión de datos relacionales en aplicaciones Java. JPA proporciona un enfoque estándar para almacenar, recuperar, actualizar y eliminar datos en bases de datos relacionales utilizando objetos Java.

Características clave de JPA:

1. **Mapeo objeto-relacional (ORM)**:
   - JPA permite mapear las clases de Java a tablas de bases de datos, lo que significa que las entidades Java se representan como registros de una tabla en la base de datos.
   - Cada objeto de tipo entidad corresponde a una fila en la tabla de la base de datos, y sus atributos corresponden a las columnas de la tabla.
2. **Entidades**:
   - En JPA, las clases de Java que representan datos son denominadas **entidades**. Estas clases deben estar anotadas con `@Entity` para indicar que son entidades persistentes.

- Un objeto de entidad puede ser guardado en la base de datos a través de operaciones de persistencia.

3. **Unidad de trabajo**:
    - JPA usa una **unidad de persistencia**, que agrupa una serie de operaciones de bases de datos dentro de un contexto transaccional. Las operaciones en entidades se realizan a través de un **EntityManager**, que gestiona las transacciones y la persistencia.
4. **Consultas JPQL (Java Persistence Query Language)**:
    - JPA usa **JPQL** para realizar consultas a la base de datos. Es similar a SQL, pero se basa en objetos Java en lugar de en tablas y columnas. Las consultas JPQL trabajan directamente con las entidades y sus atributos.
5. **Transacciones**:
    - JPA trabaja con transacciones, lo que significa que las operaciones de persistencia (como guardar, eliminar o actualizar datos) son ejecutadas dentro de una transacción para asegurar la integridad de los datos.
6. **EntityManager**:
    - El **EntityManager** es el principal componente de JPA para interactuar con las entidades. Se utiliza para gestionar el ciclo de vida de las entidades (crear, leer, actualizar y eliminar) y ejecutar consultas.

Ejemplo básico de una entidad JPA:

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Persona {
    @Id
    private Long id;
    private String nombre;
```

```
    private int edad;

    // Getters y Setters
}
```

Uso básico de JPA:

1. **Persistir una entidad**:

```
EntityManager em = entityManagerFactory.createEntityManager();
em.getTransaction().begin();

Persona persona = new Persona();
persona.setId(1L);
persona.setNombre("Juan");
persona.setEdad(30);

em.persist(persona);
em.getTransaction().commit();
em.close();
```

2. **Consultar datos con JPQL**:

```
EntityManager em = entityManagerFactory.createEntityManager();
TypedQuery<Persona> query = em.createQuery("SELECT p FROM Pe
rsona p WHERE p.nombre = :nombre", Persona.class);
query.setParameter("nombre", "Juan");
Persona persona = query.getSingleResult();
em.close();
```

Ventajas de usar JPA:

- **Abstracción**: JPA proporciona una capa de abstracción sobre el acceso a bases de datos, lo que facilita el trabajo con bases de datos sin tener que escribir SQL directamente.

- **Portabilidad**: Al usar JPA, las aplicaciones Java son independientes del proveedor de bases de datos, lo que facilita la portabilidad entre distintos sistemas de bases de datos.
- **Optimización de consultas**: JPA optimiza el acceso a la base de datos con mecanismos como el **caching** y el **fetching** perezoso (lazy loading).

En resumen, **JPA** es un estándar en Java para la gestión de la persistencia de datos, facilitando el trabajo con bases de datos mediante el uso de objetos Java en lugar de código SQL.

- **JDBC (Java Database Connectivity)**:
  - JDBC es una API en Java que permite conectar aplicaciones con bases de datos. El documento cubre los diferentes tipos de **drivers JDBC** y cómo interactuar con bases de datos a través de **conexiones**, **consultas SQL**, y **gestión de errores**. Se detallan los pasos para interactuar con bases de datos, desde la carga de drivers hasta el procesamiento de los resultados.