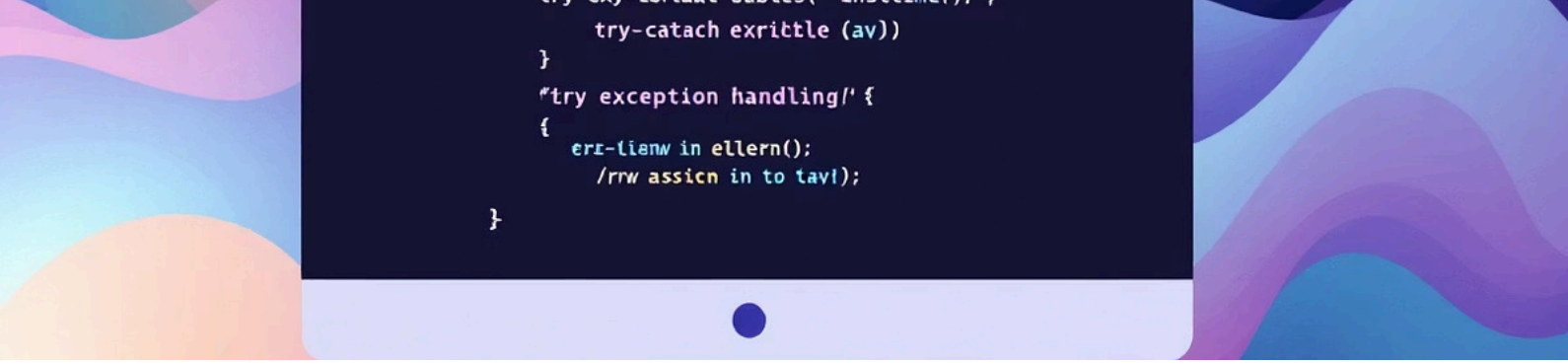
PROMETEO

Unidad 6:
Manejo de Excepciones y Colecciones

Programación

Técnico Superior de DAM / DAW



Sesión 14 – Excepciones: try-catch, throw y throws. Excepciones personalizadas

En cualquier aplicación real, por sencilla que parezca, tarde o temprano te vas a enfrentar a errores: datos que no llegan como deberían, operaciones que fallan, recursos que no existen o situaciones que no se pueden predecir. En Java, todos estos problemas se gestionan mediante un sistema de excepciones diseñado para que puedas detectar, manejar y resolver errores sin que tu programa termine de forma abrupta. Entender este mecanismo es fundamental para escribir código profesional, estable y mantenible.

Las excepciones son eventos que interrumpen el flujo normal de ejecución cuando ocurre un error o una condición inesperada. Si no se gestionan, la aplicación se detiene. Si se gestionan correctamente, puedes recuperar el control, mostrar mensajes útiles, registrar el error para diagnóstico o incluso corregir la situación. La clave no es evitar los errores (algo imposible), sino gestionarlos de manera robusta.

try-catch

El bloque try-catch es el mecanismo básico. Cualquier operación susceptible de fallar —acceder a un archivo, convertir datos, conectar a una API— se coloca dentro del bloque try. Si durante la ejecución sucede un error, Java "lanza" una excepción que salta directamente al bloque catch, donde tú decides qué hacer: informar al usuario, registrar el error, intentar una alternativa o finalizar la acción de forma controlada.

throw y throws

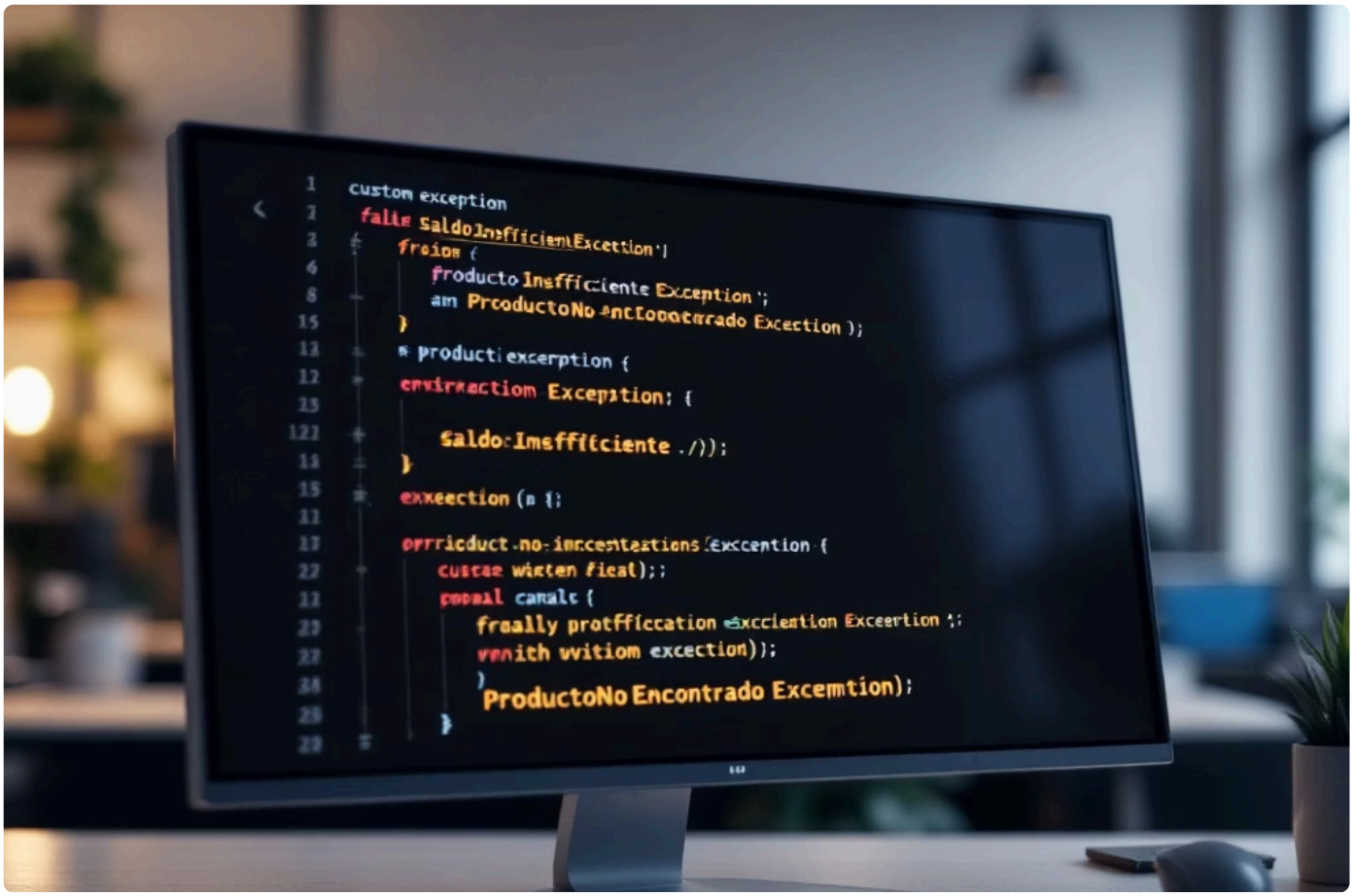
La palabra clave throw te permite lanzar excepciones manualmente cuando detectas una situación inválida que no quieres que el programa ignore. Por ejemplo, si alguien intenta retirar más dinero del que tiene un usuario, puedes lanzar una excepción antes de permitir la operación. En cambio, throws declara en la firma de un método que ese método puede lanzar determinadas excepciones, obligando a quien lo llame a gestionarlas.

Excepciones personalizadas

Las excepciones personalizadas te permiten describir condiciones específicas de tu dominio de negocio, en lugar de depender de excepciones genéricas. Una clase como SaldoInsuficienteException o ProductoNoEncontradoException aporta claridad, ayuda a otros desarrolladores a entender el contexto del error y permite manejarlo de forma diferenciada.

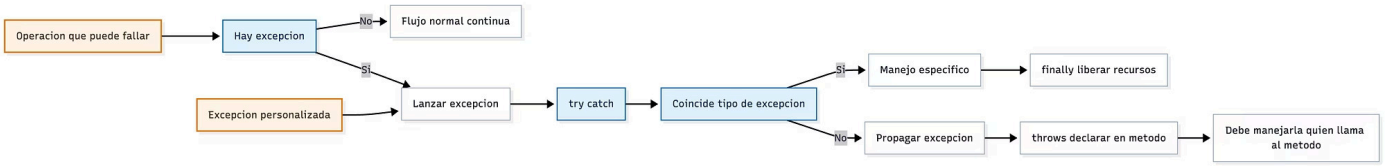
Además, puedes añadir tantos bloques catch como tipos de error esperes, lo que te permite manejar situaciones concretas de forma diferenciada. Finalmente, el bloque finally se ejecuta siempre, ocurra o no una excepción, y suele usarse para liberar recursos como conexiones o archivos abiertos.

Es una forma de crear código explícito y seguro: quien usa tu método debe decidir qué hacer si algo falla. Estas excepciones suelen extender de Exception si son verificadas o de RuntimeException si son no verificadas. Entender todos estos elementos no solo mejora la calidad de tu código; también te acerca a la forma en que funcionan sistemas profesionales a gran escala donde el manejo de errores es crítico para la disponibilidad y la experiencia del usuario.



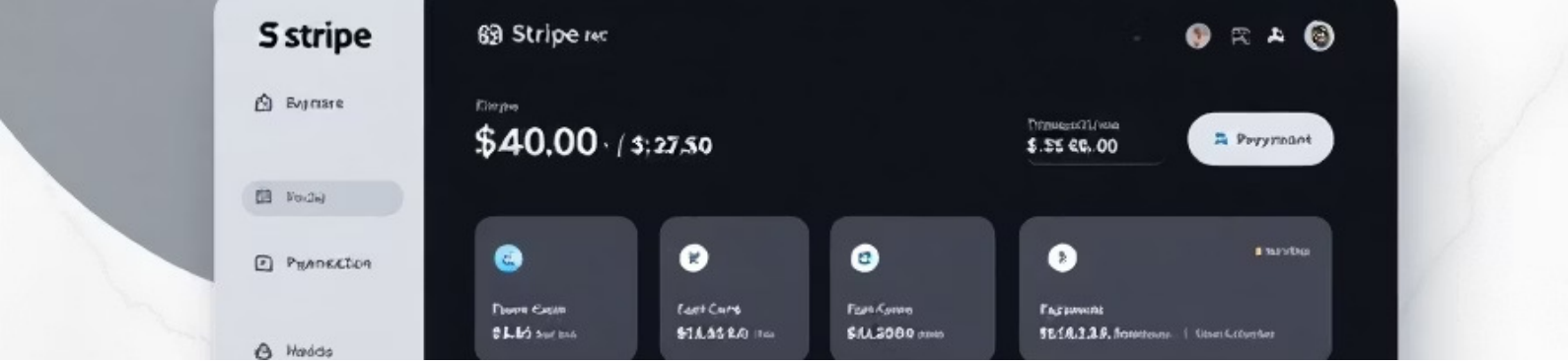
Esquema Visual

A continuación tienes un diagrama conceptual en formato Mermaid que representa el flujo del manejo de excepciones en Java, incluyendo los mecanismos try-catch, throw/throws y las excepciones personalizadas.



Descripción exhaustiva del diagrama:

- El flujo empieza en cualquier **operación que puede fallar**, representada por el nodo inicial.
- Un nodo de decisión determina si ocurre una excepción. Si no ocurre, el programa continúa su ejecución normalmente.
- Si ocurre, Java **lanza la excepción**, que entra en un bloque **try-catch**.
- Dentro del sistema de manejo, otro nodo de decisión verifica si el tipo de excepción coincide con alguno de los bloques catch definidos.
- Si existe un catch compatible, se ejecuta el manejo específico. Si no existe, la excepción se **propaga** al método que llamó al actual.
- La propagación está vinculada con la palabra clave **throws**, que declara qué excepciones debe gestionar el método llamante.
- Independientemente del resultado, el bloque **finally** libera recursos o ejecuta código obligatorio.
- En cualquier punto se pueden utilizar **excepciones personalizadas**, que entran en el mismo flujo que una excepción estándar.



Caso de Estudio – Sistema de Transacciones de Stripe

Para comprender la importancia del manejo profesional de excepciones, es útil analizar cómo lo emplea una empresa real con operaciones críticas. Stripe, la plataforma global de pagos, procesa más de 640.000 millones de dólares al año, atendiendo a millones de usuarios y comercios. En un sistema así, un mal manejo de excepciones no solo sería un fallo técnico: podría significar la pérdida de pagos, fraudes no detectados o caídas en la disponibilidad del servicio.

Excepciones especializadas

Stripe utiliza excepciones como parte esencial de su arquitectura. Por ejemplo, cuando un cliente introduce una tarjeta caducada, no se genera una excepción genérica: se lanza una TarjetaExpiradaException. Si la tarjeta es rechazada por el banco, se lanza TarjetaRechazadaException. Si un usuario intenta pagar con saldo insuficiente, aparece FondosInsuficientesException.

Try-catch anidados

Stripe utiliza try-catch anidados. En el nivel más interno se capturan errores de validación, como datos incompletos o formato inválido. En un nivel superior se capturan errores de red o de disponibilidad de sistemas externos. Finalmente, en el nivel más externo se gestionan errores críticos que pueden requerir alertas automáticas.

Declaraciones explícitas

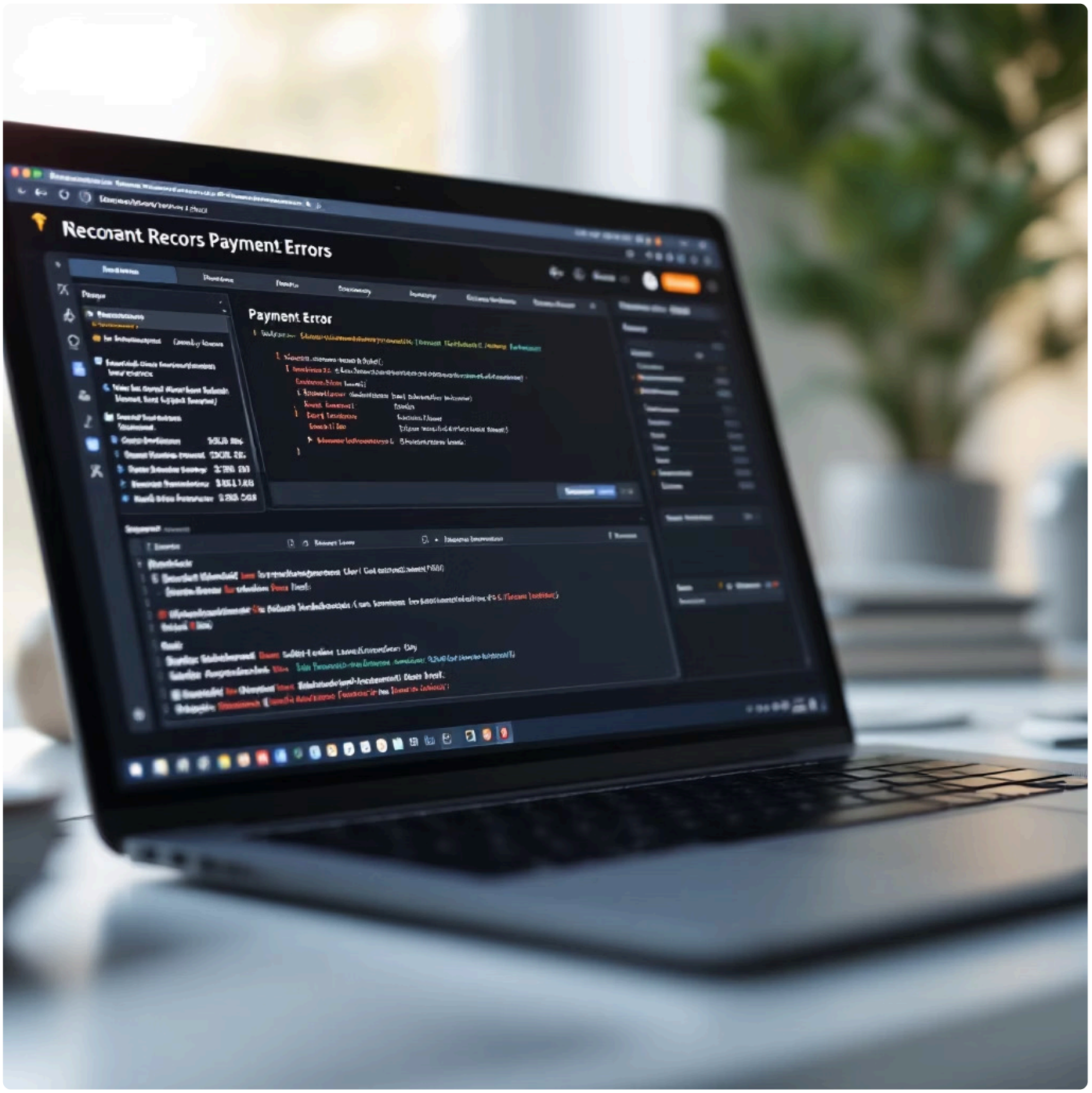
Los métodos de su SDK declaran explícitamente throws con cada tipo de excepción que pueden producir. Esto obliga a cualquier comercio que integre Stripe a manejar correctamente los distintos escenarios. Por ejemplo, un comercio debe decidir qué hacer ante una tarjeta caducada: pedir otra tarjeta, mostrar un mensaje al usuario o permitir reintento.

Registro completo

Cuando se produce un error, Stripe registra cada detalle: IP, timestamp, identificadores asociados a la transacción, mensaje técnico y una descripción pensada para el comercio. Además, genera recomendaciones automáticas como "solicitar método alternativo de pago" o "verificar datos de expiración".

Cada una contiene código de error, mensaje detallado y sugerencias para que el comercio sepa cómo reaccionar. Esta especialización permite a los desarrolladores responder de forma adaptada según la causa del fallo.

Este enfoque crea código limpio, seguro y previsible. Este manejo riguroso contribuye a su 99,99% de disponibilidad, un requisito imprescindible en el sector de pagos, donde cada milisegundo y cada excepción cuentan.



Herramientas y Consejos

Aquí tienes recomendaciones prácticas que puedes aplicar directamente en tus proyectos:

1

Usa excepciones específicas en lugar de Exception genérica

Evita escribir: catch (Exception e) porque oculta la causa real del problema y dificulta el diagnóstico. Es más profesional usar: catch (FileNotFoundException e) catch (NumberFormatException e) Además de dar claridad, te obliga a pensar en cada escenario.

2

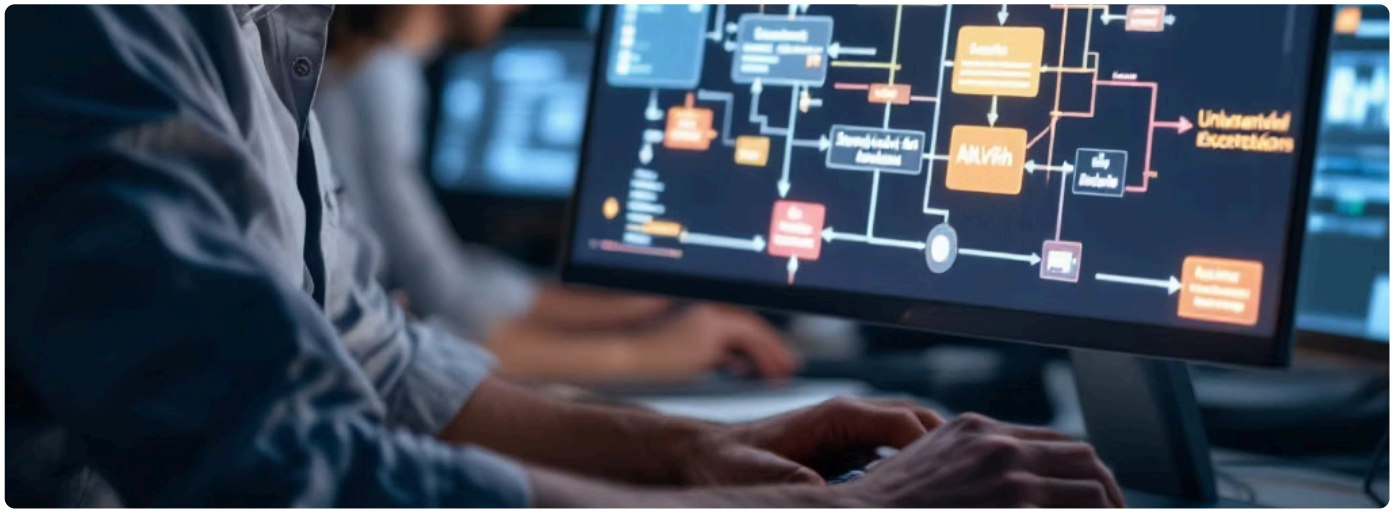
En tus excepciones personalizadas, añade información útil

Una excepción personalizada debería aportar contexto: throw new SaldoInsuficienteException("Saldo actual: " + saldo + ", requerido: " + monto); Esto acelera el diagnóstico en logs y ayuda a otros desarrolladores a entender exactamente qué ocurrió.

3

No captures excepciones que no puedes manejar

Atrapar una excepción sin actuar correctamente puede esconder errores serios: catch (IOException e) { // No hacer nada = muy mala práctica } Si no puedes corregir el error, deja que se propague.



Mitos y Realidades

❌ Mito: "Las excepciones hacen el programa más lento."

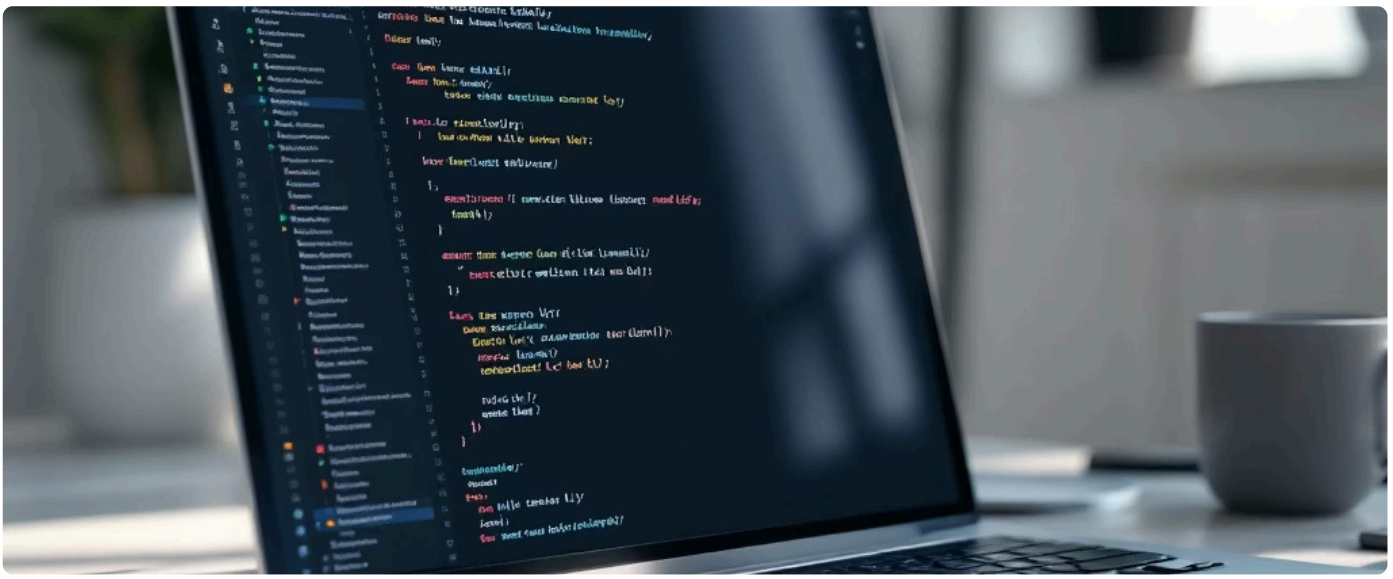
→ FALSO. El coste de usar try-catch cuando no ocurren errores es prácticamente nulo. El impacto real aparece únicamente cuando se lanza una excepción, y aun así, en la mayoría de aplicaciones es irrelevante comparado con los beneficios de un manejo robusto.

❌ Mito: "Siempre debes capturar todas las excepciones."

→ FALSO. Capturar excepciones indiscriminadamente puede ocultar problemas serios o impedir que el flujo llegue al nivel donde realmente puede gestionarse de manera eficaz. A veces dejar que la excepción se propague es la decisión más segura y profesional.

Resumen Final

- Las excepciones interrumpen el flujo normal del programa y deben manejarse correctamente.
- try-catch permite capturar errores y finally ejecuta código obligatorio.
- throw lanza excepciones manualmente; throws las declara en la firma del método.
- Las excepciones personalizadas describen errores del dominio y aportan claridad.
- Stripe demuestra cómo un manejo robusto garantiza disponibilidad y seguridad a gran escala.



Sesión 15 – Colecciones: List, Set, Map. Comparación con arrays

Cuando empiezas a trabajar con estructuras de datos en Java, el primer impulso suele ser utilizar arrays. Son sencillos, rápidos y conocidos. Pero también están limitados: su tamaño es fijo, no pueden crecer dinámicamente y no ofrecen operaciones eficientes para búsquedas, eliminaciones o inserciones frecuentes. Para aplicaciones pequeñas pueden funcionar, pero en cuanto gestionas volúmenes variables de datos o necesitas manipular listas largas de forma flexible, se quedan cortos. Ahí entran en juego las colecciones del Java Collections Framework, uno de los componentes más potentes y utilizados en programación profesional.

Una colección es una estructura de datos dinámica: puede crecer, reducirse y adaptarse al comportamiento del programa. Además, proporciona métodos avanzados para gestionar datos de forma óptima: búsquedas rápidas, eliminación eficiente, verificación de duplicados, asociaciones clave-valor, ordenación automática, entre otros. El framework se organiza en tres grandes grupos que debes dominar:

List

Mantienen el orden de inserción y permiten elementos duplicados. Son ideales cuando necesitas recorrer datos en orden o acceder por índice.

- ArrayList funciona como un array dinámico: acceso muy rápido por posición.
- LinkedList gestiona mejor las inserciones y eliminaciones frecuentes en el medio de la lista.
- Vector, aunque similar a ArrayList, está sincronizado, lo que lo vuelve menos eficiente en la mayoría de casos actuales.

Set

No admiten duplicados. Son perfectos cuando la prioridad es garantizar unicidad.

- HashSet es muy rápido para añadir y buscar elementos, aunque no mantiene orden.
- TreeSet ordena automáticamente los elementos.
- LinkedHashSet combina unicidad con preservación del orden de inserción.

Map

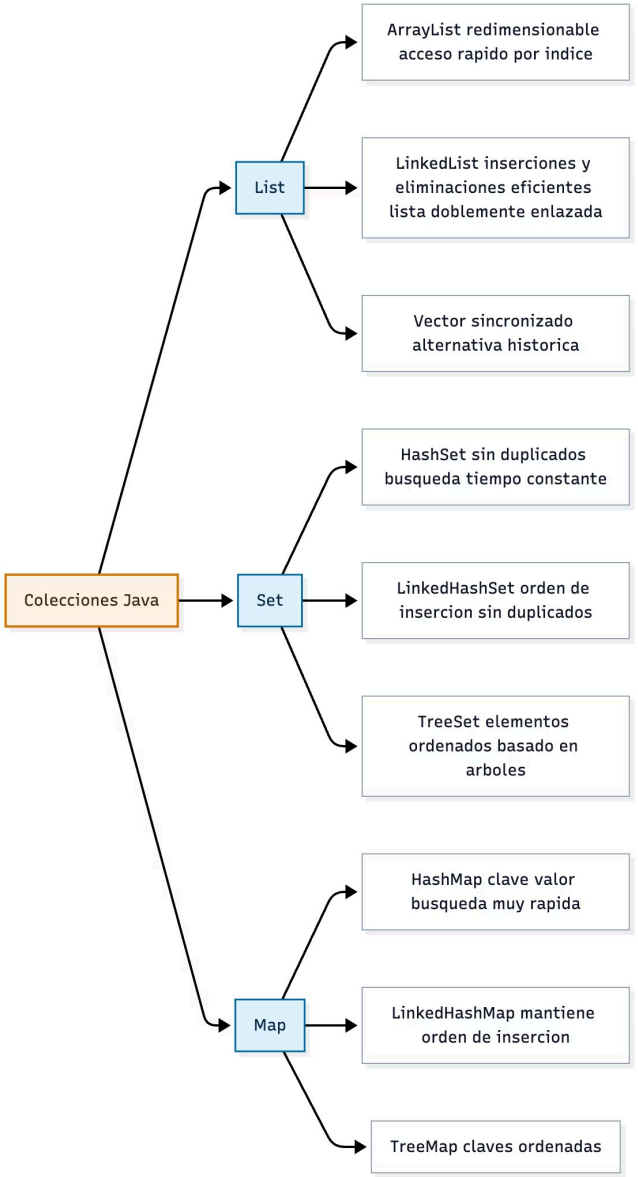
Asocian claves únicas con valores, igual que un diccionario.

- HashMap es el más rápido para búsquedas por clave.
- TreeMap mantiene las claves ordenadas.
- LinkedHashMap conserva el orden de inserción y es muy utilizado en sistemas de caché.

La diferencia clave respecto a los arrays es la flexibilidad. Un array requiere que decidas su tamaño al crearlo. Una vez fijado, no puedes ampliarlo sin crear otro array y copiar los elementos. Con una colección, ese crecimiento ocurre automáticamente y de forma eficiente. En proyectos profesionales —como motores de recomendaciones, sistemas bancarios, plataformas de comercio electrónico o aplicaciones web con miles de usuarios concurrentes — el uso adecuado de listas, conjuntos y mapas es esencial para obtener un rendimiento óptimo y un código limpio, mantenible y escalable. Lo importante no es memorizar qué hace cada clase, sino entender qué problema resuelve cada tipo de colección para elegir la estructura adecuada en cada situación.

Esquema Visual

A continuación se muestra un esquema conceptual detallado del Java Collections Framework y la relación entre List, Set y Map.



Descripción detallada del esquema

- ### Nodo principal

  El nodo principal COLECCIONES JAVA representa el conjunto completo del framework.

- ### Tres ramas principales

  De él salen tres ramas principales: List, que conserva orden y permite duplicados. Set, que evita duplicados y puede mantener o no el orden. Map, que maneja asociaciones clave-valor.

- ### Implementaciones específicas

  Cada rama se divide en implementaciones típicas con características específicas: ArrayList, eficiente para acceso por índice. LinkedList, ideal para inserciones frecuentes. HashSet, para unicidad y velocidad. TreeSet, para mantener datos ordenados. HashMap, la estructura clave-valor más rápida. LinkedHashMap, ampliamente usada en sistemas de caché.

- ### Organización visual

  Este esquema permite visualizar cómo se organiza el framework y qué aporta cada implementación.

Caso de Estudio – Sistema de Recomendaciones de Amazon

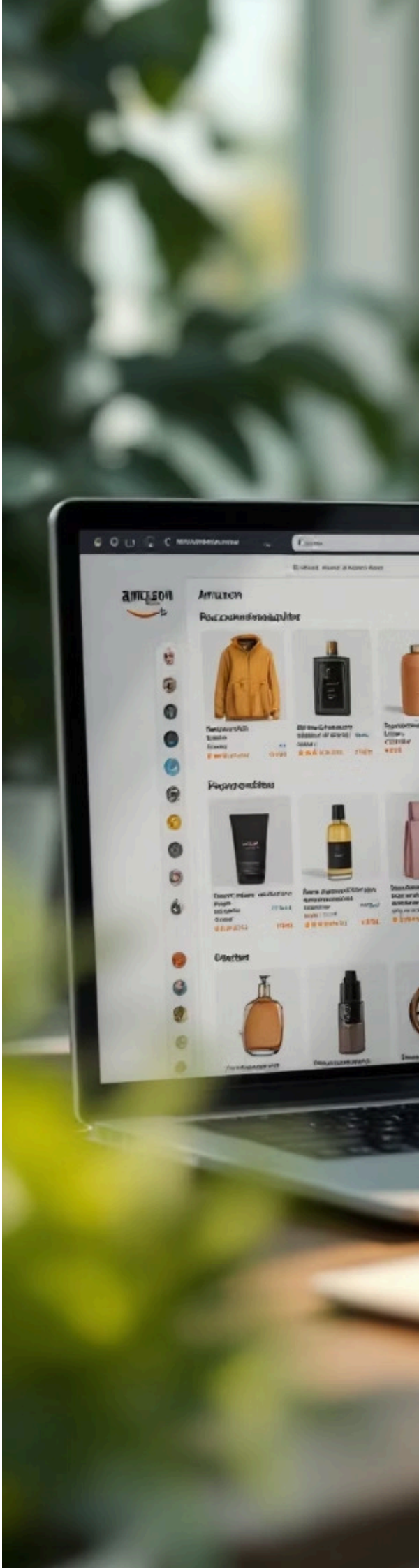
Amazon gestiona uno de los motores de recomendación más grandes del mundo. Para personalizar la experiencia de más de 200 millones de usuarios activos y manejar un catálogo de 300 millones de productos, necesita estructuras de datos rápidas, flexibles y capaces de adaptarse a operaciones intensivas. El uso estratégico de colecciones es uno de los pilares del sistema.

List para historial de compras

Cada usuario tiene un historial ordenado: el primer producto que compró, el siguiente, y así sucesivamente. El orden importa para entender patrones temporales como: compras recurrentes, estacionalidad, frecuencia de adquisición. Un ArrayList resulta ideal porque permite recorrer el historial rápidamente y acceder por índice sin penalización.

Set para categorías únicas

A partir del historial, Amazon extrae todas las categorías de productos en las que el usuario ha mostrado interés: "Deportes", "Electrónica", "Hogar", "Libros". Si usara una lista normal, podrían aparecer duplicadas docenas de veces. Un HashSet elimina duplicados automáticamente, manteniendo solo categorías únicas.



Map<String, Integer> para frecuencia de búsquedas

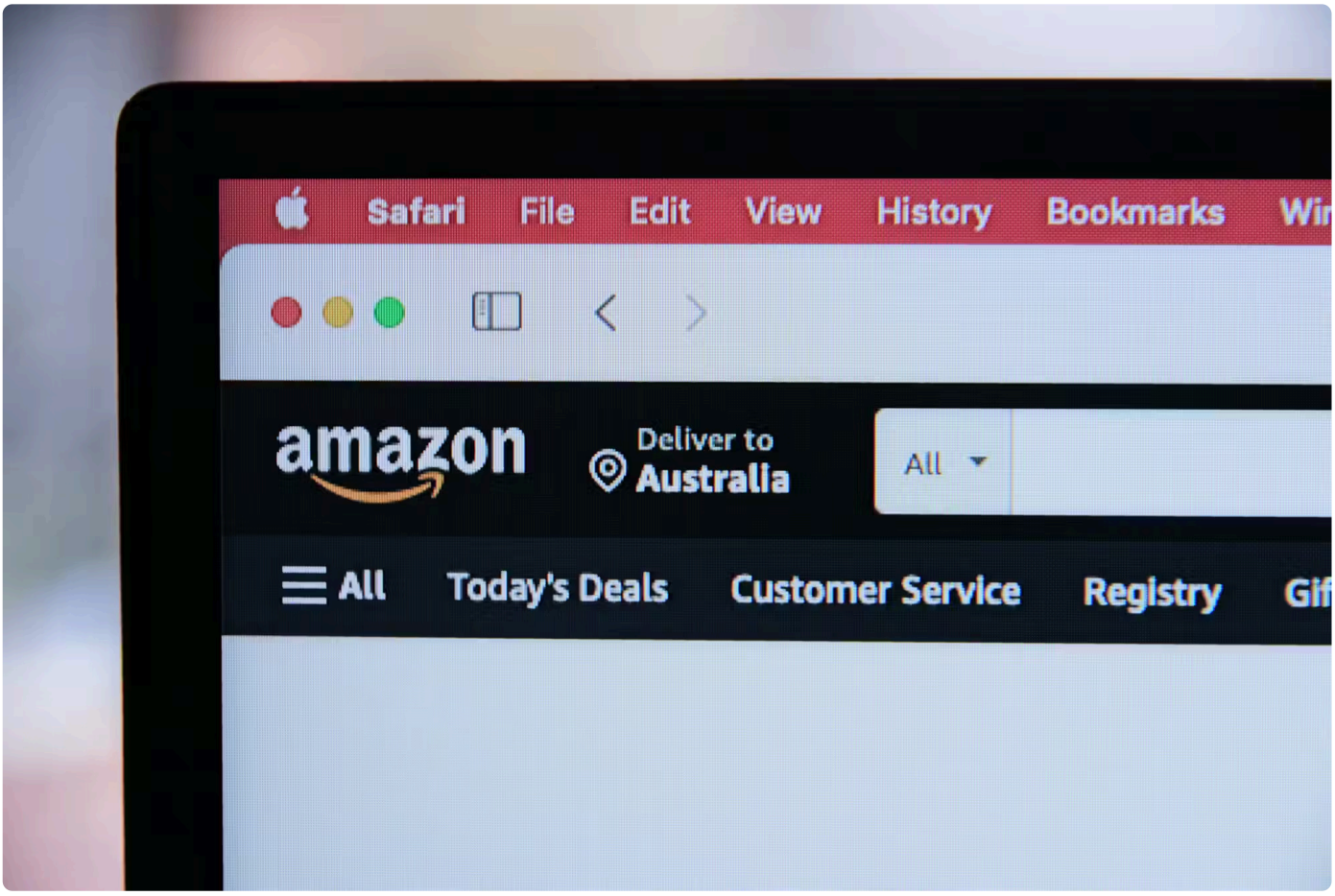
El sistema registra cuántas veces un usuario ha buscado un término específico: "auriculares" → 12, "zapatillas running" → 5, "ratón inalámbrico" → 4. Un HashMap permite almacenar cada término y su frecuencia con búsquedas O(1). Esto es esencial para priorizar recomendaciones relevantes.

LinkedHashMap para mantener el orden de recomendación

Tras combinar historial, categorías y frecuencia, el algoritmo genera una lista ordenada por relevancia. Esa lista se almacena en un LinkedHashMap, que: mantiene el orden de inserción, permite búsquedas rápidas, se adapta perfectamente a un sistema de caché.

Esto reduce drásticamente el volumen de datos y mejora los tiempos de clasificación.

El resultado es un motor capaz de generar recomendaciones personalizadas en milisegundos para millones de usuarios simultáneos. El uso eficiente de colecciones es una de las claves de su rendimiento y escalabilidad.



Herramientas y Consejos

Elige la colección según el patrón de uso

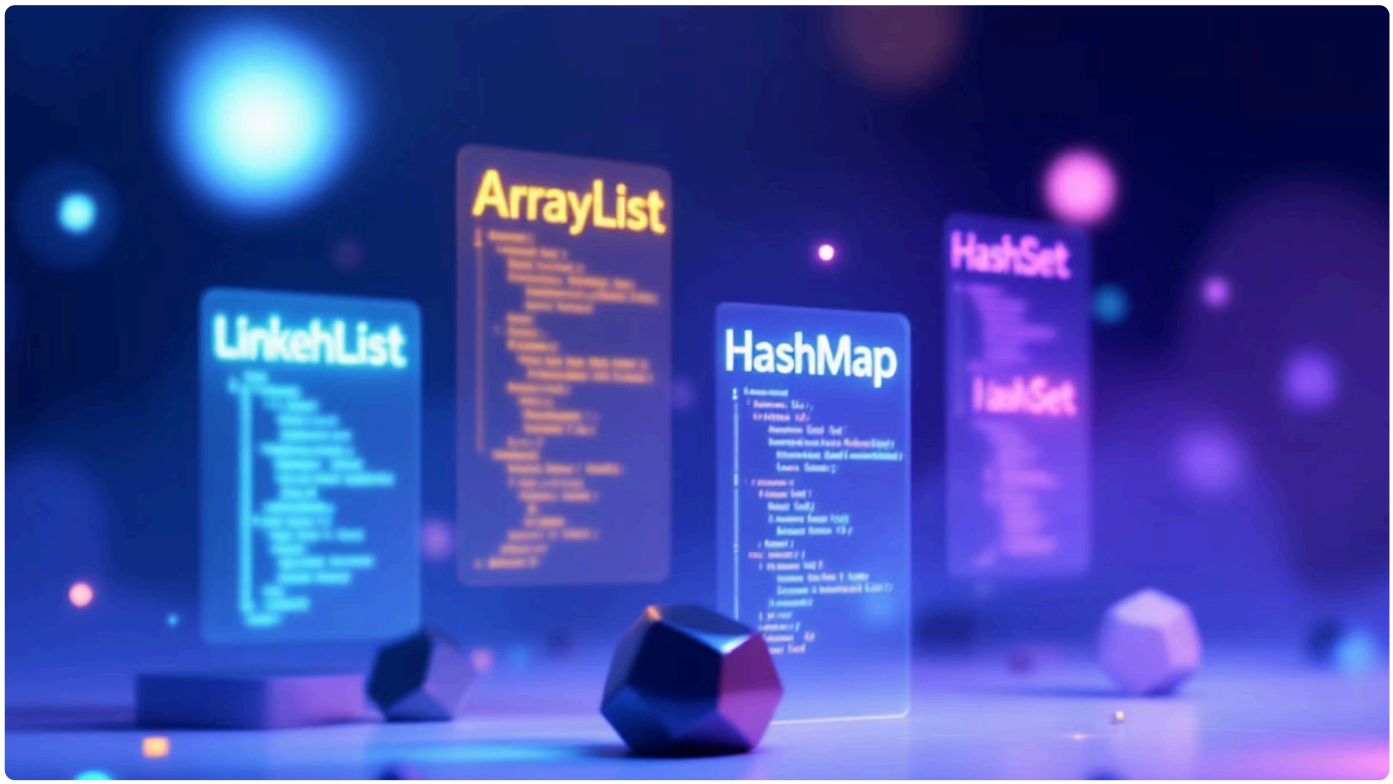
ArrayList → acceso rápido por índice o recorridos frecuentes. LinkedList → inserciones o eliminaciones en el medio de la lista. HashMap → búsquedas rápidas clave-valor. HashSet → garantizar que no existan duplicados. Elegir mal la estructura puede multiplicar los tiempos de ejecución en aplicaciones reales.

Inicializa colecciones con capacidad estimada

Cuando sabes cuántos elementos vas a manejar, mejora el rendimiento hacer: List<String> lista = new ArrayList<>(1000); Evita redimensionamientos internos costosos.

Declara usando interfaces, no implementaciones

Siempre escribe: List<String> nombres = new ArrayList<>(); en lugar de: ArrayList<String> nombres = new ArrayList<>(); Esto te permite cambiar la implementación sin afectar al código que depende de ella.



Mitos y Realidades

❌ Mito: "ArrayList siempre es mejor que LinkedList."

→ FALSO. ArrayList es excelente para acceso aleatorio, pero LinkedList supera en inserciones y eliminaciones frecuentes, especialmente en listas largas.

❌ Mito: "Las colecciones siempre consumen más memoria que los arrays."

→ FALSO. Aunque las colecciones tienen alguna sobrecarga, para datos dinámicos suelen ser más eficientes al evitar redimensionamientos manuales, copias y operaciones repetitivas sobre arrays.

Resumen Final

- Las colecciones son estructuras dinámicas que superan las limitaciones de los arrays.
- List mantiene orden y permite duplicados; Set garantiza unicidad; Map gestiona pares clave-valor.
- Amazon usa colecciones para procesar historiales, categorías y recomendaciones con eficiencia masiva.
- Elegir la implementación adecuada es clave para rendimiento y escalabilidad.