Examen

Cambios importantes han habido en Java:

Java 8 (2014):

Expresiones Lambda y Interfaces Funcionales

Stream API

Nueva API de Fecha y Hora

Optional para manejar valores nulos

Java 9 (2017):

Sistema de módulos (Project Jigsaw)

JShell (REPL para Java)

Mejoras en la API de Colecciones

Java 10 (2018):

Inferencia de tipos de variables locales (var)

Java 11 (2018):

Ejecución de archivos fuente sin compilación previa

Nuevo cliente HTTP

Java 14 (2020):

Switch expressions (versión final)

Records (preview)

Java 15 (2020):

Sealed Classes (preview)

Java 16 (2021):

Records (versión final)

Pattern Matching para instanceof

Java 17 (2021) - LTS:

Sealed Classes (versión final)

Pattern Matching para switch (preview)

Lambdas:

Son una manera concisa de implementar interfaces funcionales. Las interfaces funcionales son aquellas que tienen un único método abstracto. Los lambdas permiten representar este método de una forma compacta y sin necesidad de escribir clases anónimas.

(parametros) -> expresión

BiFunction<Integer, Integer, Integer> suma = (a, b) -> a + b;

System.out.println(suma.apply(5, 3)); -> imprime 8

Interfaces funcionales:

Una interfaz funcional en Java tiene una sola responsabilidad, es decir, define un comportamiento con un único método abstracto que debe ser implementado. Para facilitar esto, Java introdujo la anotación @FunctionalInterface, que se puede usar opcionalmente para dejar claro que la interfaz está diseñada para ser funcional. Esta anotación es opcional pero ayuda a garantizar que no se agreguen más métodos abstractos a la interfaz por error.

```
@FunctionalInterface
public interface Operacion {
   int ejecutar(int a, int b);
}
Operacion suma = (a, b) -> a + b;
System.out.println(suma.ejecutar(5, 3)); // Imprime 8
```

1. Function<T, R>

Recibe un argumento de tipo T y devuelve un resultado de tipo R.

```
Function<String, Integer> longitud = s -> s.length();
System.out.println(longitud.apply("Hola")); // Imprime 4
```

2. Consumer<T>

Recibe un argumento de tipo T y no devuelve nada (solo consume el dato).

Consumer<String> imprimir = s -> System.out.println(s);

imprimir.accept("Hola mundo"); // Imprime "Hola mundo"

3. Supplier<T>

No recibe argumentos y devuelve un valor de tipo T. Es como un "proveedor" de valores

Supplier<Double> aleatorio = () -> Math.random();

System.out.println(aleatorio.get()); // Imprime un número aleatorio

4. Predicate<T>

Recibe un argumento de tipo T y devuelve un boolean. Sirve para evaluar una condición sobre el argumento.

Predicate<Integer> esPar = n -> n % 2 == 0;

System.out.println(esPar.test(4)); // Imprime true

System.out.println(esPar.test(3)); // Imprime false

5. BiFunction<T, U, R>

Recibe dos argumentos de tipos T y U y devuelve un resultado de tipo R.

BiFunction<Integer, Integer, Integer> multiplicacion = (a, b) -> a * b;

System.out.println(multiplicacion.apply(2, 3)); // Imprime 6

6. UnaryOperator<T>

Es una subinterfaz de Function<T, T>. Recibe un solo argumento y devuelve un resultado del mismo tipo.

UnaryOperator<Integer> cuadrado = n -> n * n;

System.out.println(cuadrado.apply(5)); // Imprime 25

7. BinaryOperator<T>

Es una subinterfaz de BiFunction<T, T, T>. Recibe dos argumentos del mismo tipo y devuelve un valor del mismo tipo.

BinaryOperator<Integer> suma = (a, b) -> a + b;

System.out.println(suma.apply(2, 3)); // Imprime 5

8. BiConsumer<T, U>

Recibe dos argumentos de tipos T y U, pero no devuelve nada.

```
BiConsumer<String, Integer> imprimirInfo = (nombre, edad) ->
   System.out.println(nombre + " tiene " + edad + " años.");


imprimirInfo.accept("Ana", 25); // Imprime "Ana tiene 25 años."
```

Stream:

Un stream es una secuencia de elementos que soporta diferentes tipos de operaciones que pueden ser **intermedias** o **terminales**. Lo importante de los streams es que no almacenan datos, sino que son una forma de operar sobre datos de manera funcional y con una programación perezosa (lazy), es decir, las operaciones no se ejecutan hasta que se llame a una operación terminal.

Operaciones intermedias

1. filter(Predicate<T> predicate)

Filtra los elementos que cumplen con el predicado dado.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> pares = numeros.stream()
   .filter(n -> n % 2 == 0)
   .collect(Collectors.toList());
System.out.println(pares); // Imprime [2, 4]
```

2. map(Function<T, R> mapper)

Transforma cada elemento del stream en otro mediante la función dada.

```
List<String> nombres = Arrays.asList("Ana", "Carlos", "Daniel");

List<Integer> longitudes = nombres.stream()
   .map(String::length)
   .collect(Collectors.toList());
```

```
System.out.println(longitudes); // Imprime [3, 6, 6]
```

3. sorted()

Ordena los elementos en orden natural o usando un comparador.

```
List<String> nombres = Arrays.asList("Carlos", "Ana", "Daniel");
List<String> nombresOrdenados = nombres.stream()
  .sorted()
  .collect(Collectors.toList());
System.out.println(nombresOrdenados); // Imprime [Ana, Carlos, Daniel]
```

4. distinct()

Elimina elementos duplicados.

```
List<Integer> numeros = Arrays.asList(1, 2, 2, 3, 3, 4);
List<Integer> numerosUnicos = numeros.stream()
  .distinct()
  .collect(Collectors.toList());
System.out.println(numerosUnicos); // Imprime [1, 2, 3, 4]
```

Operaciones Terminales Comunes

1. forEach(Consumer<T> action)

Aplica una acción sobre cada elemento.

```
List<String> nombres = Arrays.asList("Ana", "Beatriz", "Carlos");
nombres.stream()
  .forEach(nombre -> System.out.println(nombre));
```

2. collect(Collector<T, A, R> collector)

Recoge los elementos del stream en una colección o estructura de datos.

```
List<String> nombres = Arrays.asList("Ana", "Beatriz", "Carlos");
Set<String> setNombres = nombres.stream()
  .collect(Collectors.toSet());
```

```
System.out.println(setNombres); // Imprime los nombres en un Set
```

3.count()

Cuenta el número de elementos en el stream.

```
List<String> nombres = Arrays.asList("Ana", "Beatriz", "Carlos");

long cantidad = nombres.stream()
   .count();

System.out.println(cantidad); // Imprime 3
```

4. findFirst()

Devuelve el primer elemento del stream.

```
List<String> nombres = Arrays.asList("Ana", "Beatriz", "Carlos");

Optional<String> primero = nombres.stream()
   .findFirst();

primero.ifPresent(System.out::println); // Imprime "Ana"
```

5.reduce(BinaryOperator<T> accumulator)

Combina los elementos del stream en uno solo, usando una operación acumulativa.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4);

int suma = numeros.stream()
   .reduce(0, (a, b) -> a + b);

System.out.println(suma); // Imprime 10
```

Streams Paralelos

Java permite procesar los streams en paralelo, lo que puede mejorar el rendimiento en algunos casos donde se manejen grandes volúmenes de datos. Para hacer un stream paralelo, simplemente usamos parallelStream() en lugar de stream():

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

int sumaParalela = numeros.parallelStream()
   .reduce(0, Integer::sum);

System.out.println(sumaParalela); // Imprime 21
```

Ventajas de los Streams

1. **Declarativo**: Permite expresar operaciones sobre colecciones de manera más clara y concisa.
2. **Lazy**: Las operaciones intermedias no se ejecutan hasta que se realiza una operación terminal.
3. **Inmutable**: No modifica la fuente original de datos.
4. **Paralelo**: Fácilmente se puede convertir en paralelo para mejorar el rendimiento.
5. **Fluidez**: Las operaciones intermedias se pueden encadenar de manera fluida.

Patrones de diseño

Creacionales:

1. Singleton:

- **Qué es:** Garantiza que solo haya una única instancia de una clase en toda la aplicación y proporciona un acceso global a esa instancia.
- **Cuándo usarlo:** Cuando necesitas asegurarte de que una clase tenga exactamente una instancia en todo el ciclo de vida de la aplicación, como la clase que maneja la configuración global o la conexión a una base de datos.
- **Ejemplo práctico:** Una aplicación bancaria solo debe tener una conexión única a la base de datos para evitar conflictos de concurrencia.

2. Multiton:

- **Qué es:** Similar al Singleton, pero permite múltiples instancias controladas, cada una identificada por una clave única.
- **Cuándo usarlo:** Cuando necesitas gestionar varias instancias únicas de una clase, pero con características particulares, por ejemplo, diferentes configuraciones para cada cliente en una aplicación multicliente.
- **Ejemplo práctico:** Si gestionas diferentes conexiones a bases de datos, pero cada cliente tiene su propia configuración, puedes usar un Multiton para asegurarte de que cada cliente tenga su instancia de base de datos propia y única.

3. Factory Method:

- **Qué es:** Proporciona una interfaz para crear objetos sin especificar las clases concretas de estos. Las subclases son las que deciden qué objetos crear.
- **Cuándo usarlo:** Cuando tu sistema necesita crear objetos sin especificar exactamente qué clase de objeto crear (esto permite la extensión del sistema de manera más flexible).
- **Ejemplo práctico:** Si tienes una aplicación de transporte, puedes usar el Factory Method para crear diferentes vehículos (carros, bicicletas), y permitir que en el futuro se puedan agregar otros tipos de vehículos sin alterar el código existente.

4. Abstract Factory:

- **Qué es:** Un patrón que permite crear familias de objetos relacionados sin especificar las clases concretas.
- **Cuándo usarlo:** Cuando tienes varias familias de objetos y quieres garantizar que los objetos de una familia sean compatibles entre sí.
- **Ejemplo práctico:** Si tienes una aplicación gráfica que debe funcionar tanto en Windows como en Mac, puedes usar una Abstract Factory para crear los botones y ventanas específicos para cada sistema operativo sin cambiar el código central.

5. Builder:

- **Qué es:** Facilita la construcción de un objeto complejo paso a paso, separando su creación de su representación final.
- **Cuándo usarlo:** Cuando la construcción de un objeto es compleja y puede tener muchas configuraciones, el patrón Builder te permite crear variantes de ese objeto sin tener que usar constructores con múltiples parámetros o múltiples clases para cada variante.
- **Ejemplo práctico:** Si estás construyendo una casa, puedes tener un constructor que permita agregar diferentes características (habitaciones, garajes) de manera flexible sin tener que crear una clase diferente para cada tipo de casa.

6. Prototype:

- **Qué es:** Permite crear nuevos objetos clonando una instancia existente, en lugar de crear una nueva desde cero.

- **Cuándo usarlo:** Cuando crear un objeto desde cero es costoso en términos de recursos o tiempo y quieres crear copias (o prototipos) de un objeto existente.
- **Ejemplo práctico:** En un videojuego, puedes tener un personaje que sea clonado para diferentes situaciones, pero con variaciones mínimas, evitando crear nuevos personajes desde cero cada vez.

7. Lazy Initialization:

- **Qué es:** Retrasa la creación o inicialización de un objeto hasta que realmente se necesite.
- **Cuándo usarlo:** Cuando un objeto es costoso de crear (en términos de tiempo o recursos) y puede que no siempre sea necesario.
- **Ejemplo práctico:** En una aplicación que se conecta a una base de datos, puedes retrasar la creación de la conexión hasta que el usuario realmente necesite acceder a la base de datos, optimizando así el uso de recursos.

8. Object Pool:

- **Qué es:** Mantiene un conjunto de objetos listos para ser reutilizados, evitando la creación repetitiva de objetos costosos.
- **Cuándo usarlo:** Cuando la creación y destrucción de objetos es costosa y se necesita mejorar el rendimiento mediante la reutilización de instancias.
- **Ejemplo práctico:** En un servidor web, las conexiones a bases de datos pueden reutilizarse en lugar de crear y destruir una nueva conexión para cada solicitud de usuario, lo que mejora el rendimiento.

Estructurales:

1. Adapter (Adaptador)

- **Qué es:** Permite que dos interfaces que son incompatibles puedan trabajar juntas. El patrón Adapter convierte la interfaz de una clase en otra interfaz que el cliente espera, sin modificar las clases involucradas.
- **Cuándo usarlo:** Cuando quieres que un sistema trabaje con una clase que tiene una interfaz incompatible o no es posible modificarla directamente.
- **Ejemplo práctico:** Supón que tu sistema tiene que utilizar una biblioteca externa para procesar pagos, pero la interfaz de esa biblioteca no coincide con

la interfaz que tu sistema espera. En lugar de modificar ambas partes, puedes crear un adaptador que haga que ambas interfaces sean compatibles.

2. Bridge (Puente)

- **Qué es:** Desacopla una abstracción de su implementación, permitiendo que ambas evolucionen de manera independiente. El patrón Bridge separa el "qué hace" (abstracción) del "cómo lo hace" (implementación).
- **Cuándo usarlo:** Cuando tienes múltiples dimensiones de variación en tu código (por ejemplo, una funcionalidad puede variar en diferentes plataformas o versiones), y no quieres duplicar código.
- **Ejemplo práctico:** Si estás creando una aplicación gráfica que debe funcionar en Windows, macOS y Linux, puedes usar un Bridge para desacoplar la lógica de la aplicación de los detalles de implementación específicos de cada plataforma.

3. Composite (Composición)

- **Qué es:** Permite que los objetos individuales y las composiciones de objetos se traten de manera uniforme. Este patrón facilita trabajar con estructuras jerárquicas, permitiendo que objetos individuales y grupos de objetos se traten de la misma manera.
- **Cuándo usarlo:** Cuando necesitas trabajar con estructuras de objetos compuestas, como árboles o gráficos, y quieres tratar los objetos individuales y las composiciones de manera similar.
- **Ejemplo práctico:** En una empresa, los empleados pueden ser gerentes o trabajadores individuales. Los gerentes pueden tener subordinados (otros empleados), y con Composite, puedes tratarlos a ambos de la misma manera sin tener que diferenciarlos.

4. Decorator (Decorador)

- **Qué es:** Permite agregar responsabilidades adicionales a un objeto de manera dinámica sin modificar su estructura básica. Es una alternativa flexible a la herencia para extender la funcionalidad de una clase.
- **Cuándo usarlo:** Cuando quieres agregar o modificar el comportamiento de objetos en tiempo de ejecución sin modificar las clases base.

- **Ejemplo práctico:** En una aplicación de mensajería, puedes tener un objeto Mensaje básico al cual quieres agregar funcionalidad extra como encriptación o compresión, pero sin alterar la clase Mensaje original.

5. Facade (Fachada)

- **Qué es:** Proporciona una interfaz unificada y simplificada a un sistema complejo que tiene múltiples subsistemas o interfaces. El patrón Facade hace que un sistema sea más fácil de usar, ocultando las complejidades internas.
- **Cuándo usarlo:** Cuando tienes un sistema con muchas partes o subsistemas complejos y quieres simplificar la interfaz para el usuario.
- **Ejemplo práctico:** En un sistema de compra en línea, podrías tener diferentes subsistemas para pagos, inventario, envíos, etc. Con una Facade, puedes simplificar el proceso de compra para el cliente, exponiendo solo los métodos clave (como "realizar pedido") y ocultando la complejidad interna.

6. Flyweight (Peso Ligero)

- **Qué es:** Optimiza el uso de memoria compartiendo objetos que son iguales o muy similares. Es útil cuando necesitas crear una gran cantidad de objetos similares y quieres reducir el consumo de memoria.
- **Cuándo usarlo:** Cuando tienes muchos objetos similares o repetidos que consumen demasiada memoria, puedes usar Flyweight para compartir esos objetos en lugar de crear nuevas instancias cada vez.
- **Ejemplo práctico:** En un editor gráfico, puedes tener miles de líneas y formas, muchas de las cuales tienen los mismos atributos (como color o grosor). El patrón Flyweight te permite compartir estos atributos comunes entre muchas instancias para ahorrar memoria.

7. Proxy (Proxy)

- **Qué es:** Proporciona un sustituto o intermediario para controlar el acceso a otro objeto. El patrón Proxy se utiliza cuando quieres controlar o retrasar el acceso a un objeto costoso de crear, cargar o inicializar.
- **Cuándo usarlo:** Cuando necesitas un control adicional sobre el acceso a un objeto (por ejemplo, para agregar lógica de seguridad, controlar el acceso remoto o diferir la carga).

- **Ejemplo práctico:** Imagina que tienes un objeto pesado, como una imagen de alta resolución. En lugar de cargarla inmediatamente, puedes usar un Proxy para diferir su carga hasta que sea necesario mostrarla en pantalla, optimizando el rendimiento.

Comportamiento:

1. Chain of Responsibility:

- **Qué es:** Permite pasar una solicitud a través de una cadena de manejadores. Cada manejador decide si procesa la solicitud o la pasa al siguiente.
- **Cuándo usarlo:** Cuando tienes múltiples objetos que pueden manejar una solicitud, y el manejador no se conoce a priori.
- **Ejemplo práctico:** Sistemas de aprobación de documentos donde diferentes niveles (gerente, director, etc.) pueden aprobar una solicitud según su rol.

2. Command:

- **Qué es:** Encapsula una solicitud como un objeto, permitiendo parametrizar clientes con diferentes solicitudes, hacer cola o registrar solicitudes.
- **Cuándo usarlo:** Cuando necesitas desacoplar el objeto que invoca la operación del objeto que sabe cómo llevarla a cabo.
- **Ejemplo práctico:** Sistemas de gestión de operaciones que permiten deshacer o rehacer acciones, como editores de texto.

3. Iterator:

- **Qué es:** Proporciona una forma de recorrer los elementos de una colección sin exponer su representación interna.
- **Cuándo usarlo:** Cuando necesitas acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente.
- **Ejemplo práctico:** Recorrer listas, árboles o conjuntos sin exponer la estructura de datos subyacente.

4. Mediator:

- **Qué es:** Define un objeto que gestiona la comunicación entre un conjunto de objetos para reducir las dependencias entre ellos.

- **Cuándo usarlo:** Cuando un conjunto de objetos se comunican de manera compleja y esta comunicación es difícil de cambiar.
- **Ejemplo práctico:** Comunicación entre diferentes módulos en un sistema de interfaz de usuario sin que estén acoplados directamente.

5. Memento:

- **Qué es:** Permite capturar y restaurar el estado interno de un objeto sin violar la encapsulación.
- **Cuándo usarlo:** Cuando necesitas proporcionar la capacidad de deshacer en tu aplicación.
- **Ejemplo práctico:** Guardar el estado en un juego o una aplicación para restaurarlo posteriormente.

6. Observer:

- **Qué es:** Permite a un objeto notificar automáticamente a sus dependientes cuando cambia su estado.
- **Cuándo usarlo:** Cuando un cambio en un objeto requiere cambiar otros, y no sabes cuántos objetos deben cambiar.
- **Ejemplo práctico:** Implementación en un sistema de suscripción/publicación como notificaciones en una aplicación.

7. State:

- **Qué es:** Permite que un objeto altere su comportamiento cuando cambia su estado interno, pareciendo cambiar de clase.
- **Cuándo usarlo:** Cuando el comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución.
- **Ejemplo práctico:** Implementación de un cajero automático que cambia su comportamiento según el estado (tarjeta insertada, sin fondos, etc.).

8. Strategy:

- **Qué es:** Define una familia de algoritmos y permite seleccionar uno de ellos en tiempo de ejecución.
- **Cuándo usarlo:** Cuando tienes múltiples algoritmos para una tarea específica y el cliente decide cuál usar.

- **Ejemplo práctico:** Cálculo de rutas en aplicaciones con diferentes estrategias (más rápida, más corta, etc.).

9. Template Method:

- **Qué es:** Define el esqueleto de un algoritmo, permitiendo a las subclases implementar algunos pasos sin modificar la estructura del algoritmo.
- **Cuándo usarlo:** Cuando tienes un algoritmo con pasos fijos pero la implementación de algunos pasos puede variar.
- **Ejemplo práctico:** Aplicaciones donde ciertas fases de un proceso son fijas y otras pueden ser personalizadas, como el proceso de creación de documentos.

10. Visitor:

- **Qué es:** Permite definir nuevas operaciones sin cambiar las clases de los elementos sobre los que opera.
- **Cuándo usarlo:** Cuando tienes que realizar operaciones en elementos de una estructura de objetos y no quieres cambiar las clases de estos elementos.
- **Ejemplo práctico:** Análisis y procesamiento de estructuras complejas como árboles de sintaxis abstracta en compiladores.

Buenas Practicas:

- Escribir código limpio y legible.
- Utilizar patrones de diseño adecuados.
- Implementar pruebas unitarias y de integración.
- Realizar revisiones de código.
- Documentar el código y los procesos.
- Mantener un control de versiones efectivo.
- Optimizar el rendimiento de las consultas en bases de datos.
- Implementar medidas de seguridad robustas.
- Realizar copias de seguridad regulares.
- Seguir los principios SOLID en la programación orientada a objetos.

Formas Normales(Bases de datos):

1. Primera Forma:

Nos lleva a no repetir datos en las tablas, por ejemplo si se tiene una tabla de ventas y esta repite una y otra vez los datos del cliente en cada venta es porque no se ha aplicado esta forma. Si tenemos una tabla ventas en esta solo debería figurar el id del cliente.

2. Segunda Forma:

(Se debe haber aplicado primero la primera forma) Nos habla de que cada columna de la tabla debe depender de la clave, si una columna no depende de la clave debería hacerle otra tabla.

3. Tercera Forma:

Ninguna columna debe depender de una columna que no tenga una clave y no puede haber datos derivados.