**¿QUÉ ES LA PROGRAMACIÓN FUNCIONAL?**

En esencia, es un paradigma de programación, es decir, un estilo de programar que se centra en la resolución de problemas de una manera específica. Aunque muchos piensan en ella como un campo reservado para expertos en cálculos complejos, lo cierto es que es accesible, incluso en lenguajes orientados a objetos como Java.

**CÓMO INFLUYE LA PROGRAMACIÓN FUNCIONAL EN JAVA?**

Java, tradicionalmente un lenguaje orientado a objetos, ha incorporado características de programación funcional desde su versión 8. Esto ofrece a los desarrolladores la posibilidad de aplicar este paradigma sin cambiar de lenguaje.

**FUNCIONES**

Son bloques de código que no necesitan pertenecer a una clase.

**INMUTABILIDAD**

Se trabaja con datos que no cambian su estado, lo cual facilita la concurrencia.

**CUALES SON LOS BENEFICIOS DE LA PROGRAMACIÓN FUNCIONAL?**

Este enfoque tiene varios beneficios:

Legibilidad: Las funciones se centran en qué resolver, no en cómo hacerlo, facilitando la comprensión del código.

Pruebas: Es más sencillo testear funciones individuales que sistemas completos.

Concurrencia: Permite liberar múltiples procesos simultáneamente de manera más sencilla.

Comportamientos claros: Con funciones simples, el comportamiento del sistema se vuelve más definido.

Gestión de estados: Minimiza el manejo de estados al enfocarse más en las funciones y los datos.

**¿QUE ES UNA FUNCIÓN EN PROGRAMACIÓN FUNCIONAL?**

En la programación funcional, las funciones son pieza clave y fundamental. Pero, ¿sabemos realmente qué es una función en este contexto? Una función es un tipo de dato que opera sobre otro dato; generalmente expresado como f(x) para denotar que una entrada x genera una salida y. Este concepto, que probablemente ya hayas encontrado en matemáticas, se refiere a la relación en la que por cada x, se genera un y consistente.

**¿CÓMO FUNCIONA UNA FUNCIÓN?**

Las funciones pueden describirse como una serie de pasos detallados que se ejecutan secuencialmente, y que pueden o no devolver un resultado. En el caso de que no se devuelva un valor, se interpretará como un dato vacío. Además, las funciones pueden definirse, almacenarse y declararse según sea necesario.

**QUÉ SON LAS FUNCIONES RECURSIVAS?**

El concepto de funciones en programación funcional va más allá. Podemos definir funciones en términos de otras funciones, lo cual es esencial en la programación recursiva. Una función recursiva es aquella que se llama a sí misma dentro de su propio cuerpo.

**¿PUEDEN LAS FUNCIONES TOMAR OTRAS FUNCIONES COMO PARÁMETROS?**

En la programación funcional, es común que las funciones reciban otras funciones como parámetros, lo que permite una gran flexibilidad y reutilización de código.

**CONSEJOS PRACTICOS**

A medida que avances en el curso de programación funcional en Java, considera los siguientes aspectos:

**Definición clara y concisa de funciones:** Asegúrate de que tus funciones sean específicas y cumplan una tarea precisa para facilitar su comprensión y mantenimiento.

**Uso de funciones recursivas:** Aprovecha el poder de las funciones recursivas para resolver problemas que presentan una estructura repetitiva.

**Modularidad y reutilización:** Diseña funciones que se puedan combinar, maximizando la reutilización de código y reduciendo la redundancia.

**CUALES SON LOS USOS PRÁCTICOS DE LAS FUNCIONES EN EL AIRE?**

Las funciones "en el aire" o in situ, refieren a aquellas definiciones de funciones que no se asignan a una variable permanente y pueden utilizarse inmediatamente donde se necesiten. Algunos de sus usos prácticos incluyen:

**Configuración de conexiones**: Puedes definir funciones que configuren conexiones a bases de datos o APIs a medida.

**Procesamiento de datos:** Ante una consulta web o datos recibidos de una base de datos, puedes utilizarlas para manipular y tratar con esos datos de forma eficiente.

**Lógica aislada:** Al igual que los strings se usan exprofeso en ciertas partes, estas funciones también se utilizan donde son necesarias específicamente.

Al ocupar funciones como ciudadanos de primera clase, facilitamos la escritura de código orientado a resultados, promoviendo estructuras más dinámicas y modulares que amplían el alcance y flexibilidad.

**COMO SE REPRESENTA UNA FUNCIÓN COMO CIUDADANO DE PRIMERA CLASE?**

Al considerar las funciones como ciudadanos de primera clase, obtenemos varias ventajas y utilidades en nuestro programa:

**Declaración y asignación:** Una función puede declararse como un tipo de variable, permitiendo asignarla y manipularla como cualquier otro tipo de dato.

**Parámetros y retornos:** Puedes pasar funciones como parámetros a otras funciones o devolverlas como resultado de una ejecución. Esto abre la puerta a un alto grado de reutilización y flexibilidad.

**Usos específicos:** Las funciones pueden declararse en el momento, sobre la marcha, para casos específicos, como procesar datos de una consulta web sin necesidad de utilizar la lógica en otro lugar.

Esto último es especialmente útil para mantener el código limpio y modular, aislando las partes donde se necesita lógica concreta para transformaciones o creaciones de datos.

**¿QUÉ ES UNA FUNCIÓN EN PROGRAMACIÓN Y SU IMPORTANCIA?**

En el mundo de la programación, hablar de funciones como ciudadanos de primera clase es entrar a un nivel de entendimiento más profundo sobre las posibilidades que ofrecen los lenguajes modernos. Un ciudadano de primera clase implica que las funciones son reconocidas de la misma manera que otros tipos de datos, como los números enteros o los strings. Esta capacidad transforma la forma en que los desarrolladores manejan y utilizan funciones en sus códigos, dado que ahora podemos definir, almacenar, y manipular funciones de manera más flexible. Veamos cómo funciona este concepto y su aplicación práctica.

**¿QUÉ ES UNA FUNCIÓN PURA?**

Las funciones puras, también conocidas como "pure functions" en inglés, son un concepto crucial en el mundo de la programación funcional. Estas funciones tienen una característica esencial: siempre producen el mismo resultado para los mismos parámetros de entrada, independientemente del contexto o del entorno en el que se ejecuten. Imagina una función simple que suma dos números; si siempre sumas 5 y 3, el resultado será siempre 8. No importa si la ejecutas en una computadora local o en un servidor remoto, o si es martes o jueves; el resultado es predecible y constante.

**PORQUE SON IMPORTANTES LAS FUNCIONES PURAS?**

Las funciones puras presentan varias ventajas significativas en el desarrollo de software:

**Determinismo**: Puedes predecir fácilmente el resultado al saber los parámetros de entrada. Esto es invaluable cuando escribes pruebas para tu código, ya que permite crear casos de prueba consistentes y confiables.

**Ausencia de efectos secundarios:** Las funciones puras no alteran el estado del sistema. Esto significa que no cambian una base de datos, no crean archivos ni modifican otras partes del software. Su aislamiento del sistema garantiza que no afecten a otros componentes.

**Facilidad de prueba y mantenimiento:** Al no depender de factores externos ni modificar el estado del sistema, resulta más sencillo probar y mantener el código en proyectos complejos.

**CÓMO IDENTIFICAR UNA FUNCIÓN PURA?**

La identificación de una función pura puede parecer sencilla, pero requiere atención a ciertos detalles:

Resultados Consistentes: La función debe retornar el mismo resultado siempre que se le pase el mismo conjunto de argumentos. Por ejemplo, una función que cálcula potencias siempre devolverá el mismo resultado para el mismo exponente.

Sin estados compartidos: Una función pura no debe depender de variables globales o datos que puedan cambiar durante la ejecución del programa. Esto asegura que su comportamiento se mantenga constante.

Sin efectos secundarios: Como ya mencionamos, una función pura no altera el estado del mundo externo. Esto significa que no debe interactuar directamente con el sistema de archivos, bases de datos, o interfaces de red.

**¿QUÉ SON LAS FUNCIONES IMPURAS?**

La identificación de una función pura puede parecer sencilla, pero requiere atención a ciertos detalles:

**Resultados Consistentes:** La función debe retornar el mismo resultado siempre que se le pase el mismo conjunto de argumentos. Por ejemplo, una función que cálcula potencias siempre devolverá el mismo resultado para el mismo exponente.

**Sin estados compartidos:** Una función pura no debe depender de variables globales o datos que puedan cambiar durante la ejecución del programa. Esto asegura que su comportamiento se mantenga constante.

**Sin efectos secundarios:** Como ya mencionamos, una función pura no altera el estado del mundo externo. Esto significa que no debe interactuar directamente con el sistema de archivos, bases de datos, o interfaces de red.

**INTERACCIÓN ENTRE FUNCIONES PURAS E IMPURAS**

Existen reglas claras sobre cómo deben interactuar estas funciones entre sí:

Función pura invocando a otra pura: Esto es perfectamente válido y mantiene la previsibilidad y el determinismo de la ejecución.

Función pura invocando a una impura: Esto es algo que debe evitarse, ya que rompe la pureza de la función y su capacidad de generar resultados predecibles.

Comprender estas distinciones ayuda a los desarrolladores a crear aplicaciones más robustas y fáciles de mantener, promoviendo un diseño de software más limpio y eficiente.

**¿QUÉ ES UN EFECTO SECUNDARIO EN PROGRAMACIÓN?**

Los efectos secundarios son un elemento fundamental en el desarrollo de software, y entenderlos puede cambiar radicalmente tu perspectiva sobre cómo estructurar tu código. Un efecto secundario es cualquier cambio observable desde fuera del sistema como resultado de una operación. Por ejemplo, una función que cambia el color de un elemento en pantalla resulta en un efecto observable desde el exterior y, por ende, se categoriza como un efecto secundario.

Notablemente, estos efectos secundarios son inevitables en el desarrollo de software moderno. Operaciones como la lectura de archivos, la consulta de bases de datos o la modificación de datos son procesos implícitos que requieren la interacción del software con su entorno. Por lo tanto, no se pueden eliminar completamente, pero sí se puede trabajar para minimizar su ocurrencia.

**¿POR QUÉ REDUCIR LOS EFECTOS SECUNDARIOS?**

Minimizar los efectos secundarios en tu código puede traer múltiples beneficios:

**Mejor estructura del código:** Ayuda a mantener las responsabilidades del código bien separadas y definidas.

**Más funciones puras:** Permite aumentar el uso de funciones puras, que son predecibles y más fáciles de testear.

**Código robusto:** Facilita la creación de un código más estable y escalable.

Tener un número mayor de funciones puras en tu sistema reduce los errores potenciales. Al minimizar los efectos secundarios, se incrementa la capacidad de aislación y testeo, permitiendo escribir más pruebas sobre el código para evitar errores conocidos y futuros.

**¿CÓMO GESTIONAR LOS EFECTOS SECUNDARIOS EN UN SISTEMA?**

La clave para manejar eficientemente los efectos secundarios es la estructuración adecuada de tu sistema. Una recomendación es que las funciones impuras sean únicamente puntos de entrada para la información. Una vez que los datos ingresan al sistema, deben ser manipulados y transformados por funciones puras siempre que sea posible. Este enfoque permite mantener la mayor parte del sistema libre de efectos secundarios, lo que se traduce en:

**Menos errores:** Reduce la cantidad de errores al testear funcionalidades específicas.

**Mejor estabilidad:** Favorece un sistema más estable que se adapta mejor al cambio.

**Capacidad de escalabilidad:** Facilita el escalamiento y la mejora del sistema a largo plazo.

Tener un sistema robusto y preparado para escalamiento no solo mejora la eficiencia, sino que también da lugar a una reducción de casos inesperados, mejorando así la confiabilidad y mantenibilidad del software. Continúa aprendiendo y practicando cómo diseñar sistemas cada vez más efectivos y, con paciencia, alcanzarás la maestría en el arte del código limpio y bien estructurado.

Mejor tener funciones puras, ya que puedo tener mayor cantidad de pruebas.

**QUE ES UNA FUNCIÓN DE ORDEN MAYOR?**

En el mundo de la programación funcional, las funciones juegan un papel muy importante. No solo son ciudadanos de primera clase, sino que también existen funciones de orden mayor que enriquecen aún más nuestras opciones de codificación. Pero, ¿qué características definen a una función de orden mayor? En términos sencillos, una función de orden mayor cumple con una de las siguientes características:

* Toma una o más funciones como parámetros.
* Retorna una función como resultado de su ejecución.

Estas capacidades permiten crear código más modular y reutilizable, optimizando así el flujo de trabajo del desarrollador y potenciando el uso eficiente de recursos.

**VENTAJAS**

**Pasar Comportamientos:** Puedes pasar funciones como mecanismos de filtrado para bases de datos o para la validación de datos como contraseñas. Esto te permite reutilizar lógica compleja a través de diferentes partes de tu aplicación.

**Medios de Comunicación:** Facilitan la comunicación entre diferentes componentes de un sistema. Por ejemplo, al procesar datos entrantes de manera asíncrona, puedes utilizar funciones para manejar los datos recibidos.

**Compartir Lógica:** Las funciones de orden mayor centralizan la lógica común, permitiendo compartir reglas lógicas o transformaciones entre diferentes invocaciones. Esto simplifica el mantenimiento del código, ya que cualquier cambio en la lógica base se aplica a todas las invocaciones relacionadas.

**QUE SON LAS LAMBDA?**

Las Lambdas en Java son funciones anónimas. Es decir, son esas pequeñas funciones que no tienen un nombre explícito. Esta simplicidad les otorga un papel crucial en ciertos contextos. Las funciones lambda en Java son una herramienta poderosa que permite crear funciones anónimas, facilitando el manejo y manipulación de expresiones funcionales de manera clara y concisa. Estas funciones permiten introducir bloques de código más compactos que pueden ser evaluados en tiempo de ejecución sin necesidad de almacenarse bajo un nombre específico. Aprender a utilizar las funciones lambda correctamente es esencial para cualquier desarrollador que quiera emplear programación funcional en Java. Las lambdas se implementan generalmente mediante interfaces funcionales, que son interfaces definidas con una sola función abstracta. Esto permite a Java inferir el tipo de la lambda automáticamente, reduciendo así verborrea de código.

**VENTAJAS**

Usar una Landa tiene sus ventajas precisamente por su naturaleza anónima:

**Uso único:** A menudo necesitas una función que se va a utilizar solamente una vez. Por ejemplo, podrías requerir una función para filtrar archivos por extensión en un momento específico del sistema. Reducen el código redundante y evitan la creación de clases anónimas sin necesidad.

**Simplicidad:** Un filtro para una base de datos en un solo lugar, como seleccionar alumnos que aprobaron, es un uso típico de una Landa. Estas funciones son generalmente de una línea, lo cual mejora significativamente la lectura del código. Permiten expresar comportamientos de manera compacta y clara.

**Legibilidad:** Al ser funciones simples, las Landas permiten que el código sea más accesible y fácil de entender sin necesidad de nombres o referencias más complejas.

**Facilitan operaciones en colecciones:** Trabajar con streams y otras APIs que utilizan funciones de orden superior es más sencillo y directo.

**¿CÓMO ES LA SINTAXIS?**

**Parámetros**: Se colocan dentro de paréntesis. En caso de varios parámetros, se separan por comas.

**Operador 'flecha' ->:** Separa los parámetros del cuerpo de la función.

**Cuerpo:** Puede ser una sola expresión o un bloque de sentencias delimitado por llaves si se trata de múltiples expresiones.

**CONSEJOS LAMBDA**

**Uso de tipos primitivos y tipo de retorno**: Al definir lambdas, es importante notar que trabajan sobre objetos y no sobre tipos primitivos. Es necesario utilizar clases envolventes como Integer en lugar de int.

**Claridad y legibilidad del código:** Si bien las lambdas pueden hacer el código más conciso, es importante mantener la legibilidad del código. Esto puede requerir que a veces definamos explícitamente los tipos involucrados.

Al adoptar las lambdas y otras características de la programación funcional, los desarrolladores de Java pueden escribir código más limpio y eficiente, simplificando operaciones que de otra manera serían mucho más complejas.

**INMUTABILIDAD**

La inmutabilidad de los datos es uno de los ejes principales de la programación funcional. Este concepto se refiere a la incapacidad de cambiar el valor de un dato después de su creación. Es crucial entender cómo la inmutabilidad ayuda a prevenir errores y efectos secundarios, facilitando el diseño de funciones puras.

**VENTAJAS**

**Estabilidad de los datos:** Una vez creado, un dato no cambia, lo cual es ideal para identificadores únicos, nombres, y listas estáticas.

**Funciones puras:** Al tener datos que no cambian, es más sencillo crear funciones puras que no generan efectos secundarios.

**Programación concurrente:** Facilita la ejecución en procesadores múltiples, ya que no hay necesidad de sincronizar los cambios en los datos.

**DESVENTAJAS**

**Sobrecarga de instancias**: Cada modificación requiere una nueva instancia del dato, lo que puede ser complejo en el código.

**Diseño cuidadoso:** Requiere atención especial al diseño de objetos para mantener la inmutabilidad, especialmente cuando se trata de listas y otros objetos mutables por naturaleza.

**EJEMPLOS**

**CLASE MUTABLE**

La clase MutablePerson es un ejemplo típico donde las propiedades de un objeto pueden ser alteradas accidentalmente por funciones externas:



public class MutablePerson {

private String name;

private List<String> emails;

// Constructor, getters, setters...

}



En este ejemplo, una función impura puede alterar la lista de correos debido a que se tiene un setter accesible, lo que demuestra un problema común en clases mutables.

**CLASE INMUTABLE**

Para lograr la inmutabilidad, creamos una clase ImmutablePerson modificando el acceso a sus propiedades:

:



public final class ImmutablePerson {

private final String name;

private final List<String> emails;

public ImmutablePerson(String name, List<String> emails) {

this.name = name;

this.emails = new ArrayList<>(emails); // Copia defensiva

}

public String getName() {

return name;

}

public List<String> getEmails() {

return new ArrayList<>(emails); // Copia defensiva

}

}



**Uso de final:** Prevenimos la extensión de la clase y aseguramos que las propiedades sean fijas.

**Copias defensivas**: Al devolver una nueva copia de la lista de correos, evitamos que funciones externas puedan modificarla.

**¿QUÉ ES UN PREDICADO EN PROGRAMACIÓN FUNCIONAL?**

En programación funcional, un predicado es una función que trabaja sobre un tipo de dato pero genera un valor booleano, determinando si algo es "verdadero" o "válido". Java ha incorporado la interfaz Predicate que simplifica la creación de este tipo de funcionalidades.

**VENTAJAS**

Al adoptar predicados, puedes:

* Mejorar la legibilidad del código.
* Simplificar validaciones y filtrados de datos.
* Mantener un código más limpio y fácil de mantener.
* Incrementar la eficiencia en la ejecución mediante el uso de funciones lambda.

Los predicados, junto con otras herramientas de programación funcional, proporcionan un enfoque más declarativo y efectivo para manejar operaciones complejas.

**CONSUMER (CONSUMIDOR)**

Un Consumer es una interfaz genérica que se utiliza para realizar operaciones sobre un objeto de tipo específico. No retorna ningún valor, sino que toma un objeto, lo consume y realiza ciertas acciones sobre él.

**SUPPLIER (PROVEEDOR)**

Por otro lado, un Supplier es una interfaz genérica que se encarga de generar datos de un cierto tipo. A diferencia del Consumer, el Supplier no recibe argumentos y se utiliza principalmente para generar y proporcionar datos cuando sea necesario.

**PORQUE SON ÚTILES ESTAS INTERFACES?**

Estas interfaces son extremadamente útiles en muchos escenarios:

**Eficiencia:** Permiten la creación y gestión de datos solo cuando es necesario, optimizando el uso de recursos.

**Flexibilidad**: Facilitan la creación de algoritmos modulares donde las operaciones sobre datos se pueden definir y modificar de forma independiente.

**Reutilización de código:** Los Consumers y Suppliers pueden ser usados en múltiples contextos, promoviendo la reutilización del código.

**¿QUÉ ES UN BINARYOPERATOR?**

En Java, un BinaryOperator es una funcionalidad que opera sobre un cierto tipo definido, es decir, trabaja con un solo tipo, tanto de entrada como de salida. Este concepto es parte del paquete java.util.function.

**BI-FUNCTIONS**

Las BiFunction son fundamentales cuando necesitamos trabajar con más de un parámetro. A diferencia de las funciones simples que manejan un solo tipo de datos, las BiFunction pueden recibir dos tipos de datos de entrada y generar un tercer tipo como resultado. Además de las BiFunction tradicionales, podemos encontrar variantes que permiten trabajar eficientemente con un solo tipo de dato, como el BinaryOperator antes mencionado. La BinaryOperator se simplifica si tanto los datos de entrada como el resultado son del mismo tipo.

**¿QUE ES UNA INTERFAZ SAM EN JAVA?**

En Java, una interfaz SAM (Single Abstract Method) es aquella que contiene un único método abstracto definido. Este concepto es crucial ya que permite el uso de expresiones lambda, proporcionando un enfoque más limpio y eficiente para escribir funciones anónimas.

**Definición**: Una interfaz SAM tiene un solo método abstracto que requiere implementación.

**Ejemplo**: Si definimos una interfaz funcional con un único método, esta puede ser utilizada como una función.

Java introdujo en la versión 8, la anotación @FunctionalInterface para mostrar que una interfaz es funcional. Sin embargo, si añadimos más métodos abstractos a la interfaz, dejaría de ser funcional, y el compilador nos arrojaría un error.

**¿CÓMO TRANSFORMAR ENTEROS EN FECHAS CON LAMBDA?**

Supongamos que necesitamos convertir tres números enteros en una fecha, podemos utilizar nuestra TriFunction para esta tarea.

TriFunction<Integer, Integer, Integer, LocalDate> createLocalDate =

(day, month, year) -> LocalDate.parse(year + "-" + month + "-" + day, DateTimeFormatter.ofPattern("yyyy-M-d"));

**Usamos LocalDate.parse:** Este método nos permite convertir una cadena de texto en un objeto LocalDate.

**DateTimeFormatter**: Para asegurar que la cadena se interpreta correctamente, usamos un DateTimeFormatter con el patrón adecuado.

**¿CÓMO CALCULAR LA EDAD DE UNA PERSONA?**

Extendiéndonos más allá de la conversión básica, podemos calcular la edad de una persona utilizando las funciones que hemos implementado. Imaginemos que queremos saber cuántos años han pasado desde una fecha específica hasta hoy.

TriFunction<Integer, Integer, Integer, Integer> calculateAge =

(day, month, year) -> {

LocalDate birthDate = createLocalDate.apply(day, month, year);

return Period.between(birthDate, LocalDate.now()).getYears();

};

Al calcular la edad, es fundamental asegurarnos de que los días y meses menores a 10 estén correctamente formateados para evitar errores en la conversión.

Para manejar adecuadamente los días y meses con un solo dígito, añadimos una función que añade un cero a la izquierda cuando es necesario.

Function<Integer, String> addLeadingZero = x -> (x < 10 ? "0" + x : x.toString());

Podemos utilizar esta función dentro de nuestras operaciones con fechas para asegurar el formato correcto:

String dayFormatted = addLeadingZero.apply(day);

String monthFormatted = addLeadingZero.apply(month);

**¿CÓMO REFERENCIAR MÉTODOS EN JAVA?**

Supongamos que tienes una lista de elementos y deseas aplicarles una operación funcional sin necesidad de redefinir o crear nuevos métodos desde cero. Aquí es donde las referencias a métodos en Java resultan muy útiles. Este operador (::)) te permite utilizar métodos existentes al hacer referencia a ellos.

**¿POR QUÉ USAR REFERENCIAS EN LUGAR DE CREADORES DE FUNCIONES?**

Es normal preguntarse por qué no simplemente crear nuevas instancias de las funciones necesarias. La razón principal es la eficiencia y la claridad del código. Usar referencias a métodos:

* Reduce el código redundante.
* Facilita la lectura y mantenimiento al reutilizar funciones existentes.
* Optimiza el rendimiento al evitar la creación de objetos innecesarios.

**¿CUÁNDO SON ÚTILES LAS REFERENCIAS A MÉTODOS?**

Las referencias a métodos son particularmente útiles cuando:

* Tienes métodos estáticos o de instancia ya definidos que coinciden con la firma de la función que necesitas.
* Deseas mantener el código preciso y claro, utilizando funciones predefinidas que realizan una tarea específica.
* Necesitas realizar operaciones repetitivas sobre estructuras de datos, como listas o conjuntos, aplicando funciones conocidas sin redefinirlas.

Recuerda que los métodos son, de hecho, funciones ligadas a un objeto o clase, por lo que pueden ser utilizados de forma funcional siempre que cumplan con la misma cantidad y tipo de parámetros que las funciones a las cuales sustituyen. La clave está en que el método referenciado debe coincidir en definición, garantizando así que la funcionalidad se mantenga sin excepción.

**¿QUÉ ES LA INFERENCIA DE TIPOS?**

Este concepto permite que el compilador deduzca automáticamente el tipo de dato que se está utilizando dentro de una función o expresión, sin necesidad de que el programador lo especifique explícitamente. Esto no solo hace el código más limpio y legible, sino que también reduce el riesgo de errores.

**COMO FUNCIONA?**

Para entender cómo funciona, tomemos como ejemplo un lenguaje de programación que utiliza inferencia de tipos durante la compilación. El proceso comienza cuando definimos una función. Supongamos que realizamos una función que recibe un Integer y devuelve un String. Esta función, a la que llamaremos convertidor, toma el valor entero, lo multiplica por dos y luego lo convierte en un String.



public String convertidor(Integer valor) {

return Integer.toString(valor \* 2);

}

En este caso, aunque no especificamos que el parámetro es un entero dentro del método, el compilador es capaz de determinar que valor es un entero debido al contexto en el que se usa y cómo se manipula.

**¿PORQUE ES IMPORTANTE?**

La inferencia de tipos tiene múltiples beneficios en el desarrollo de software:

**Simplificación del Código:** Al no tener que declarar explícitamente los tipos, el código se vuelve más sencillo y menos verboso.

**Error de Tipos Reducidos**: Durante la compilación, el sistema puede verificar que los datos coinciden con los tipos esperados, minimizando los errores.

**Facilita la Refactorización:** Con la inferencia de tipos, modificar y refactorizar el código se hace mucho más simple y rápido, ya que el propio compilador se encarga de gran parte del trabajo de validación.

**DESVENTAJA**

Aunque la inferencia de tipos ofrece grandes beneficios, también tiene sus límites. En algunas situaciones, la falta de especificación explícita puede llevar a confusiones sobre qué tipo de datos se está manipulando, dificultando el mantenimiento en proyectos de gran envergadura. Por ello, es importante encontrar un equilibrio entre uso de inferencia y especificación clara de tipos para mantener el código legible y gestionable.

**INTERFACES**

En Java 8, la personalización de interfaces adquiere un nuevo significado, brindando más flexibilidad y control sobre las funciones que podemos crear y utilizar. Esta capacidad permite definir métodos con un cuerpo dentro de una interfaz, lo cual se lleva a cabo por medio de la palabra clave default. A lo largo de esta explicación, exploraremos cómo estructurar y utilizar estas interfaces personalizadas creando métodos y funciones únicos.

**¿CÓMO CREAR UNA INTERFAZ FUNCIONAL PERSONALIZADA?**

Crear una interfaz funcional personalizada en Java 8 es un proceso sencillo, pero poderoso. Imagina que deseamos implementar una nueva clase llamada StringFunctions. Dentro de esta clase, crearemos una interfaz funcional llamada InterfazStringOperacion. Lo crucial para esta interfaz es tener:

* Un método abstracto sin definición.
* Métodos con definición usando la palabra default.

Por ejemplo, podríamos definir un método abstracto que devuelva un entero y un método default que reciba un String como argumento. Este método default tendría un cuerpo que define su comportamiento..



public interface InterfazStringOperacion {

int obtenerNumero();

default void operacionTexto(String texto) {

int x = obtenerNumero();

while (x-- > 0) {

System.out.println(texto);

}

}

}

**¿QUÉ ES EL ENCADENAMIENTO DE LLAMADAS?**

El encadenamiento de llamadas, también conocido como "chaining", es un concepto frecuentemente utilizado en programación, aunque no es exclusivo de la programación funcional. Permite conectar la salida de una ejecución con la entrada de otra, facilitando códigos más eficientes y legibles. Este patrón es especialmente útil cuando se trabaja con funciones u objetos que devuelven una instancia de sí mismos.

**VENTAJAS**

El encadenamiento de llamadas ofrece diversas ventajas en el desarrollo de software:

**Código más limpio y legible:** Al evitar repetir el nombre del objeto constantemente, se mejora la claridad del código.

**Eficiencia**: Reduce la creación de variables innecesarias para cada resultado intermedio, ahorrando memoria y tiempo de ejecución.

**Flexibilidad:** Facilita la adición de nuevas operaciones o métodos al objeto, permitiendo que el código evolucione con menos riesgos de errores.

**¿QUÉ SON LAS FUNCIONES DE ORDEN MAYOR?**

Las funciones de orden mayor son una parte fascinante y esencial de la programación funcional. Este tipo de funciones puede tomar como parámetro otra función o devolver una función como resultado. También pueden darse ambos casos a la vez. Esto nos permite crear composiciones de funciones, donde podemos combinar múltiples funciones para realizar tareas más complejas.

**VENTAJAS**

Las funciones de orden mayor y la composición son fundamentales para escribir código más limpio y modular. Permiten:

Reutilización del código: Al desglosar lógicas complejas en pequeñas funciones, es fácil reutilizar y combinar estas funciones en diferentes contextos.

Mantenimiento y legibilidad del código: Las funciones claras y específicas hacen que el código sea más fácil de leer y mantener.

Escalabilidad: Podemos agregar o modificar funcionalidades sin reescribir todo el código.

**CLASE OPTIONAL**

Este componente no solo mejora la programación funcional sino que también previene errores comunes como el NullPointerException, al permitir manejar de forma explícita la ausencia de valores. Las clases opcionales son herramientas valiosas al manejar datos en programación, ya que permiten asegurar que los datos procesados sean seguros al dar la posibilidad de que un valor pueda o no existir. Esto es esencial para prevenir errores comunes como los null-pointer exceptions. Al implementar clases opcionales, el programador puede manejar valores ausentes de manera más elegante y segura, evitando operaciones en datos inexistentes y simplificando el flujo de trabajo.

**MANEJO**

Cuando desarrollamos, enfrentamos situaciones en las que métodos deben retornar datos como listas o strings. En Java, la práctica común es retornar una colección vacía en vez de null. Sin embargo, cuando se trata de otros tipos de datos, como enteros, los problemas se multiplican. A menudo, se optaba por retornar valores "degenerados" como 0 o -1, pero esto podría llevar a confusiones y errores.

Es aquí cuando la clase Optional entra en juego. Optional actúa como un contenedor que puede o no contener un valor. Si hay un dato presente, podemos operar sobre él; si no lo hay, podemos definir un valor por defecto. De esa manera, evitamos manipulaciones directas con datos nulos que puedan generar errores imprevistos.

**VENTAJA**

El uso de la clase Optional en Java no solo incrementa la seguridad al evitar errores de tipo NullPointerException, sino que también promueve un estilo de programación más declarativo y legible, lo que puede ser crucial en grandes proyectos donde la consistencia del manejo de valores es fundamental. Abrazar Optional es un paso hacia la adopción de programaciones más funcionales y elegantes, y te ayudará a escribir código más robusto y mantenible.

**QUE ES UN STREAM?**

Un Stream es una secuencia de elementos respaldada por fuentes de datos como listas, mapas, arreglos u otras colecciones. Permite realizar distintas operaciones de manera declarativa, enfocándose en el "qué" en lugar del "cómo". Esto se logra a través de una API que soporta operaciones como filtrado, mapeo y reducción sobre la secuencia de datos, ofreciendo un enfoque más limpio en comparación con los bucles tradicionales. Un Stream es una secuencia de elementos sobre la que se pueden realizar diversas operaciones. La principal ventaja de los Streams es que permiten realizar operaciones en cadena sobre una colección, haciendo que el código sea más claro y fluido.

**CÓMO TRABAJAR CON MÚLTIPLES DATOS USANDO STREAMS EN JAVA**

Trabajar con múltiples datos en Java a menudo plantea desafíos, especialmente cuando se trata de procesar colecciones grandes o complejas. Java 8 introduce el concepto de Stream, que ofrece una forma más efectiva de manejar estas operaciones al permitir el procesamiento de datos de manera más fluida y concisa. Usando Stream, es posible realizar operaciones avanzadas que transformen y procesen datos de maneras más complejas, como el uso de funciones map y filter para transformar y seleccionar datos.

**DESVENTAJAS DE STREAM**

Un aspecto crítico a tener en cuenta al trabajar con Stream es que solo pueden consumirse una vez. Después de operar sobre él, el Stream se considera "consumido" y no puede reutilizarse ni realizar más operaciones. Intentar hacerlo resulta en excepciones, lo cual es importante para evitar errores durante la ejecución.

**CUALES SON LOS TIPOS DE OPERACIONES EN STREAMS?**

Existen dos tipos de operaciones principales en Streams:

**Operaciones intermedias:** Generan un nuevo Stream. Como su nombre indica, son intermedias y usualmente no producen resultados inmediatos. Ejemplos incluyen filter, map, y sorted.

**Operaciones terminales:** Estas completan el procesamiento del Stream y devuelven un resultado final, como un valor o una colección. Ejemplo de operaciones terminales son forEach, collect, y reduce.

**¿CÓMO SE UTILIZA EL MÉTODO PEEK PARA DEPURACIÓN?**

Uno de los métodos particularmente útiles para la depuración en Streams es peek. Este permite observar los elementos de un Stream sin modificarlo. Es como un microscopio que muestra lo que ocurre dentro del Stream.



courses.stream()

.filter(c -> c.contains("Java"))

.peek(System.out::println)

.collect(Collectors.toList());



Aquí, peek imprime cada elemento que pasa por el filtro, permitiendo comprobar qué datos se están procesando.

**PORQUE USAR STREAMS EN LUGAR DE LISTAS?**

El uso de Streams sobre las listas tradicionales presenta varias ventajas:

**Mayor legibilidad:** Las operaciones en cadena ofrecen un flujo de código más claro y comprensible.

**Facilidad de uso:** Puedes realizar múltiples operaciones sobre los elementos sin tener que tratar cada elemento individualmente.

**Seguridad y funcionalidad:** Al trabajar con opcionales y funciones de orden superior como filter y map, puedes manejar la presencia o ausencia de datos de maneras más seguras y limpias.

Streams ofrecen la posibilidad de realizar operaciones complejas en un menor número de líneas, lo que no solo ahorra tiempo sino que también reduce la posibilidad de errores humanos.

Cuando hablamos de pasar lambdas a una operación de Stream, en realidad, estamos delegando a Java la creación de un objecto basado en una interfaz.

Estas interfaces las mencionamos en clases anteriores. Solo como repaso, listo algunas a continuación:

**Consumer<T>:** recibe un dato de tipo T y no genera ningún resultado

**Function<T,R>**: toma un dato de tipo T y genera un resultado de tipo R

**Predicate<T>:** toma un dato de tipo T y evalúa si el dato cumple una condición

**Supplier<T>:** no recibe ningún dato, pero genera un dato de tipo T cada vez que es invocado

**UnaryOperator<T>** recibe un dato de tipo T y genera un resultado de tipo T

Estas interfaces (y otras más) sirven como la base de donde generar los objetos con las lambdas que pasamos a los diferentes métodos de Stream. Cada una de ellas cumple esencialmente con recibir el tipo de dato de el Stream y generar el tipo de retorno que el método espera.

**OPERACIONES**

A estas funciones que reciben lambdas y se encargan de trabajar (operar) sobre los datos de un Stream generalmente se les conoce como Operaciones.

Existen dos tipos de operaciones: intermedias y finales.

Cada operación aplicada a un Stream hace que el Stream original ya no sea usable para más operaciones. Es importante recordar esto, pues tratar de agregar operaciones a un Stream que ya esta siendo procesado es un error muy común.

En este punto seguramente te parezcan familiares todas estas operaciones, pues vienen en forma de métodos de la interfaz Stream. Y es cierto. Aunque son métodos, se les considera operaciones, puesto que su intención es operar el Stream y, posterior a su trabajo, el Stream no puede volver a ser operado.

**COLLECTORS**

Una vez que has agregado operaciones a tu Stream de datos, lo más usual es que llegues a un punto donde ya no puedas trabajar con un Stream y necesites enviar tus datos en otro formato, por ejemplo, JSON o una List a base de datos.

Existe una interfaz única que combina todas las interfaces antes mencionadas y que tiene como única utilidad proveer de una operación para obtener todos los elementos de un Stream: Collector.

Collector<T, A, R> es una interfaz que tomará datos de tipo T del Stream, un tipo de dato mutable A, donde se iran agregando los elementos (mutable implica que podemos cambiar su contenido, como un LinkedList), y generara un resultado de tipo R.

Usando java.util.stream.Collectors podemos convertir muy sencillamente un Stream en un Set, Map, List, Collection, etc. La clase Collectors ya cuenta con métodos para generar un Collector que corresponda con el tipo de dato que tu Stream está usando. Incluso vale la pena resaltar que Collectors puede generar un ConcurrentMap que puede ser de utilidad si requieres de multiples threads.

Usar Collectors.toXXX es el proceso inverso de usar Collection.stream(). Esto hace que sea fácil generar APIs publicas que trabajen con estructuras/colecciones comunes e internamente utilizar Stream para agilizar las operaciones de nuestro lado.

Tipos de retorno

Hasta este punto, la única manera de obtener un dato que ya no sea un Stream es usando Collectors, pues la mayoría de operaciones de Stream se enfocan en operar los datos del Stream y generar un nuevo Stream con los resultados de la operación.

Sin embargo, algunas operaciones no cuentan con un retorno. Por ejemplo, forEach, que es una operación que no genera ningún dato. Para poder entender qué hace cada operación basta con plantear qué hace la operación para poder entender qué puede o no retornar.

**EJEMPLO:**

La operación de findAny trata de encontrar cualquier elemento que cumpla con la condición del Predicate que le pasamos como parámetro. Sin embargo, la operación dice que se devuelve un Optional. ¿Qué pasa cuando no encuentra ningún elemento? ¡Claro, por eso devuelve un Optional! Porque podría haber casos en que ningún elemento del Stream cumpla la condición.

**CONCLUSION**

Por ahora, hemos entendido que cada operación en un Stream consume hasta agotar el Stream. Y lo hace en un objeto no reusable. Esto implica que tenemos que decidir en nuestro código cuándo un Stream es un elemento temporal para una función o cuándo realmente una función sera la última en tocar los datos del Stream.

**¿CÓMO MANEJAR CADENAS DE TEXTO EN JAVA?**

Java es un lenguaje de programación robusto que ofrece diversas formas de manejar cadenas de texto. Las clases y métodos asociados con las cadenas nos permiten manipular y procesar texto de manera efectiva. Es crucial comprender cómo emplear estas herramientas para elaborar código eficiente y organizado.

Las cadenas de texto, o Strings, no solo son una clase, sino también una interfaz que permite la implementación de diversas operaciones. A menudo, utilizamos Strings junto con sus tipos específicos para manejar diferentes tipos de datos, como enteros o caracteres. Esta versatilidad facilita tareas como la conversión, filtrado, y validación de texto.

**CÓMO OPERAR CON CONCURRENTE STREAMS?**

Java se destaca en el manejo de concurrencia, permitiendo el procesamiento simultáneo de tareas. Utilizar parallel() en Streams permite que el procesamiento se distribuya a través de múltiples núcleos:

IntStream paraleloStream = paresStream.parallel();

Este tipo de procesamiento paralelo puede mejorar significativamente el rendimiento al procesar grandes volúmenes de datos, pues aprovecha la capacidad total del procesador.

**CUANDO UTILIZAR STREAMS PARALELOS?**

Elegir entre Streams regulares y Streams paralelos depende del volumen de datos y la importancia del orden de los mismos. Aquí algunos consejos:

**Utiliza Streams regulares** cuando trabajes con volúmenes de datos pequeños o cuando el orden de los datos sea crítico.

**Opta por Streams paralelos** al manejar cientos de miles o millones de elementos siempre que el orden no sea relevante.

Adoptando estas prácticas, no solo optimizas el rendimiento de tu aplicación, sino que también aseguras operaciones más seguras y eficientes.

**OPERACIONES TERMINALES**

Las operaciones terminales son aquellas operaciones que como resultado no generan un nuevo Stream. Su resultado puede variar según la operación. La utilidad de estas es poder generar un valor final a todas nuestras operaciones o consumir los datos finales. La razón principal para querer esto es que los datos deberán salir en algún punto de nuestro control y es con las operaciones terminales que hacemos esto.

Pensemos, por ejemplo, en un servidor web. Recibe una petición de datos, convierte la petición en un Stream<JSON>, procesa los datos usando filter o map, convierte de JSON a datos locales que sean manipulables por código Java y hace consumo de una base de datos. Todo esto mediante streams de diferentes tipos. Pero eventualmente tiene que devolver una respuesta para quien le hizo la petición.

¿Qué pasa si quien hizo la petición no esta usando Java? No podemos enviarle un objeto de tipo Stream a un código hecho en Python o en JavaScript… es ahi donde una operación final nos ayuda a convertir nuestro Stream de Java en algún tipo de dato que sea mas comprensible.

Otro ejemplo claro es si estamos creando una librería o creando código que más gente en nuestro equipo usará. Al crear nuestros métodos y clases usamos streams por aquí y lambdas por allá, pero al exponer estos métodos para uso de otros desarrolladores no podemos obligarlos a usar Stream.

Las razones son variadas. No queremos obligar y limitar a quienes usen nuestro código a trabajar con un solo tipo dato. No sabemos qué versión de Java está usando quien use nuestro código. No sabemos si Stream está disponible en su parte del código (por ejemplo, en Android no estaba disponible del todo), etc.

Es por ello que quisiéramos proveer de algo mas simple: listas, primitivos o incluso dar algún mecanismo para poder usar código externo de nuestro lado.

Las operaciones terminales más comunes que se encuentran en Stream son:

* anyMatch()
* allMatch()
* noneMatch()
* findAny()
* findFirst()
* min()
* max()
* reduce()
* count()
* toArray()
* collect()
* forEach()

Revisaremos qué hacen y qué utilidad tienen durante esta lectura.

**OPERACIONES TERMINALES DE COINCIDENCIA**

anyMatch, allMatch, noneMatch

Las operaciones anyMatch, allMatch y noneMatch sirven para determinar si en un Stream hay elementos que cumplan con un cierto Predicate. Esto puede ser una forma simple de validar los datos de un Stream. Son terminales pues las tres retornan un boolean.

**OPERACIONES TERMINALES DE BÚSQUEDA**

**findAny, findFirst**

Estas operaciones retornan un Optional<T> como resultado de buscar un elemento dentro del Stream.

La diferencia entre ambas es que findFirst retornara un Optional conteniendo el primer elemento en el Stream si el Stream tiene definida previamente una operación de ordenamiento o para encontrar elementos. De lo contrario, funcionará igual que findAny, tratando de devolver cualquier elemento presente en el Stream de forma no determinista (random)

Si el elemento encontrado es null, tendrás que lidiar con una molesta NullPointerException. Si el Stream esta vacío, el retorno es equivalente a Optional.empty().

La principal razón para usar estas operaciones es poder usar los elementos de un Stream después haber filtrado y convertido tipos de datos. Con Optional nos aseguramos que, aún si no hubiera resultados, podremos seguir trabajando sin excepciones o escribiendo condicionales para validar los datos.

**OPERACIONES TERMINALES DE REDUCCIÓN**

Esta operación existe en tres formas:

* reduce(valorInicial, BinaryOperator)
* reduce(BinaryAccumulator)
* reduce(valorInicial, BinaryFunction, BinaryOperator)

La diferencia entre los 3 tipos de invocación:

**reduce(BinaryAccumulator)**

Retorna un Optional del mismo tipo que el Stream, con un solo valor resultante de aplicar el BinaryAccumulator sobre cada elemento o Optional.empty() si el stream estaba vacío. Puede generar un NullPointerException en casos donde el resultado de BinaryAccumulator sea null.

**reduce(valorInicial, BinaryOperator)**

Retorna un valor del mismo tipo que el Stream después de aplicar BinaryOperator sobre cada elemento del Stream. En caso de un Stream vacío, el valorInicial es retornado.

**reduce(valorInicial, BinaryFunction<V, T, V>, BinaryOperator<V>)**

Genera un valor de tipo V después de aplicar BinaryFunction sobre cada elemento de tipo T en el Stream y obtener un resultado V.

Esta version de reduce usa el BinaryFunction como map + reduce. Es decir, por cada elemento en el Stream se genera un valor V basado en el valorInicial y el resultado anterior de la BinaryFunction. BinaryOperator se utiliza en streams paralelos (stream.parallel()) para determinar el valor que se debe mantener en cada iteración.

**REVISAR Y VOLVER A ESTUDIAR ESTE TEMA**

**COUNT**

Una operación sencilla: sirve para obtener cuantos elementos hay en el Stream.

La principal razón de usar esta operación es que, al aplicar filter o flatMap, nuestro Stream puede crecer o disminuir de tamaño y, tal vez, de muchas operaciones solo nos interese saber cuántos elementos quedaron presentes en el Stream. Por ejemplo, cuantos archivos se borraron o cuantos se crearon por ejemplo.

**TOARRAY**

Agrega todos los elementos del Stream a un arreglo y nos retorna dicho arreglo. La operación genera un Object[], pero es sposible hacer castings al tipo de dato del Stream.

**COLLECT**

Mencionamos la operación collect en la lectura sobre operaciones y collectors, donde mencionamos que:

Collector<T, A, R> es una interfaz que tomara datos de tipo T del Stream, un tipo de dato mutable A, donde se irán agregando los elementos (mutable implica que podemos cambiar su contenido, como un LinkedList) y generara un resultado de tipo R. Usando java.util.stream.Collectors podemos convertir sencillamente un Stream en un Set, Map, List, Collection, etc. La clase Collectors ya cuenta con métodos para generar un Collector que corresponda con el tipo de dato que tu Stream esta usando. Incluso vale la pena resaltar que Collectors puede generar un ConcurrentMap que puede ser de utilidad si requieres de multiples threads.

**FOREACH**

Tan simple y tan lindo como un clásico for. forEach es una operación que recibe un Consumer<T> y no tiene un valor de retorno (void). La principal utilidad de esta operación es dar un uso final a los elementos del Stream.

**CONCLUSIONES**

Las operaciones terminales se encargan de dar un fin y liberar el espacio usado por un Stream. Son también la manera de romper los encadenamientos de métodos entre streams y regresar a nuestro código a un punto de ejecución lineal. Como su nombre lo indica, por lo general, son la ultima operación presente cuando escribes chaining:

**OPERACIONES INTERMEDIAS**

Se le dice operación intermedia a toda operación dentro de un Stream que como resultado devuelva un nuevo Stream. Es decir, tras invocar una operación intermedia con un cierto tipo de dato, obtendremos como resultado un nuevo Stream conteniendo los datos ya modificados.

El Stream que recibe la operación intermedia pasa a ser "consumido" posterior a la invocación de la operación, quedando inutilizable para posteriores operaciones. Si decidimos usar el Stream para algún otro tipo de operaciones tendremos un IllegalStateException.

**OPERACIONES DISPONIBLES**

La interfaz Stream cuenta con un grupo de operaciones intermedias. A lo largo de esta lectura explicaremos cada una de ellas y trataremos de aproximar su funcionalidad. Cada operación tiene implementaciones distintas según la implementación de Stream, en nuestro caso, haremos solo aproximaciones de la lógica que sigue la operación.

Las operaciones que ya están definidas son:

filter(…)

map(…)

flatMap(…)

distinct(…)

limit(…)

peek(…)

skip(…)

sorted(…)

Analicemos qué hace cada una de ellas y hagamos código que se aproxime a lo que hacen internamente.

**FILTER**

filter se encarga de iterar cada elemento del Stream y evaluar con el Predicate si el elemento debe estar o no en el Stream resultante. Si nuestro Predicate es sencillo y no incluye ningún ciclo o llamadas a otras funciones que puedan tener ciclos, la complejidad del tiempo es de O(n), lo cual hace que el filtrado sea bastante rápido.

Usos comunes de filter es limpiar un Stream de datos que no cumplan un cierto criterio. Como ejemplo podrías pensar en un Stream de transacciones bancarias, mantener el Stream solo aquellas que superen un cierto monto para mandarlas a auditoria, de un grupo de calificaciones de alumnos filtrar únicamente por aquellos que aprobaron con una calificación superior a 6, de un grupo de objetos JSON conservar aquellos que tengan una propiedad en especifico, etc.

Entre mas sencilla sea la condición de filtrado, más legible sera el código. Te recomiendo que, si tienes más de una condición de filtrado, no le temas a usar varias veces filter

En Java, el método filter() es una operación intermedia que se utiliza en el marco de trabajo de Streams para filtrar elementos de una secuencia que cumplen con una condición específica. Este método se emplea dentro de una cadena de operaciones en un Stream y no modifica la secuencia original; en su lugar, devuelve un nuevo Stream con los elementos que pasan la prueba especificada en el filtro.

**MAP**

El método map() en Java es otra operación intermedia de los Streams, que se utiliza para transformar los elementos de un Stream aplicando una función a cada uno de ellos. Es muy útil cuando necesitas modificar los elementos de la secuencia antes de continuar con otras operaciones.

EJE (BD)

Stream<DatabaseID> ids = DatabaseUtils.getIds().stream();

Stream<User> users = ids.map(id -> db.getUserWithId(id));

O, puesto de otra forma, por cada DatabaseID en el Stream inicial, al aplicar map genera un User:

DatabaseID(1234) -> map(…) -> User(Sinuhe Jaime, @Sierisimo)

DatabaseID(4321) -> map(…) -> User(Diego de Granda, @degranda10)

DatabaseID(007) -> map(…) ->User(Oscar Barajas, @gndx)

Esto resulta bastante practico cuando queremos hacer alguna conversión de datos y realmente no nos interesa el dato completo (solo partes de él) o si queremos convertir a un dato complejo partiendo de un dato base.

**FLATMAP**

El método flatMap() en Java es una operación intermedia de los Streams, que también se utiliza para transformar los elementos de un Stream, pero con una diferencia clave: despliega o aplana los elementos. Es decir, cuando aplicamos flatMap(), cada elemento del Stream original puede ser transformado en un Stream de elementos, y flatMap() aplana esos Streams dentro de un solo Stream.

Este método es especialmente útil cuando cada elemento del Stream original genera una secuencia de elementos en lugar de un solo elemento.

En ocasiones no podremos evitar encontrarnos con streams del tipo Stream<List<Courses>>, donde tenemos datos con muchos datos…

Este tipo de streams es bastante común y puede pasarte por multiples motivos. Se puede tornar difícil operar el Stream inicial si queremos aplicar alguna operación a cada uno de los elementos en cada una de las listas.

Si mantener la estructura de las listas (o colecciones) no es importante para el procesamiento de los datos que contengan, entonces podemos usar flatMap para simplificar la estructura del Stream, pasándolo de Stream<List<Courses>> a Stream<Courses>.

Visto en un ejemplo más "visual":

Stream<List<Courses>> coursesLists; // Stream{List["Java", "Java 8 Functional", "Spring"], List["React", "Angular", "Vue.js"], List["Big Data", "Pandas"]}

Stream<Courses> allCourses; // Stream{ ["Java", "Java 8 Functional", "Spring", "React", "Angular", "Vue.js", "Big Data", "Pandas"]}

**DISTINCT**

Lo que hace es comparar cada elemento del Stream contra el resto usando el método equals. De esta manera, evita que el Stream contenga elementos duplicados. La operación, al ser intermedia, retorna un nuevo Stream donde los elementos son únicos. Recuerda que para garantizar esto es recomendable que sobrescribas el método equals en tus clases que representen datos. El método distinct() en Java es una operación intermedia de los Streams que se utiliza para eliminar los elementos duplicados de un Stream, de manera que los elementos que aparezcan más de una vez sean representados solo una vez en el resultado. Este método no toma ningún parámetro y devuelve un nuevo Stream con los elementos únicos del Stream original, conservando su orden de aparición.

**LIMIT**

El método limit() en Java es una operación intermedia de los Streams que se utiliza para restringir el número de elementos en un Stream a un número específico. En otras palabras, limit(n) devuelve un nuevo Stream que contiene solo los primeros n elementos del Stream original. Si el Stream original contiene menos de n elementos, el Stream resultante contendrá todos los elementos disponibles. La operación limit recibe un long que determina cuántos elementos del Stream original seran preservados. Si el número es mayor a la cantidad inicial de elementos en el Stream, básicamente, todos los elementos seguirán en el Stream. Un detalle interesante es que algunas implementaciones de Stream pueden estar ordenadas (sorted()) o explícitamente no ordenadas (unordered()), lo que puede cambiar drásticamente el resultado de la operación y el performance.

**PEEK**

peek funciona como una lupa, como un momento de observación de lo que está pasando en el Stream. Lo que hace esta operación es tomar un Consumer, pasar los datos conforme van estando presentes en el Stream y generar un nuevo Stream idéntico para poder seguir operando.

Usarlo puede ayudarnos a generar logs o registros de los datos del Stream

**SKIP**

Esta operación es contraria a limit(). Mientras limit() reduce los elementos presentes en el Stream a un número específico, skip descarta los primeros n elementos y genera un Stream con los elementos restantes en el Stream. El método skip() en Java es una operación intermedia de los Streams que se utiliza para omitir los primeros n elementos de un Stream y devolver un nuevo Stream que contiene los elementos restantes. Es útil cuando necesitas saltarte una cierta cantidad de elementos en el comienzo de un Stream y procesar el resto.

**SORTED**

El método sorted() en Java es una operación intermedia de los Streams que se utiliza para ordenar los elementos de un Stream en un orden determinado. Por defecto, sorted() ordena los elementos de menor a mayor según el orden natural de los elementos (para tipos que implementan Comparable, como Integer, String, etc.). También puedes proporcionar un Comparator para especificar un orden personalizado.

**CONCLUSIONES**

Las operaciones intermedias nos permiten tener control sobre los streams y manipular sus contenidos de manera sencilla sin preocuparnos realmente por cómo se realizan los cambios.

Recuerda que las operaciones intermedias tienen la funcionalidad de generar nuevos streams que podremos dar como resultado para que otras partes del código los puedan utilizar.

Aunque existen otras operaciones intermedias en diferentes implementaciones de Stream, las que aquí listamos están presentes en la interfaz base, por lo que entender estas operaciones te facilitara la vida en la mayoría de los usos de Stream.

**¿CÓMO CONVIERTO UN STREAM A UNA LISTA DE ENTEROS EN JAVA?**

Entender cómo manipular y transformar datos es fundamental para cualquier desarrollador en el mundo real. Muchas veces, las operaciones que realizamos no solo se imprimen en pantalla, sino que se utilizan para procesar grandes cantidades de información en servidores web, peticiones, bases de datos y mucho más. En este contexto, Java nos ofrece poderosas herramientas para trabajar con Streams y convertirlos en estructuras de datos estáticas y manejables, como listas de enteros.

Un Stream en Java nos permite procesar datos de manera sencilla y eficiente. En este caso específico, podemos transformar un Stream de enteros en una lista, lo que es especialmente útil cuando se trabaja en aplicaciones web o de escritorio.

**QUE ES GRADLE?**

Gradle es una herramienta de construcción que es fundamental para la gestión de proyectos en el desarrollo de software. Con Gradle, puedes automatizar tareas como compilar el proyecto, generar archivos para compartir o simplemente ejecutar aplicaciones. Los archivos de configuración en Gradle son cruciales para este proceso.

**QUE ES FEIGN?**

Feign es una librería Java que facilita la creación de clientes HTTP de manera declarativa. Permite definir de manera sencilla cómo deben ser las peticiones HTTP mediante anotaciones, lo que mejora la legibilidad y el mantenimiento del código. Configurar una API usando Feign en Java puede ayudarte a consumir servicios web con facilidad. Feign simplifica el manejo de peticiones HTTP mediante la creación de interfaces que definen cómo se debe realizar una solicitud a una API.

**QUE ES YARGS?**

Yargs Commander es una librería excelente para transformar los argumentos recibidos por terminal en objetos Java. Esto nos ofrece la posibilidad de manejar argumentos de manera estructurada y efectiva.

**¿QUÉ ES UNA CLASE VALIDADORA?**

Dispondremos una clase llamada ValidArgumentValidator que implementará una interfaz similar a yargs. Esta clase nos permitirá interceptar peticiones y realizar comprobaciones antes de que los valores pasen al resto del programa.

**QUE ES VALID\_INPUT\_VALIDATOR?**

Es importante evitar que los usuarios ingresen datos inválidos como espacios en blanco o caracteres no alfanuméricos. Para esto, creamos una clase valid\_input\_validator:

**Expresión regular:** Utilizamos regex para validar que solo se ingresen letras y números.

**Mensajes claros:** Informa al usuario cuando el criterio de búsqueda no sea válido.