

Fundamentos de la POO

Muy bien, la Programación Orientada a Objetos (POO) se ha convertido en un paradigma ampliamente utilizado que nos permite crear software estructurado, modular y fácilmente escalable. No está de más repetir que se basa en la idea de que el mundo real se puede modelar a través de entidades llamadas objetos, los cuales tienen propiedades (atributos) y comportamientos (métodos) que interactúan entre sí.

Estos conceptos ya los mencionamos, pero los explicaremos a profundidad.

- **Encapsulamiento**
- **Herencia**
- **Abstracción**
- **Polimorfismo:**

1. ¿Qué son las clases y objetos?

Una **clase** es una plantilla o un modelo que define las propiedades y el comportamiento de un tipo de objeto. Es una estructura que contiene variables (también conocidas como atributos) para almacenar datos y métodos para realizar operaciones sobre esos datos. Puedes pensar en una clase como un plano o una descripción de cómo se creará un objeto.

Un **objeto**, por otro lado, es una instancia de una clase. Es una entidad real que se crea en memoria y puede interactuar con otros objetos a través de mensajes. Cada **objeto** tiene su propio conjunto de datos (atributos) y puede realizar acciones (métodos) definidos en su clase. Por ejemplo, si tienes una clase llamada "**Coche**", un objeto de esa clase podría ser un coche específico con su propio color, marca, modelo, etc.

Aca ver video Henry 1 de clase 1

2. ¿Qué es el encapsulamiento?

El encapsulamiento es un principio clave de la POO que se refiere a ocultar los detalles internos de un objeto y exponer solo la funcionalidad necesaria. Esto mejora la seguridad y facilita el mantenimiento del código.

Aca ver video Henry 2 de clase 1

3. Herencia

La abstracción es la capacidad de simplificar complejidades al modelar objetos del mundo real. Aprenderemos a identificar y aplicar la abstracción en nuestro diseño de clases y objetos.

Aca ver video Henry 3 de clase 1

4. Abstracción

La **abstracción** en POO consiste en representar las características esenciales de un objeto, ocultando los detalles irrelevantes y creando una clase base para su reutilización.

Aca ver video Henry 4 de clase 1

5. Polimorfismo

El polimorfismo es un principio que permite que diferentes objetos respondan a la misma llamada de método de manera única. Esto proporciona flexibilidad y extensibilidad en nuestros programas. Veremos cómo utilizar el polimorfismo en nuestra programación orientada a objetos.

En la siguiente video clase, nos sumergimos en estos fascinantes conceptos, ampliéndolos conceptualmente y viendo su implementación de forma práctica a través de la sintaxis y código en Java:

Aca ver video Henry 5 de clase 1

¡Ahora sí! Al comprender y dominar estos conceptos, estarás en camino de aprovechar al máximo el poder de la POO y crear aplicaciones más flexibles, mantenibles y eficientes. Así que prepárate para sumergirte en el mundo de las clases y objetos en la Programación Orientada a Objetos, ¡donde la imaginación y la creatividad se combinan para crear software innovador y de calidad!

UML como herramienta para el análisis de diseño

UML (Unified Modeling Language) es un lenguaje de modelado visual utilizado en el desarrollo de software para representar y comunicar el diseño y la estructura de un sistema. Fue creado para unificar los enfoques

y estándares anteriores de modelado y se ha convertido en un estándar de facto en la industria del desarrollo de software.

UML proporciona un conjunto de notaciones gráficas que permiten a los desarrolladores, analistas y otros profesionales del software representar diferentes aspectos de un sistema de manera clara y comprensible. Algunos de los diagramas más comunes en UML incluyen:

DIAGRAMA DE CLASE

DIAGRAMA DE OBJETOS

DIAGRAMA DE CASOS DE USO

DIAGRAMA DE SECUENCIA

Muestra las clases del sistema, sus atributos, métodos y las relaciones entre ellas, como la herencia y la asociación.

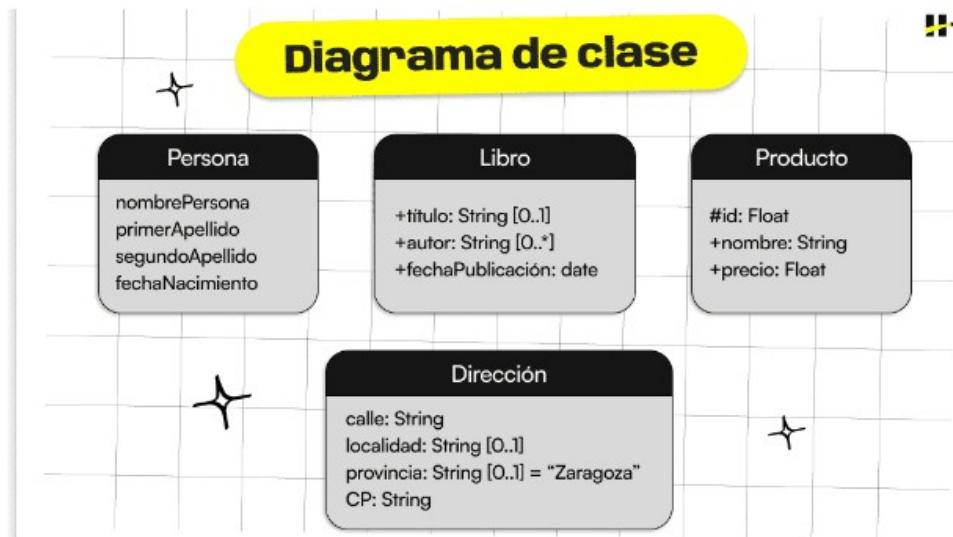


DIAGRAMA DE CLASE

DIAGRAMA DE OBJETOS

DIAGRAMA DE CASOS DE USO

DIAGRAMA DE SECUENCIA

Representa una instantánea de un sistema en un momento específico, mostrando los objetos y las relaciones entre ellos.

Automóvil: vehículo

Marca: YY
Modelo: XX
Cilindrada: 2000CC

Automóvil: vehículo

Automóvil

Vehículo

Wikipedia: proyecto

Pablo: usuario

Maria: usuario

Victor: usuario

DIAGRAMA DE CLASE	DIAGRAMA DE OBJETOS	DIAGRAMA DE CASOS DE USO	DIAGRAMA DE SECUENCIA	DIAGRAMA DE ESTADOS
-------------------	---------------------	--------------------------	-----------------------	---------------------

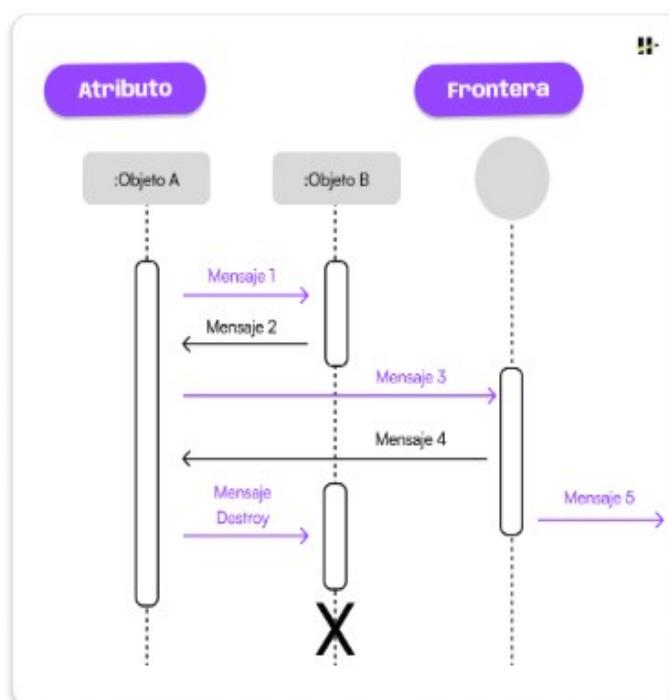
Describe las interacciones entre los actores (usuarios o sistemas externos) y el sistema, mostrando los diferentes casos de uso y cómo se relacionan.



E

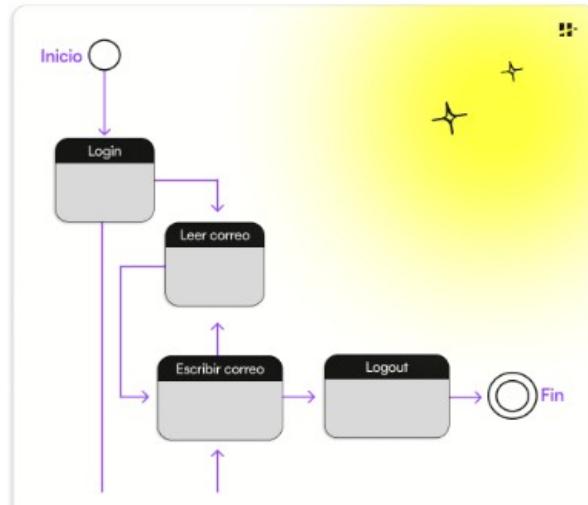
DIAGRAMA DE CLASE	DIAGRAMA DE OBJETOS	DIAGRAMA DE CASOS DE USO	DIAGRAMA DE SECUENCIA
-------------------	---------------------	--------------------------	-----------------------

Muestra la interacción entre los objetos a lo largo del tiempo, resaltando el orden de los mensajes enviados entre ellos.



DE	DIAGRAMA DE CASOS DE USO	DIAGRAMA DE SECUENCIA	DIAGRAMA DE ACTIVIDAD	DIAGRAMA DE ESTADO
----	--------------------------	-----------------------	-----------------------	--------------------

Representa los diferentes estados de un objeto y las transiciones entre ellos en respuesta a eventos.



DE	DIAGRAMA DE CASOS DE USO	DIAGRAMA DE SECUENCIA	DIAGRAMA DE ACTIVIDAD	DIAGRAMA DE ESTADO
----	--------------------------	-----------------------	-----------------------	--------------------

Ilustra el flujo de trabajo o proceso de un sistema, mostrando las actividades y las decisiones que se toman.



stos son solo algunos de los diagramas disponibles en UML. Cada uno de ellos se utiliza para enfocarse en un aspecto específico del sistema y proporciona una representación visual clara y concisa.

El uso de UML tiene varios beneficios, como

1

Comunicación efectiva: UML proporciona un lenguaje común y visualmente comprensible que facilita la comunicación entre los miembros del equipo de desarrollo, los clientes y otros interesados.

2

Diseño estructurado: UML ayuda a los desarrolladores a visualizar y comprender la estructura y las relaciones del sistema, lo que facilita la implementación y el mantenimiento.

3

Documentación: Los diagramas UML sirven como documentación del sistema, permitiendo a los desarrolladores y otros involucrados comprender y analizar el diseño y la funcionalidad del sistema.

4

Análisis y modelado: UML proporciona herramientas y técnicas para analizar y modelar sistemas complejos, ayudando a identificar problemas potenciales y mejorar el diseño antes de la implementación.

En resumen, UML es un lenguaje de modelado visual utilizado en el desarrollo de software para representar y comunicar el diseño y la estructura de un sistema. Proporciona un conjunto de diagramas gráficos que ayudan a comprender y documentar el sistema, facilitando la comunicación y el análisis durante el proceso de desarrollo de software.

Como recomendación, podemos usar alguna de las siguientes aplicaciones para el desarrollo de los diagramas:

App web: <https://app.diagrams.net/>

•Aplicación de escritorio: <https://staruml.io/download/>

Propiedades y métodos estáticos

Las propiedades o métodos estáticos se asocian con la clase en lugar de una instancia específica de la misma. Esto quiere decir que para poder utilizar dicha variable (propiedad) o un método declarados como estáticos, no es necesario realizar un new de una clase, no se necesita instanciarla; se pueden usar llamando directamente a la clase misma.

Se declaran con la palabra clave "static" y están disponibles para todas las instancias de la clase o la clase misma sin instanciar, desde el momento en que se inicia la aplicación, y mueren una vez que la aplicación se cierra.

Entonces, toda propiedad, variable o método estático sigue un flujo particular durante el ciclo de vida:

- a) Se declaran y se inicializan una vez cuando se carga la clase en memoria.
- b) Pueden ser accedidos y utilizados por cualquier instancia de la clase o directamente a través del nombre de la clase.
- c) Permanecen en memoria durante toda la ejecución del programa hasta que se finaliza.

En cuanto a las propiedades estáticas:

- Pueden ser accedidas directamente a través del nombre de la clase, incluso antes de crear una instancia de la clase.
- Se utilizan para almacenar datos que deben ser compartidos entre todas las instancias de la clase

En cuanto a los métodos estáticos:

- No pueden acceder a variables de instancia ni utilizar métodos no estáticos.
- Se utilizan para operaciones que no dependen del estado de una instancia particular.

Revisemos estos puntos a través de un pequeño ejemplo:

```
public class Contador {  
    private static int count = 0; // Propiedad estática  
    public Contador() {  
        count++; // Incrementa el contador cada vez que se crea una instancia  
    }  
    public static int getCount() {  
        return count; // Método estático para obtener el contador  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Contador c1 = new Contador();  
        Contador c2 = new Contador();  
        System.out.println("Contador: " + Contador.getCount()); // Resultado:  
        Contador: 2  
    }  
}
```

En este ejemplo, la propiedad estática count se inicializa una vez y se comparte entre todas las instancias de la clase Contador. Cada vez que se crea una instancia, se incrementa el contador compartido. Esto ilustra el ciclo de vida y la utilidad de las propiedades y métodos estáticos en Java.

Declaración y uso de una constante en Java:

Las constantes en Java son valores que no pueden ser modificados una vez que han sido asignados. Estos valores se declaran utilizando la palabra clave 'final', lo que indica que no pueden ser alterados durante la ejecución del programa.

Para declarar una constante en Java, se utiliza la siguiente sintaxis:

```
public class MiClase {  
    public static final int MI_CONSTANTE = 42;  
}
```

En este ejemplo, 'MI_CONSTANTE' es una constante de tipo entero con el valor 42.

Es interesante notar que Java no utiliza una palabra clave específica como "const" en JavaScript u otros lenguajes para definir constantes. En su lugar, combina las características static y final para lograr el mismo efecto. Esto se debe a que Java considera que el uso de constantes es una necesidad común en la programación y desea proporcionar una forma consistente y explícita de definirlas.

La combinación de static y final para crear una constante en Java tiene ventajas. La palabra clave static permite acceder a la constante a través de la clase sin necesidad de crear una instancia de la misma, lo que resulta útil en situaciones donde se necesita utilizar la constante en diferentes partes del programa. Por otro lado, la palabra clave final garantiza que el valor la constante a lo largo del programa.

Las constantes se utilizan para almacenar valores que son conocidos y no deben cambiar durante la ejecución del programa. Esto mejora la legibilidad del código y evita errores al no permitir modificaciones accidentales de estos valores.

```
public class OtraClase {  
    public static void main(String[] args) {  
        System.out.println("El valor de la constante es: " +  
MiClase.MI_CONSTANTE);  
    }  
}
```

Ventajas de Usar Constantes

- **Claridad y legibilidad del código:** al nombrar las constantes de forma descriptiva, se hace evidente su propósito en el código.
- **Mantenimiento sencillo:** si necesitas cambiar el valor, solo tienes que actualizar la constante en un lugar y esta modificación se reflejará en todas las partes del código donde se usa.
- **Evita errores:** al hacer que los valores sean inmutables, se previenen errores que podrían surgir debido a modificaciones inadvertidas.

Buenas prácticas a tener en cuenta:

- Las convenciones sugieren que los nombres de las constantes deben estar en mayúsculas y separados por guiones bajos si tienen más de una palabra para mejorar la legibilidad (por ejemplo, 'VALOR_PI').
- En Java, las constantes son estáticas por convención para indicar su inmutabilidad y acceso global.

Ejemplo integrado

```
public class Constantes {  
    public static final double VALOR_PI = 3.14159; // Declaración de una  
constante  
  
    public static void main(String[] args) {  
        System.out.println("El valor de PI es: " + VALOR_PI); // Output: El  
valor de PI es: 3.14159  
    }  
}
```

En este ejemplo, la constante 'VALOR_PI' se declara con un valor específico y se utiliza en otro lugar del programa. Al hacerlo, aseguramos que este valor se mantenga constante y sea fácilmente comprensible para otros desarrolladores.

Definición de constantes en Java mediante la palabra clave final

En el siguiente link encontrarás otro ejemplo y su explicación sobre el tema. Adicionalmente, si te interesa profundizar un poco más en el concepto, puedes encontrar un tutorial: <https://www.tutorialesprogramacionya.com/javaya/detalleconcepto.php?punto=66&codigo=144&inicio=60>

String, más que un tipo, ¡una librería!

En Java, 'String' es una clase que representa una secuencia de caracteres y nos permite trabajar con texto y manipularlo de diversas formas. Es una de las clases fundamentales y más utilizadas en el lenguaje y se encuentra en el paquete 'java.lang', lo que significa que no es necesario importarla explícitamente en nuestros programas.

Características principales

Inmutabilidad

Las instancias de la clase `String` en Java son inmutables, lo que significa que una vez creada una cadena, su contenido no puede ser modificado. Cualquier operación que parezca modificar la cadena realmente crea una nueva cadena con el resultado de la operación.

```
String str = "Hola";
str = str + ", mundo"; // Se crea una nueva cadena
```

Secuencia de Caracteres

Una cadena es una secuencia de caracteres Unicode. Esto significa que puede contener letras, números, símbolos y caracteres especiales de varios idiomas.

```
String unicodeString = "¡Hola, मनु होस, こんにちは";
```

Representación en Memoria

Las cadenas en Java se almacenan en el "String Pool", un área especial de la memoria gestionada por la JVM. Esto mejora la eficiencia y el rendimiento, ya que evita la duplicación de cadenas idénticas.

Operaciones de Manipulación

La clase `String` proporciona numerosos métodos para manipular y trabajar con cadenas, como concatenación, extracción de subcadenas, búsqueda y reemplazo de caracteres, transformación a mayúsculas o minúsculas, entre otros.

Veamos algunas operaciones básicas a través del siguiente ejemplo:

```
public class OperacionesString {
    public static void main(String[] args) {
        String str = "Hola, mundo!";
        // Longitud de la cadena
        int longitud = str.length();
        System.out.println("Longitud: " + longitud);
        // Obtener el carácter en una posición específica
        char primerCaracter = str.charAt(0);
        System.out.println("Primer carácter: " + primerCaracter);
        // Concatenación
        String concatenada = str.concat(" Es un buen día.");
        System.out.println("Concatenada: " + concatenada);
        // Subcadena
        String subcadena = str.substring(6, 11);
        System.out.println("Subcadena: " + subcadena);
        // Reemplazo
        String reemplazada = str.replace("mundo", "universo");
        System.out.println("Reemplazada: " + reemplazada);
    }
}
```

¡A tener en cuenta!

En Java, al crear objetos de la clase String, normalmente usamos el constructor para especificar el valor que queremos asignar. Sin embargo, con los objetos String, hay una forma más rápida y conveniente de crearlos, conocida como "literal". Un literal de cadena es simplemente una secuencia de caracteres encerrada entre comillas dobles (Es como estamos creando la cadena "Hola, mundo!" en el ejemplo anterior). Cuando usamos un literal para crear un objeto String, no necesitamos llamar explícitamente al

constructor de la clase. Java reconoce automáticamente que queremos crear un objeto String y asigna el valor del literal a ese objeto.

Usos Comunes de la Clase `String`

- **Entrada/Salida de Datos:** La lectura y escritura de datos desde y hacia el usuario a menudo se hace utilizando cadenas.
- **Manipulación de Texto:** Para trabajar con texto, esencial en cualquier programa, desde procesamiento de archivos hasta presentación de información al usuario.
- **Comunicación en Red:** Las operaciones de red a menudo implican el uso de cadenas para enviar y recibir datos.

Para conocer aún más sobre las funcionalidades que los Strings tienen incorporados, te dejamos este link de referencia: <http://www.itlp.edu.mx/web/java/Tutorial%20de%20Java/Cap3/string.html>

Interfaces

Empecemos por entender, brevemente, qué es la herencia múltiple.

La herencia múltiple es un concepto de la programación orientada a objetos que permite a una clase heredar atributos y métodos de múltiples clases padre. Esto significa que, una clase puede adquirir características de varias clases diferentes. Sin embargo, en Java, no se permite la herencia múltiple directa, lo que significa que una clase no puede heredar de múltiples clases a la vez.

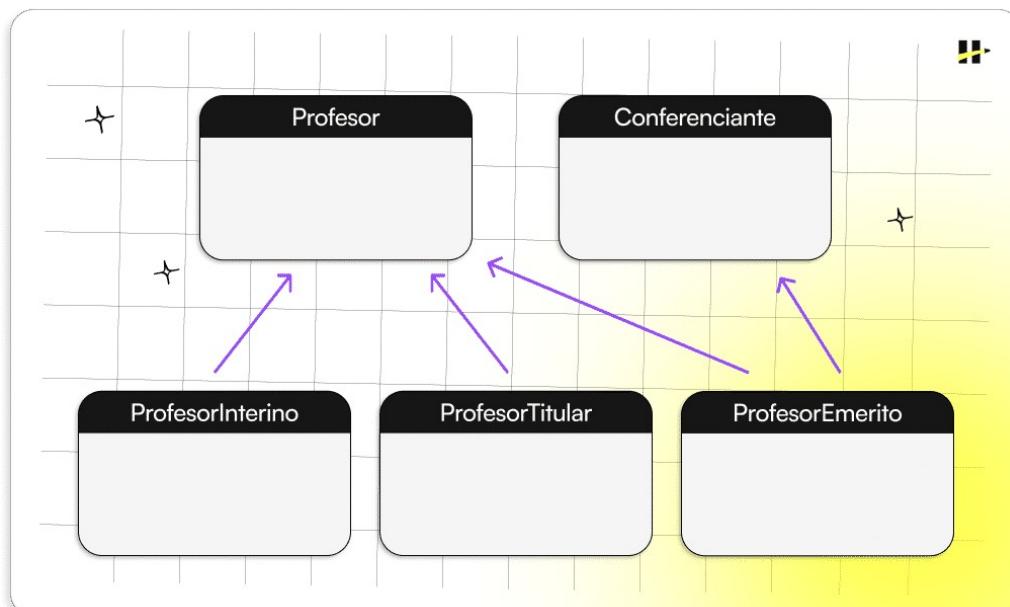
Para solucionar esta limitación, Java introduce el concepto de interfaces.

Las interfaces en Java tienen su origen en la necesidad de proporcionar una forma de lograr la abstracción y la implementación múltiple en una arquitectura orientada a objetos.

Una interfaz en Java es **similar a una clase abstracta, pero se utiliza principalmente para definir un conjunto de métodos abstractos y constantes que las clases que las implementan deben seguir**. Una clase puede implementar múltiples interfaces, lo que le permite adquirir diferentes conjuntos de comportamientos sin la necesidad de herencia múltiple. No puede tener implementaciones de métodos. Es verdad que también puede tener un método estático y otro default sin retorno (de tipo void), pero estos no forman parte del enfoque de esta clase ni del upskilling, por lo cual nos concentraremos en la principal característica de una interfaz: la de servir como un contrato.

Cuando decimos que una interfaz es un "contrato", nos referimos a que establece un conjunto de reglas y requisitos que una clase debe cumplir si decide implementar dicha interfaz. Al implementar una interfaz, una clase se compromete a proporcionar implementaciones concretas para todos los métodos declarados en la interfaz. Esto garantiza que las clases que implementan la interfaz tengan un comportamiento coherente y cumplan con las expectativas definidas por la interfaz.

Comparando con una clase abstracta, tanto las interfaces como las clases abstractas permiten definir métodos abstractos, es decir, métodos sin implementación concreta. Sin embargo, la principal diferencia radica en que una clase abstracta puede contener métodos con implementación, mientras que una interfaz sólo puede tener métodos abstractos y constantes. Además, una clase puede heredar de una sola clase abstracta, pero puede implementar múltiples interfaces.



Definición de una Interfaz en Java

Para definir una interfaz en Java, utilizamos la palabra clave `interface`. Aquí hay un ejemplo simple de una interfaz llamada `Calculadora` que define dos métodos: `sumar` y `restar`.

```
public interface Calculadora {  
    int sumar(int a, int b);  
    int restar(int a, int b);  
}
```

En este ejemplo, la interfaz `Calculadora` tiene dos métodos sin implementación: `sumar` y `restar`. Cualquier clase que implemente esta interfaz debe proporcionar una implementación concreta para estos métodos.

Implementación de una Interfaz en una Clase

Para implementar una interfaz en una clase, utilizamos la palabra clave `implements`. A continuación se muestra un ejemplo de una clase `CalculadoraBasica` que implementa la interfaz `Calculadora`.

```
public class CalculadoraBasica implements Calculadora {  
    @Override  
    public int sumar(int a, int b) {  
        return a + b;  
    }  
  
    @Override  
    public int restar(int a, int b) {  
        return a - b;  
    }  
}
```

En esta clase, hemos implementado los métodos `sumar` y `restar` según lo definido en la interfaz `Calculadora`.

Usos comunes de las Interfaces en Java

- **Implementación múltiple:** Una clase puede implementar múltiples interfaces, permitiendo una forma de herencia múltiple en Java.
- **Contratos y abstracción:** Las interfaces definen contratos que las clases deben seguir, proporcionando una forma de abstracción y estandarización del comportamiento.
- **Interoperabilidad:** Las interfaces facilitan la interacción entre diferentes partes de un sistema al establecer un conjunto común de métodos que se deben implementar.
- **Polimorfismo:** Permite el uso de polimorfismo, donde una interfaz puede ser utilizada para referenciar instancias de diferentes clases que la implementan.

En resumen, las interfaces en Java ofrecen una forma de lograr ciertos aspectos de la herencia múltiple al permitir que una clase implemente múltiples interfaces. Al hacerlo, una clase adquiere un conjunto específico de comportamientos definidos por esas interfaces, cumpliendo así un contrato y asegurando una coherencia en el diseño de nuestras aplicaciones.

Sugerimos complementar esta explicación con este video: <https://youtu.be/ZwtQJ8oYfWE>

Interfaces funcionales

Una interfaz funcional en Java es una interfaz que contiene exactamente un solo método abstracto. Este método es conocido como el "método funcional". La principal utilidad de una interfaz funcional es proporcionar un único punto de entrada para una función, permitiendo así su uso en expresiones lambda.

Características principales:

- **Un solo método abstracto:** Sólo debe contener un método abstracto sin implementación.
- **Múltiples métodos default o static:** Puede tener múltiples métodos default o static con implementaciones.
- **Usos de expresiones lambda:** Las interfaces funcionales se utilizan en Java principalmente para habilitar la programación funcional, permitiendo la implementación concisa de comportamientos a través de expresiones lambda.

Uso de `@FunctionalInterface`

La anotación `@FunctionalInterface` es opcional pero recomendada para marcar una interfaz como funcional. Esta anotación asegura que la interfaz cumple con la definición de una interfaz funcional (contiene un solo método abstracto). Si accidentalmente agregamos otro método abstracto, el compilador generará un error.

¡Nota breve sobre anotaciones en Java!

Las anotaciones en Java proporcionan información adicional sobre código (metadata), que se puede leer por otras herramientas y frameworks en tiempo de compilación, tiempo de ejecución o incluso tiempo de diseño. Se colocan como etiquetas arrobadas antes de la declaración de una clase, método o variable. Estas ayudan a mejorar la estructura y claridad del código, así como a automatizar tareas repetitivas.

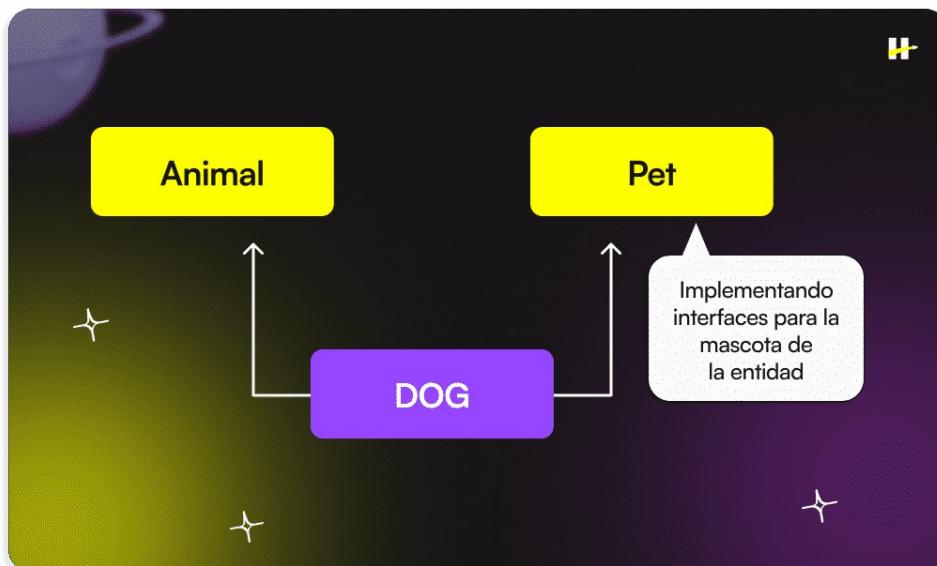
Ejemplo de Uso de `@FunctionalInterface`

Supongamos que queremos crear una interfaz funcional para representar una operación matemática. Aquí está un ejemplo:

```
@FunctionalInterface  
interface OperacionMatematica {  
    int operar(int a, int b);  
}  
  
public class Calculadora {  
    public static void main(String[] args) {  
        OperacionMatematica suma = (a, b) -> a + b;  
        OperacionMatematica resta = (a, b) -> a - b;  
  
        System.out.println("Suma: " + suma.operar(10, 5)); // Output: 15  
        System.out.println("Resta: " + resta.operar(10, 5)); // Output: 5  
    }  
}
```

En este ejemplo, definimos la interfaz funcional `'OperacionMatematica'` con un solo método `'operar'`. Usamos la anotación `'@FunctionalInterface'` para indicar que es una interfaz funcional. Luego, creamos expresiones lambda para representar la suma y la resta como operaciones matemáticas.

El objetivo de utilizar una interfaz funcional es facilitar la programación funcional en Java, permitiendo que los métodos puedan ser tratados como argumentos, devoluciones de llamada o expresiones lambda. Esto nos brinda flexibilidad y expresividad en el código, y nos ayuda a escribir programas más concisos y legibles.



Veamos estos conceptos con un ejemplo en video: <https://youtu.be/7MIB-K9AMxY>

Exploraremos brevemente las definiciones y ejemplos de algunas de las interfaces funcionales más comunes en Java: `Predicate`, `Function` y `Consumer`.

`Predicate<T>`

La interfaz funcional `'Predicate<T>'` representa una condición que se puede evaluar para cualquier tipo de dato. Tiene un método llamado `'test'` que toma un argumento de tipo `'T'` y devuelve un valor booleano

(seguramente habrás notado que hay un símbolo de diamante `<T>` junto al nombre de la interfaz, este es un concepto llamado "genéricos", que nos permite crear clases e interfaces que pueden trabajar con diferentes tipos de datos sin especificarlos de antemano. Pero no te preocupes, profundizaremos en este tema más adelante, por el momento, pensemos que T puede ser cualquier tipo dentro de nuestra aplicación).

Ejemplo:

```
import java.util.function.Predicate;
public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> esPar = num -> num % 2 == 0;

        System.out.println(esPar.test(4)); // Output: true
        System.out.println(esPar.test(7)); // Output: false
    }
}
```

Function<T, R>

La interfaz funcional `Function<T, R>` representa una función que toma un argumento de tipo `T` y devuelve un resultado de tipo `R`. Tiene un método llamado `apply`.

Ejemplo:

```
import java.util.function.Function;
public class FunctionExample {
    public static void main(String[] args) {
        Function<Integer, String> intToString = num -> "El número es: " + num;

        String result = intToString.apply(42);
        System.out.println(result); // Output: El número es: 42
    }
}
```

Consumer<T>

La interfaz funcional `Consumer<T>` representa una operación que toma un argumento de tipo `T` y no devuelve ningún resultado. Tiene un método llamado `accept`.

Ejemplo:

```
import java.util.function.Consumer;
public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> printMessage = message -> System.out.println("Mensaje: " + message);

        printMessage.accept("Hola, mundo!"); // Output: Mensaje: Hola, mundo!
    }
}
```

Comparable<T>

Esta interfaz define un único método llamado `compareTo()`, que se utiliza para comparar un objeto con otro objeto del mismo tipo.

La firma del método `compareTo()` en la interfaz `Comparable<T>` es la siguiente:

```
public interface Comparable<T> {
    int compareTo(T object);
```

```

    }}
```

```

public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public int compareTo(Person otherPerson) {
        // Comparación basada en la edad de las personas
        return Integer.compare(this.age, otherPerson.age);
    }

    public static void main(String[] args) {
        Person person1 = new Person("Alice", 25);
        Person person2 = new Person("Bob", 30);

        int result = person1.compareTo(person2);
        if (result < 0) {
            System.out.println(person1.getName() + " es menor que " +
person2.getName());
        } else if (result > 0) {
            System.out.println(person1.getName() + " es mayor que " +
person2.getName());
        } else {
            System.out.println(person1.getName() + " es igual a " +
person2.getName());
        }
    }
}

```

El objetivo principal de la interfaz `Comparable<T>` es proporcionar una forma de comparar objetos y determinar su orden. Al implementar esta interfaz en una clase, podemos definir cómo se debe realizar la comparación entre dos objetos de esa clase. Esto es especialmente útil cuando necesitamos ordenar una lista de objetos según algún criterio específico.

Para ahondar un poco más en este tema, pueden ver los siguientes videos:

<https://youtu.be/mBqx4ze3-x4>

<https://youtu.be/0zcnv4hVvVw>

Manejo de errores y excepciones

En Java, el manejo de errores y excepciones es esencial para escribir aplicaciones fiables y robustas. Estas situaciones anómalas pueden ocurrir durante la ejecución y deben ser manejadas adecuadamente para garantizar un flujo controlado del programa.

Los errores, representados por la clase `Error`, indican problemas graves que generalmente no se pueden recuperar y deberían evitar que el programa continúe su ejecución normal. Algunos ejemplos de errores son:

- `OutOfMemoryError`: Se produce cuando la JVM se queda sin memoria.
- `StackOverflowError`: Se produce cuando la pila de llamadas al método está llena.

Por otro lado, las excepciones, representadas por la clase `Exception`, son situaciones excepcionales que pueden manejarse durante la ejecución del programa. Estas situaciones no son necesariamente fatales y permiten una recuperación controlada.

Esto quiere decir que, básicamente son un evento o situación que ocurre durante la ejecución de un programa y que interrumpe su flujo normal. Podemos pensar en las excepciones como un flujo indeseado, pero no desconocido. Es decir, aunque no siempre podamos prever qué excepciones específicas se producirán, sabemos que en algún momento surgirán y debemos estar preparados para manejárlas adecuadamente.

Es importante recordar que las excepciones son inevitables. No podemos evitar que ocurran pero sí podemos aprender a lidiar con ellas de manera efectiva. Aprendemos a manejar excepciones a partir de la experiencia, encontrándonos con ellas y buscando soluciones.

Es necesario recordar que las excepciones no son exclusivas de los lenguajes de programación orientados a objetos, pero en Java son una parte fundamental. Como programadores, debemos ser capaces de lanzar, gestionar e incluso crear nuestras propias excepciones de manera adecuada, siguiendo las prácticas y estándares propios del lenguaje.

Las excepciones pueden dividirse en dos tipos principales

Excepciones verificadas (Checked Exceptions): Subclases de `Exception` que no son subclases de `RuntimeException`. Deben manejarse explícitamente usando `try-catch` o declararse en la cláusula `throws` del método.

A continuación podemos ver un ejemplo básico de cómo manejar una excepción usando los bloques **try/catch**:

```
import java.io.IOException;
public class CheckedExceptionExample {

    public static void main(String[] args) {
        try {
            // Intenta ejecutar código que puede lanzar una excepción
            throw new IOException("Error de E/S");
        } catch (IOException e) {
            // Maneja la excepción
            System.err.println("Excepción atrapada: " + e.getMessage());
        }
    }
}
```

Excepciones no verificadas (Unchecked Exceptions): Subclases de `RuntimeException`. Estas no están obligadas a ser capturadas o declaradas y suelen ser errores en el código del programador.

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int result = 10 / 0; // Esto arrojará ArithmeticException
    }
}
```

El manejo de excepciones nos permite controlar el flujo de nuestra aplicación y tomar decisiones sobre si la aplicación se rompe o no. No queremos que sea la aplicación misma quien decida eso, sino nosotros como programadores. ¡Nosotros tenemos el poder!

Excepciones en JAVA

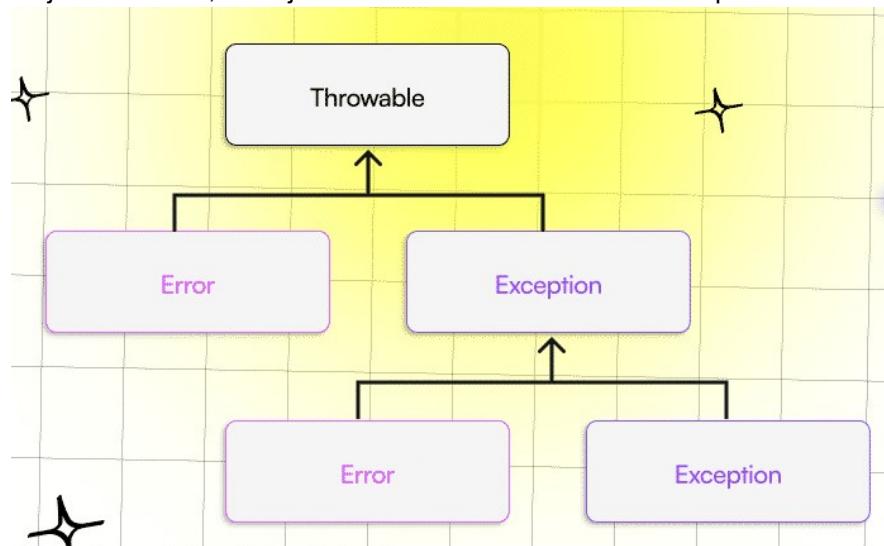
Cuando se produce una excepción en Java, se crea un objeto especial para representar esa excepción. Y como sabemos, en Java, todo es un objeto, así que no debería resultarnos extraño.

Este objeto de excepción contiene información muy valiosa sobre el error que ocurrió durante la ejecución del programa. Imaginen que este objeto es como un **gabinete de archivos o fichero** que guarda todos los detalles necesarios para entender qué salió mal y por qué.

Ahora bien, existe una **clase muy importante** en Java llamada **Throwable**, que actúa como la clase padre de todas las excepciones. Podemos decir que establece una jerarquía para organizar y clasificar las diferentes excepciones que pueden ocurrir.

La clase **Throwable** tiene dos subclases principales: **Exception** y **Error**, ya tratamos ambas en la sección anterior y vimos que las excepciones se utilizan para representar situaciones excepcionales que pueden ocurrir durante la ejecución de un programa, como errores de entrada/salida, problemas de conexión a una base de datos, entre otros. Por otro lado, los errores (Error) representan problemas más graves que generalmente están fuera del control del programador, como la falta de memoria en el sistema.

La jerarquía de excepciones en Java nos permite manejar diferentes tipos de errores de manera más específica. Podemos capturar excepciones específicas y tomar acciones particulares para manejarlas correctamente. Esto nos brinda mayor flexibilidad y control sobre el flujo de nuestro programa. Para comprender mejor este tema, te dejamos este **video** en donde se explora más a fondo este tema:



https://youtu.be/mttQnRc-z_I

Manejo del try catch

Como ya vimos brevemente en una sección previa, una forma común de controlar las excepciones es mediante el uso de bloques try/catch. Esto nos permite capturar las posibles excepciones que puedan ocurrir en un bloque de código y tomar acciones específicas para manejarlas. El flujo de ejecución sigue un orden: primero se ejecuta el bloque try, y si ocurre una excepción, se busca un bloque catch que pueda manejarla.

```
try {  
    // Código que puede lanzar una excepción  
// ...  
} catch (ExceptionType1 ex) {  
    // Manejar las excepciones del tipo ExceptionType1  
    // Puedes usar el objeto ex para obtener información sobre la excepción  
}
```

En un bloque catch, podemos declarar múltiples excepciones separadas por un operador de tubo (pipe -> |) o concatenando un catch tras otro.

```
try {  
    // Código que puede lanzar una excepción  
// ...  
} catch (ExceptionType1 | ExceptionType2 e) {  
    // Manejar las excepciones del tipo ExceptionType1 o ExceptionType2  
}  
-----  
try {  
    // Código que puede lanzar una excepción  
    // ...  
}
```

```

} catch (ExceptionType1 e) {
    // Manejar la excepción del tipo ExceptionType1
} catch (ExceptionType2 e) {
    // Manejar la excepción del tipo ExceptionType2

```

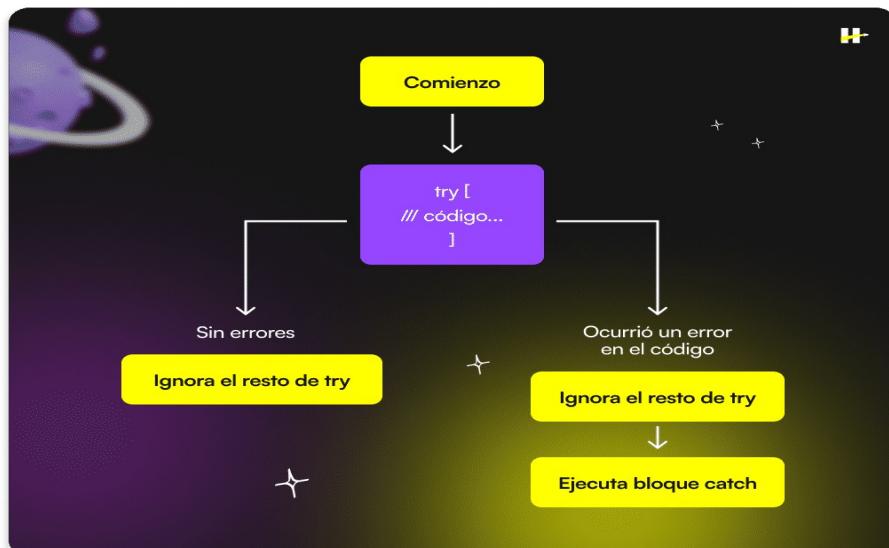
Es importante tener en cuenta que existe una jerarquía de excepciones, por lo que es recomendable declarar las excepciones más específicas primero y luego las más generales y abarcadoras. De esta manera, garantizamos que las excepciones sean capturadas y manejadas correctamente.

El bloque **finally** se ejecuta siempre, independientemente de si se lanzó una excepción o no. Es útil para liberar recursos que se han adquirido en el bloque try o realizar otras operaciones que deben realizarse de manera obligatoria.

```

try {
    // Código que puede lanzar una excepción
    // ...
} catch (Exception e) {
    // Manejar la excepción
} finally {
    // Código que se ejecuta siempre
    // Liberar recursos u otras operaciones necesarias
}

```



Es importante destacar que podemos obviar el bloque **finally** en un bloque try/catch. El bloque finally se utiliza para ejecutar código que debe ser ejecutado sin importar si se produce o no una excepción. Sin embargo, en algunas situaciones, podemos optar por no incluirlo y solo utilizar el bloque catch para manejar las excepciones.

Veamos un muy breve ejemplo concreto de los casos de manejo de excepciones que acabamos de aprender:

```

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
            arr[5] = 10; // Esto lanzará ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException | NullPointerException e) {
            System.err.println("Excepción atrapada: " + e.getMessage());
        } finally {
            System.out.println("Este bloque se ejecuta siempre.");
        }
    }
}

```

Propagando excepciones

Otro aspecto clave es la propagación de excepciones. En ciertos casos, podemos necesitar pasar la responsabilidad de manejar una excepción a otro componente o método. Esto se conoce como propagación

de excepciones. Podemos hacerlo de dos formas: arrojando una excepción de forma explícita con la palabra clave `throw` e instanciando la excepción que buscamos arrojar, o de forma implícita, simplemente **propagando la excepción** a través de la firma del método, con la palabra `throws`, sin manejarla en ese componente en particular. La decisión de propagar una excepción depende del programador y la lógica de negocio definida.

Propagación Implícita

Cuando un método lanza una excepción, **no la maneja internamente** pero la declara con `'throws'`, se produce una propagación implícita. Esto significa que la excepción se propaga automáticamente al método que lo llamó.

```
public class ExceptionPropagationExample {  
    public static void main(String[] args) {  
        try {  
            doSomething(); // Llamada al método que lanza una excepción  
        } catch (Exception e) {  
            System.err.println("Excepción capturada en el método main: " +  
e.getMessage());  
        }  
    }  
    public static void doSomething() throws Exception {  
        // Un bloque de código que lanza una excepción  
    }  
}
```

En este ejemplo, el método `'doSomething()'` lanza una excepción, y como no la maneja ni declara con `'throws'`, la excepción se propaga al método que lo llamó, en este caso, el método `'main'`.

Propagación Explícita

También podemos propagar una excepción explícitamente usando la palabra clave `'throw'`. Esto es útil cuando queremos agregar información adicional o personalizar un mensaje antes de propagar la excepción.

```
public class ExceptionPropagationExample {  
    public static void main(String[] args) {  
        try {  
            doSomething();  
        } catch (Exception e) {  
            System.err.println("Excepción capturada en el método main: " +  
e.getMessage());  
        }  
  
        public static void doSomething() {  
            try {  
                // Algo que puede lanzar una excepción  
                int result = 10 / 0;  
            } catch (ArithmException e) {  
                // Agregar información adicional antes de propagar  
                throw new CustomException("Error aritmético: " + e.getMessage());  
            }  
        }  
    }  
}
```

En este ejemplo, la excepción `'ArithmException'` se captura en el método `'doSomething()'`, se personaliza el mensaje y se lanza una nueva excepción (`'CustomException'`) con un mensaje mejorado.

Para profundizar en este tema te alentamos a mirar este video: <https://youtu.be/QSohwTY04Go>

Excepciones customizadas

En Java, tenemos la flexibilidad de crear nuestras propias excepciones personalizadas al extender la clase base `Exception`. Esto nos permite definir nuestras propias excepciones que se adaptan específicamente a las necesidades de nuestro programa y nos brindan un mayor control sobre el manejo de errores. Veamos un video introductorio al tema y después, veremos algunas consideraciones <https://youtu.be/v43Tl9dMPE>

La personalización de excepciones es especialmente útil cuando queremos capturar y manejar situaciones excepcionales que son específicas de nuestro programa o dominio. Por ejemplo, si estamos desarrollando una aplicación bancaria, podríamos definir una excepción personalizada llamada `'SaldolInsuficienteException'` para manejar los casos en los que un cliente intenta realizar una transacción con un saldo insuficiente en su cuenta.

Al crear nuestras excepciones personalizadas, podemos agregar información adicional y mensajes descriptivos que ayuden a comprender la causa del error. Esto es útil tanto para los desarrolladores que trabajarán con nuestro código como para los usuarios finales que puedan encontrar estas excepciones.

¿Cómo creamos una excepción customizada?

Para crear una excepción personalizada, debes definir una nueva clase que extiende `Exception` o una de sus subclases. Puedes agregar campos, métodos y constructores adicionales según tus necesidades.

```
public class CustomException extends Exception {  
    // Campos, métodos y constructores personalizados  
  
    public CustomException(String message) {  
        super(message);  
    }  
}
```

En este ejemplo, `CustomException` es una excepción personalizada que hereda de `Exception`. Hemos definido un constructor para establecer el mensaje de la excepción.

Puedes lanzar tu excepción personalizada de manera explícita usando la palabra clave `throw`, como ya vimos en la sección de propagación, seguida de una instancia de tu clase de excepción.

```
public class CustomExceptionExample {  
    public static void main(String[] args) {  
        try {  
            throw new CustomException("Este es un mensaje personalizado.");  
        } catch (CustomException e) {  
            System.err.println("Excepción capturada: " + e.getMessage());  
        }  
    }  
}
```

En este ejemplo, lanzamos la excepción personalizada `CustomException` y luego la capturamos y mostramos el mensaje.

Puedes usar tu excepción personalizada en tu código y capturarla como lo harías con cualquier otra excepción. Por ejemplo, en el siguiente bloque de código, la excepción es propagada de manera implícita:

```
public class CustomExceptionUsageExample {  
    public static void main(String[] args) {  
        try {  
            performCustomOperation();  
        } catch (CustomException e) {  
            System.err.println("Excepción capturada: " + e.getMessage());  
        }  
  
        public static void performCustomOperation() throws CustomException {  
            // Algo que puede lanzar la excepción personalizada  
        }  
    }  
}
```

Vale la pena notar, que al personalizar nuestras excepciones, estamos siguiendo el principio de encapsulamiento, que es fundamental en la programación orientada a objetos. Al definir nuestras propias excepciones, estamos encapsulando la lógica de manejo de errores en una unidad coherente y modular, lo que hace que nuestro código sea más legible, mantenible y reutilizable.

¡No tengas miedo en personalizar las excepciones! Es una habilidad valiosa que te permitirá tener un mayor control sobre el manejo de errores en sus programas y mejorar la experiencia de usuario.

Puedes complementar lo aprendido en esta sección con el siguiente video: <https://youtu.be/LvkxCgs5Lhw>

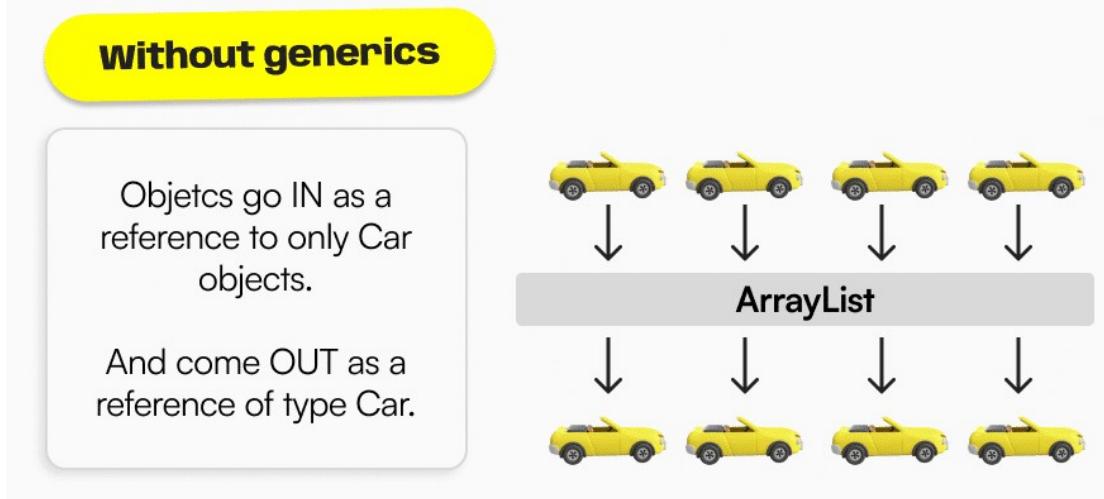
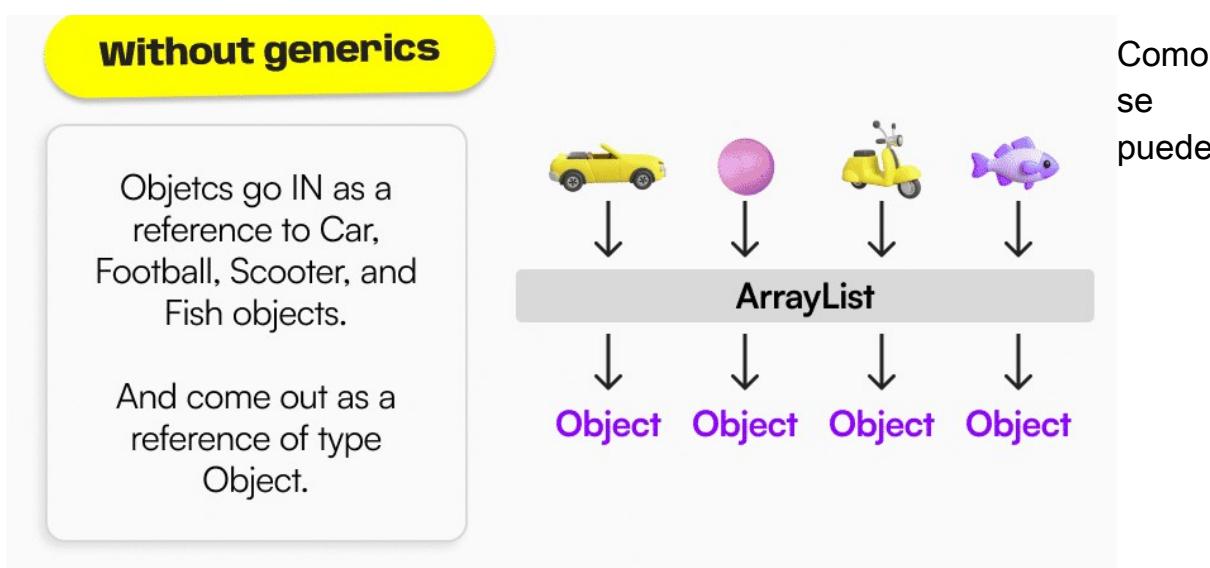
¿Por qué necesitamos genéricos?

Los **genéricos** en **Java** ofrecen la **posibilidad de crear clases, interfaces y métodos** que pueden ser parametrizados con tipos específicos.

Esta característica brinda **flexibilidad** y **reutilización de código**, permitiéndonos desarrollar componentes que funcionen con diferentes tipos de datos sin necesidad de duplicar código.

La idea central de los genéricos es permitir a los **desarrolladores crear estructuras de datos y algoritmos que sean independientes** del tipo de datos con el que se vaya a trabajar. Esto significa que podemos escribir una clase o método capaz de manejar diferentes tipos de datos de manera segura y coherente.

Es importante tener en cuenta que los genéricos **no funcionan con tipos de datos primitivos** en Java. Esto se debe a que los genéricos operan en tiempo de compilación y, en el fondo, el motor de Java "**borra**" los parámetros de tipo y los implementa como tipo Object. Por lo tanto, los genéricos en Java solo pueden trabajar con tipos de referencia.



observar en la figura anterior, sin el uso de genéricos, es posible colocar cualquier tipo de objeto en un ArrayList sin que el compilador de Java genere un error.

En contraste, al utilizar genéricos, podemos restringir el tipo de elementos que se pueden añadir a un ArrayList, lo que resulta en una colección de tipo seguro. Esto permite detectar y resolver posibles problemas en tiempo de compilación en lugar de tiempo de ejecución.

Asimismo, el siguiente video te proporcionará más información sobre el uso de genéricos en Java y su importancia en la creación de colecciones seguras y reutilizables.

Ver video https://youtu.be/MFu8a_LpnIc

Tipos de genéricos: clases y métodos

Sin el uso de genéricos, podríamos tener una clase llamada `CajaDeHerramientas` con dos listas: `herramientasHerrero` y `herramientasCarpintero`.

A continuación, se muestra un ejemplo de cómo podríamos instanciar y usar esas listas sin genéricos:

```
public class CajaDeHerramientas {  
    private List<HerramientasHerrero> herramientasHerrero;  
    private List<HerramientasCarpintero> herramientasCarpintero;  
  
    public CajaDeHerramientas() {  
        herramientasHerrero = new ArrayList<>();  
        herramientasCarpintero = new ArrayList<>();  
    }  
  
    public void agregarHerramienta(Herramienta herramienta) {  
        if (herramienta instanceof HerramientasHerrero) {  
            herramientasHerrero.add((HerramientasHerrero) herramienta);  
        } else if (herramienta instanceof HerramientasCarpintero) {  
            herramientasCarpintero.add((HerramientasCarpintero) herramienta);  
        }  
    }  
  
    // Otros métodos de la clase...  
}
```

En este ejemplo, utilizamos el operador `instanceof` para validar el tipo de la herramienta y luego realizar un cast al tipo correspondiente antes de agregarlo a la lista correspondiente.

Ahora, podemos refactorizar este ejemplo utilizando genéricos para lograr el mismo resultado de una manera más elegante y segura. La sintaxis básica de los genéricos se define en la clase y en los métodos dentro de esa clase de la siguiente manera:

```
public class CajaDeHerramientas<T> {  
    private List<T> herramientas;
```

```

public CajaDeHerramientas() {
    herramientas = new ArrayList<>();
}

public void agregarHerramienta(T herramienta) {
    herramientas.add(herramienta);
}

// Otros métodos de la clase...
}

```

En este caso, hemos utilizado el tipo genérico 'T' para representar el tipo de herramienta. Esto nos permite crear una instancia de 'CajaDeHerramientas' para cualquier tipo de herramienta y agregar herramientas directamente a la lista sin necesidad de validaciones o castings adicionales.

Aquí hay un ejemplo de cómo se usaría la clase 'CajaDeHerramientas' con genéricos:

```

CajaDeHerramientas<HerramientasHerrero> cajaHerrero = new
CajaDeHerramientas<>();
cajaHerrero.agregarHerramienta(new HerramientasHerrero());

CajaDeHerramientas<HerramientasCarpintero> cajaCarpintero = new
CajaDeHerramientas<>();
cajaCarpintero.agregarHerramienta(new HerramientasCarpintero());

```

En este ejemplo, hemos creado dos instancias de '**CajaDeHerramientas**' para diferentes tipos de herramientas y agregamos herramientas directamente a las listas correspondientes sin necesidad de validar tipos o realizar castings.

Los genéricos nos permiten definir clases y métodos que pueden trabajar con diferentes tipos de datos de manera segura y sin duplicar código.

En este video vas a ver un ejemplo del uso de Generics:

[VER VIDEO 1 DE CLASE DE HENRY](#)

Bounded Generics o genéricos restringidos

Al trabajar con **genéricos** en Java, podemos aplicar restricciones a los parámetros de tipo. Estas restricciones nos permiten definir límites en los tipos que pueden ser utilizados en una clase o método genérico. Dos tipos comunes de restricciones son el upper bound (límite superior) y el lower bound (límite inferior).

Upper bound (límite superior)

Al utilizar el upper bound, especificamos que el tipo de parámetro debe ser una clase en particular o cualquier subclase de esa clase. Esto se logra utilizando la palabra clave "extends" seguida del nombre de la clase o interfaz.

Ejemplo:

```
public class MiClase<T extends MiSuperClase> {  
    // Código de la clase  
}
```

En este ejemplo, el tipo de parámetro T se restringe a ser una clase que extienda o implemente MiSuperClase.

Lower bound (límite inferior)

Al utilizar el lower bound, especificamos que el tipo de parámetro debe ser una clase en particular o cualquier superclase de esa clase. Esto se logra utilizando el wildcard "?" junto con la palabra clave "super" seguida del nombre de la clase.

Ejemplo

```
public class MiClase<T super MiSuperClase> {  
    // Código de la clase  
}
```

En este ejemplo, el tipo de parámetro T se restringe a ser una clase que sea MiSuperClase o una superclase de MiSuperClase.

Estas restricciones nos permiten definir de manera más precisa los tipos que pueden ser utilizados en una clase o método genérico, lo que proporciona mayor flexibilidad y seguridad en la programación genérica.

Material extra genericos: <https://youtu.be/GKJl-4oNUWg>

¿Qué son los wildcards?

Los wildcards o comodines en Java se representan con el signo de interrogación '?'. Estos comodines se utilizan cuando queremos trabajar con tipos genéricos pero no conocemos el tipo específico al que se refieren.

El uso de wildcards en Java nos permite escribir código genérico más flexible y reutilizable al trabajar con tipos genéricos desconocidos o con límites superiores o inferiores específicos. Esto facilita la interoperabilidad y la adaptabilidad en el manejo de colecciones y componentes genéricos.

Ejemplo

```
public void miMetodo(List<?> lista) {  
    // Código del método  
}
```

En este ejemplo, el método miMetodo acepta una lista de cualquier tipo, pero el tipo exacto es desconocido. Esto permite que el método sea más genérico y pueda trabajar con diferentes tipos de listas.

Bounded Wildcards (Comodines con restricciones): Se utilizan para establecer restricciones en el tipo referido. Pueden ser de dos tipos: Upper Bounded Wildcards (comodines con límite superior) y Lower Bounded Wildcards (comodines con límite inferior).

Upper Bounded Wildcards: Se representan utilizando el keyword "extends" seguido del tipo límite superior. Esto permite que el tipo referido sea el tipo límite o cualquier subtipo de ese tipo.

Ejemplo

```
public void miMetodo(List<? extends Number> lista) {  
    // Código del método  
}
```

En este ejemplo, el método miMetodo acepta una lista de cualquier tipo que sea una subclase de Number. Esto significa que podemos pasar una lista de Integer, Double, Float, etc.

Lower Bounded Wildcards: Se representan utilizando el keyword "super" seguido del tipo límite inferior. Esto permite que el tipo referido sea el tipo límite o cualquier superclase de ese tipo.

Ejemplo

```
public void miMetodo(List<? super Integer> lista) {  
    // Código del método  
}
```

En este ejemplo, el método miMetodo acepta una lista de cualquier tipo que sea Integer o una superclase de Integer. Esto significa que podemos pasar una lista de Integer, Number, Object, etc.

El uso de wildcards nos brinda flexibilidad al trabajar con tipos genéricos y nos permite escribir código más genérico y reutilizable en situaciones donde no conocemos el tipo exacto al que se refiere.

VER VIDEO 2 DE CLASE DE HENRY

HOMEWORK

Crear una clase llamada Utilidades con 2 métodos para manipular listas:

- Un método será imprimirElementos() -> donde se recorrerá la lista, los elementos pueden ser de cualquier tipo .
- Otro método será copiarElementos() -> donde se deberá usar bounded generic para asegurar que la lista de destino sea de un tipo igual o superclase del tipo de la lista de origen .
- Además, utilizar una wildcard para indicar que la lista de origen puede ser de cualquier tipo .

Para probar esta clase genérica, realizar operaciones en el main, creando listas de diferentes tipos (Integer, String o creando alguna clase personalizada) y utilizando los métodos imprimirElementos() y copiarElementos()

ejercicio resuelto: https://drive.google.com/drive/folders/1akyPCEvOqQzy7ULY-noUXpT9Wm68M9oD?usp=share_link

Continuamos con el proyecto...

Durante el repaso de la última clase del módulo 2, es importante verificar que hayamos implementado correctamente los siguientes elementos en nuestro proyecto integrador:

- **Transición de arrays a colecciones:** Hemos refactorizado todas las manipulaciones de arrays en nuestra aplicación para que utilicen colecciones, ya sea List o Set. Esto nos permite aprovechar las ventajas de las operaciones y funcionalidades proporcionadas por las colecciones en Java.
- **Utilización de Map<K, V>:** Hemos implementado al menos un Map en nuestro proyecto, donde hemos utilizado una clave (K) y un valor (V) para almacenar y recuperar información de forma eficiente.
- **Llamadas funcionales:** Hemos incorporado al menos dos llamadas funcionales en nuestro código. Estas llamadas pueden ser utilizadas para realizar validaciones, iteraciones o mapeos en nuestras colecciones. La utilización de funciones como filter(), map() o forEach() nos permite realizar estas operaciones de manera más concisa y legible.
- **Uso de clases o interfaces con genéricos:** Hemos implementado al menos una clase o interfaz en nuestro proyecto que utiliza genéricos. Esto nos brinda flexibilidad y reutilización de código al poder trabajar con diferentes tipos de datos sin tener que duplicar la implementación.

Para esta última etapa y cierre de módulo en nuestro Proyecto Integrador, ¡no nos olvidemos de nuestra última lección! vamos a implementar lo aprendido sobre **Generics**.

Debemos generar al menos un método en nuestro proyecto que acepte cualquier tipo de objeto para realizar alguna lógica específica.

Por ejemplo, un método para listar objetos, ya sean gastos o categorías. Esto nos permitirá trabajar con diferentes tipos de datos de manera genérica y adaptable.

A continuación, te dejamos una aproximación a cómo puedes implementar en el Proyecto Integrador todos los temas aprendidos hasta el momento. Pero ten en cuenta que este hito del PI es solo una idea base de cómo lo puedes hacer tú, ¡recuerda intentarlo por tu cuenta antes de ver esta posible solución!

VER VIDEO 3 HENRY CLASE 3

Modelo de Proyecto Integrador M2

Clíckea en "Proyecto" para acceder al archivo modelo de Proyecto Integrador de este primer módulo.

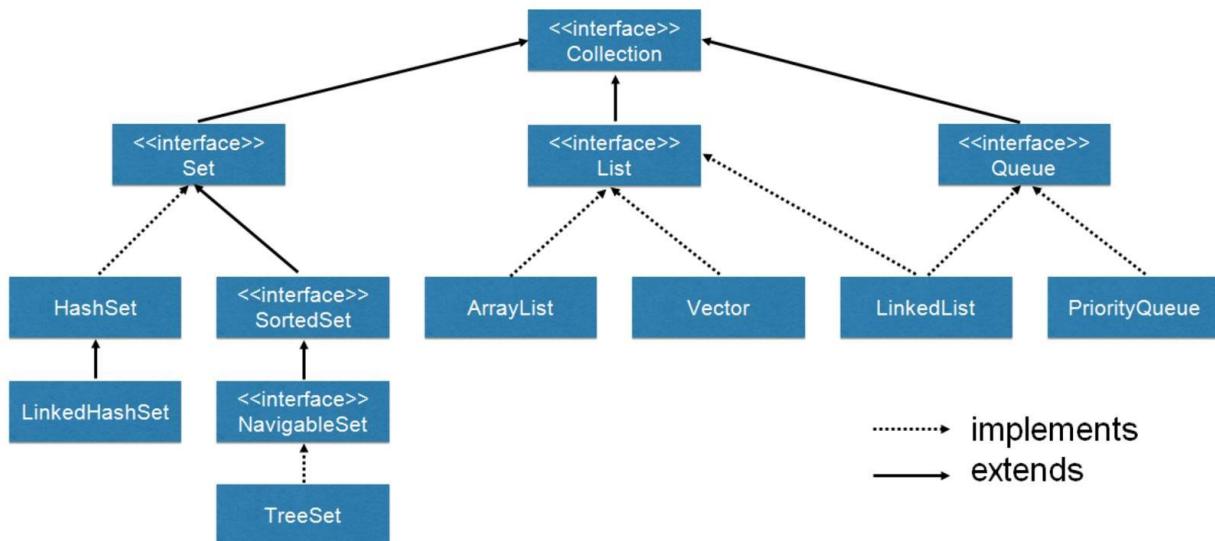
<https://drive.google.com/file/d/1SGRJgcEfBrHSHzG3cSrfhbxn3NEPDeFD/view?usp=sharing>

Sobre la librería de Collections

En **Java**, las interfaces de colecciones forman una jerarquía donde la **interfaz Collection** se encuentra en la parte superior. Esto significa que **Collection** es la interfaz base de la cual derivan otras interfaces de colecciones más específicas.

Pensemos en un diagrama general de esta jerarquía de interfaces de colecciones. En la cima se encuentra la interfaz Collection, y debajo de ella encontramos otras interfaces como List, Set, Queue, etc. Cada una de estas interfaces tiene sus propias características y métodos específicos, pero todas comparten algunos métodos comunes definidos en la interfaz Collection.

Collection Interface



La **librería de colecciones en Java** proporciona estas interfaces genéricas para trabajar con colecciones de objetos. Estas interfaces son altamente flexibles y permiten manipular y gestionar **colecciones** de manera eficiente.

Es importante mencionar que...

Estas interfaces genéricas son diferentes de los **arrays** (arreglos) en Java. Mientras que los arrays son estructuras de datos fijas con una longitud determinada, las colecciones ofrecen mayor flexibilidad y dinamismo. Las **colecciones** pueden crecer o disminuir su tamaño según sea necesario y proporcionan métodos convenientes para **agregar, eliminar y buscar elementos**.

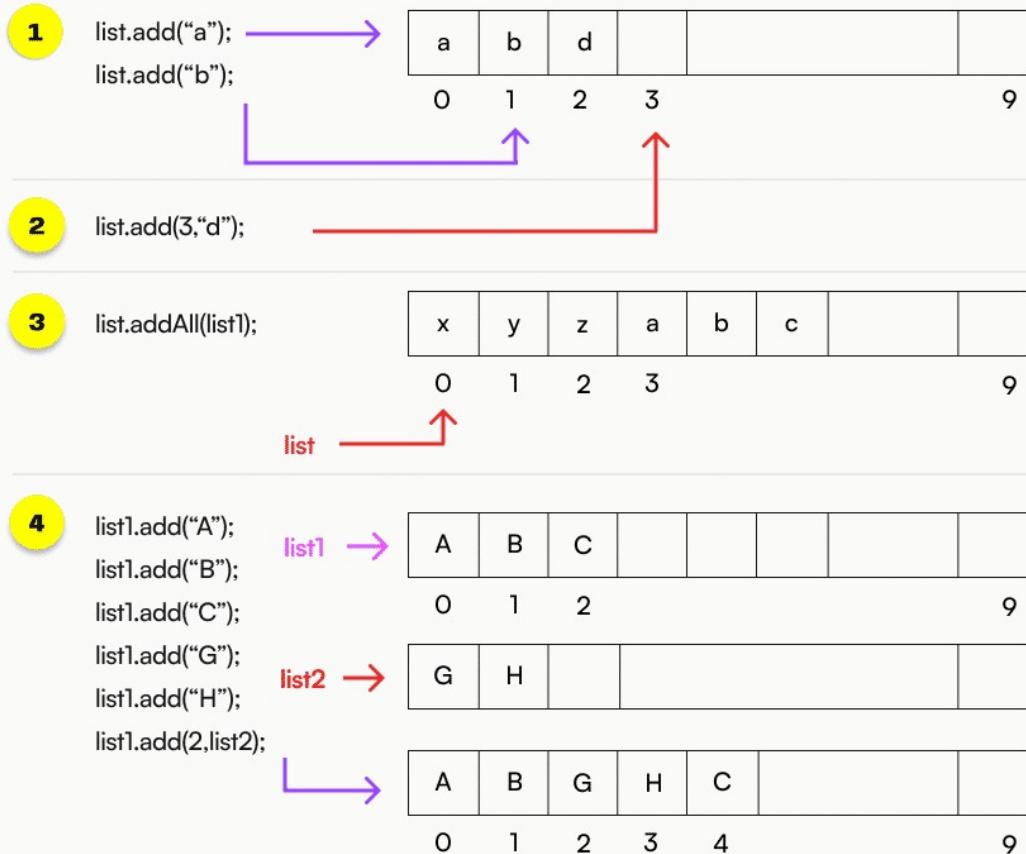
En resumen, la jerarquía de interfaces de colecciones en Java se basa en la **interfaz Collection** como la **interfaz base**, y a partir de ella se derivan otras interfaces más específicas. La librería de **colecciones de Java** ofrece estas interfaces genéricas para trabajar con colecciones de objetos, brindando métodos útiles y flexibilidad en el manejo de datos. Es importante tener en cuenta la **diferencia** entre un **array** y una **colección**, ya que las colecciones ofrecen mayor dinamismo y funcionalidad en comparación con los arrays.

Para entender mejor estos conceptos, te proponemos ver este video: https://youtu.be/mE1vnyN_QgU

Listas

En el **mundo de la programación**, a menudo necesitamos **manejar colecciones de elementos**, ya sea para almacenar datos, procesar información o realizar operaciones específicas. Una de las **estructuras de datos más utilizadas para este propósito es la lista** (`List<tipo>`).

En Java, una **lista es una interfaz que proporciona una forma ordenada de almacenar y manipular elementos**. Una lista nos permite mantener una secuencia de elementos en un orden específico y **realizar diversas operaciones** sobre ellos, como agregar, eliminar, buscar y acceder a elementos individuales.



La **interfaz List en Java** define un conjunto de métodos estándar para trabajar con **listas**, lo que nos permite aprovechar su funcionalidad sin preocuparnos por los detalles de implementación. Es importante destacar que la **interfaz List** es solo una especificación y no se puede **instanciar** directamente.

Sin embargo, podemos usar clases que implementen la interfaz **List**, como **ArrayList**, **LinkedList** o **Vector**.

Las **listas en Java tienen algunas características clave**. En primer lugar, mantienen el **orden de inserción de los elementos**, lo que significa que los elementos se colocan en la lista en el orden en que los agregamos. Además, **las listas nos permiten tener elementos duplicados, es decir, podemos tener varios elementos idénticos en una lista**.

El **acceso a los elementos** de una lista se realiza mediante un índice, lo que significa que podemos acceder a un elemento en particular utilizando su **posición en la lista**. También podemos modificar o eliminar elementos según su posición.

Otra característica importante de **las listas en Java** es que tienen un **tamaño variable**, lo que significa que podemos **agregar o eliminar elementos en cualquier momento sin restricciones**. Esto nos brinda flexibilidad para adaptar la lista a nuestras necesidades cambiantes.

ArrayList y **LinkedList** son implementaciones de la interfaz List en Java y se utilizan para **almacenar y manipular colecciones** de elementos en orden secuencial. Sin embargo, difieren en la forma en que se organizan y acceden a los elementos en la lista, lo que afecta su rendimiento en diferentes operaciones.

ArrayList

- ArrayList está respaldado por un arreglo (array) de tamaño dinámico, lo que significa que puede crecer y encogerse automáticamente según sea necesario.

- Permite un acceso rápido a los elementos a través de su índice.
- Proporciona un tiempo de acceso constante ($O(1)$) para recuperar un elemento por índice.
- Es eficiente en la lectura de elementos y recorridos secuenciales.

- Sin embargo, puede ser menos eficiente en la inserción o eliminación de elementos en el medio de la lista, ya que puede requerir el desplazamiento de los elementos siguientes en el arreglo.

¿Cómo creamos un arrayList?

```
List<String> ejemploArrayList = new ArrayList<>();  
ejemploArrayList.add("ejemplo"); //Con este método se agregan elementos a la lista
```

LinkedList

- LinkedList se implementa como una lista doblemente enlazada, donde cada elemento contiene una referencia al siguiente y al anterior en la lista.
- Ofrece un tiempo de inserción y eliminación constante ($O(1)$) en cualquier posición de la lista, ya que solo se requieren ajustes de los enlaces.
- No es tan eficiente como ArrayList en el acceso aleatorio por índice, ya que puede requerir un recorrido secuencial desde el principio o el final de la lista.
- Es más eficiente en la inserción o eliminación de elementos en el medio de la lista, ya que no requiere el desplazamiento de elementos en un arreglo.

```
LinkedList<String> linkedList = new LinkedList<>();  
linkedList.add("Manzana"); //Con este método se agregan elementos a la lista
```

Comparación

- En general, si se requiere un acceso frecuente a los elementos por índice o se realizan operaciones de lectura secuencial, ArrayList tiende a ser más eficiente debido a su implementación de matriz dinámica.
- Por otro lado, si se realizan operaciones frecuentes de inserción o eliminación de elementos en cualquier posición de la lista, LinkedList es más eficiente debido a su estructura de lista doblemente enlazada.
- La elección entre ArrayList y LinkedList depende de las operaciones y requisitos específicos de tu aplicación. Si no estás seguro, es recomendable realizar pruebas de rendimiento para evaluar cuál se adapta mejor a tus necesidades.

Es importante tener en cuenta que estas son solo algunas de las diferencias clave entre ArrayList y LinkedList, y que hay otros factores a considerar, como el uso de memoria, el tamaño de la lista y el tipo de operaciones que se realizan con mayor frecuencia.

En resumen, las listas en Java nos permiten almacenar y manipular elementos en un orden específico, mantener duplicados y proporcionar métodos útiles para realizar operaciones comunes. Son ampliamente utilizadas en aplicaciones Java para gestionar colecciones de datos y facilitar el procesamiento de información.

Set

Un Set en Java es una interfaz que define una colección de elementos únicos, lo que significa que no puede contener duplicados. Esta interfaz se utiliza cuando necesitamos almacenar elementos sin preocuparnos por su orden o posición específica.

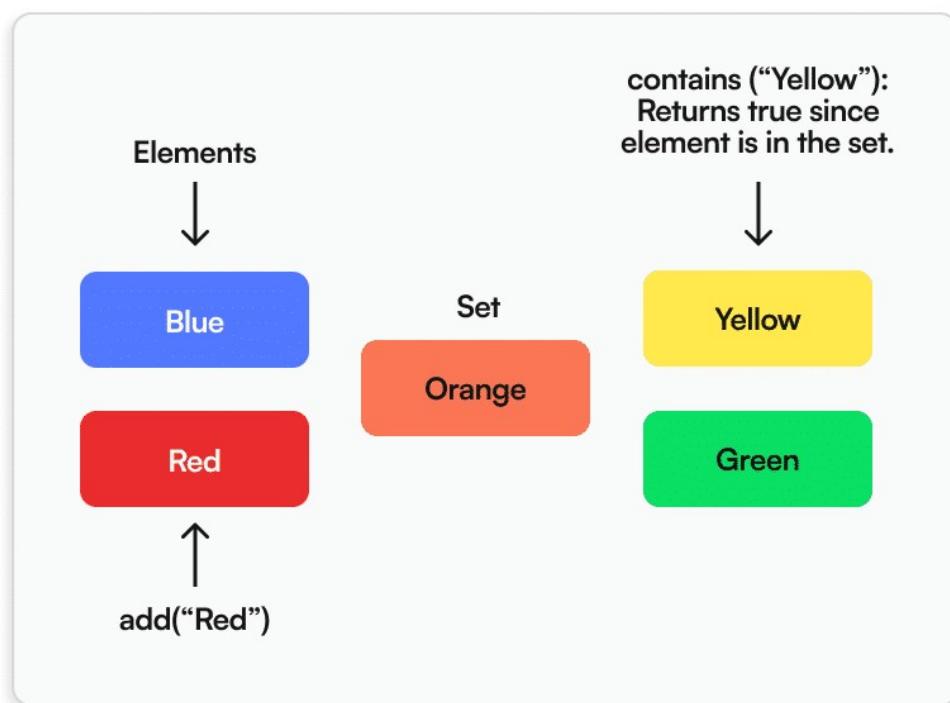
Imagínalo como una colección de objetos donde solo puedes tener una instancia de cada objeto en el Set. Si intentas agregar un elemento duplicado, simplemente se ignorará y no se agregará nuevamente.

La interfaz Set nos proporciona varios métodos para agregar, eliminar y buscar elementos en la colección. Algunos de los métodos más comunes son add(), remove() y contains(). También podemos realizar operaciones de conjunto como la unión, intersección y diferencia entre conjuntos utilizando métodos especiales. Los métodos más comunes que maneja son los mismos que la interfaz List<>, ya que sigue siendo una colección, pero tiene también métodos particulares a sus implementaciones.

Unordered set of books



Unordered collection of elements



Un **aspecto importante** de los **Sets** en **Java** es que no están ordenados, lo que significa que no podemos garantizar un orden específico de los elementos en el Set. Si necesitas mantener un orden particular, debes utilizar una implementación específica como **TreeSet**, que ordena automáticamente los elementos en función de su valor.

La elección de la implementación del Set dependerá de tus necesidades. Algunas implementaciones comunes son **HashSet**, **TreeSet** y **LinkedHashSet**. HashSet es la más eficiente en términos de tiempo de ejecución, mientras que TreeSet mantiene los elementos ordenados y LinkedHashSet mantiene el orden de inserción.

Los **Sets en Java** son útiles cuando queremos almacenar una colección de elementos únicos y no nos importa el orden de los elementos. Son eficientes para buscar elementos y garantizan que no haya duplicados en la colección.

Recuerda que para utilizar un **Set** en Java debemos importar la clase correspondiente y crear una instancia de ella. A partir de ahí, podemos utilizar los métodos proporcionados por la interfaz Set para agregar, eliminar y buscar elementos en el conjunto.

HashSet y **TreeSet** son implementaciones de la interfaz Set en Java que se utilizan para almacenar y manipular conjuntos de elementos únicos. Ambas clases tienen características distintas en términos de rendimiento y ordenamiento de elementos.

HashSet

- HashSet se basa en una tabla hash para almacenar los elementos.
- No garantiza un orden específico de los elementos almacenados.
- Ofrece un tiempo de inserción y búsqueda constante ($O(1)$) en promedio, lo que lo hace eficiente en términos de rendimiento.
- No permite elementos duplicados, ya que utiliza la función de hash para verificar la igualdad de los elementos.

Ejemplo:



- Ejemplo:

```
Set<String> hashSet = new HashSet<>();  
  
hashSet.add("Apple");  
hashSet.add("Banana");  
hashSet.add("Orange");  
hashSet.add("Apple"); // No se agregará, ya que es duplicado  
  
System.out.println(hashSet); // Output: [Orange, Banana, Apple]
```

TreeSet

- TreeSet implementa una estructura de árbol balanceado (generalmente un árbol rojo-negro) para almacenar los elementos.
- Mantiene los elementos ordenados de acuerdo con su orden natural o mediante un comparador personalizado.
- Ofrece un tiempo de inserción y búsqueda logarítmica ($O(\log n)$), lo que lo hace eficiente en términos de búsqueda.
- Permite elementos únicos y garantiza un orden ascendente de los elementos.

Ejemplo:



```
Set<Integer> treeSet = new TreeSet<>();  
  
treeSet.add(5);  
treeSet.add(3);  
treeSet.add(7);  
treeSet.add(1);  
  
System.out.println(treeSet); // Output: [1, 3, 5, 7]
```

Comparación entre Set y List

A continuación, veamos algunas de las características de cada uno para entender en qué se diferencian:

Característica	List	Set
Ordenamiento	Puede mantener un orden específico	No garantiza un orden específico
Elementos duplicados	Permite elementos duplicados	No permite elementos duplicados
Acceso por índice	Permite acceder a elementos por índice	No admite acceso por índice
Implementaciones comunes	ArrayList, LinkedList, Vector	HashSet, TreeSet, LinkedHashSet
Interfaz iterable	Sí	Sí

Características

Ordenamiento

List puede mantener un orden específico de los elementos, lo que significa que puedes acceder a ellos por índice y recuperarlos en el orden en que se agregaron. Por otro lado, **Set** no garantiza un orden específico de los elementos.

Elementos duplicados

List permite elementos duplicados, lo que significa que puedes tener múltiples elementos idénticos en la lista. En cambio, **Set** no permite elementos duplicados, lo que significa que cada elemento en un conjunto debe ser único.

Acceso por índice

List admite el acceso a elementos por índice utilizando el método `get()`, lo que te permite recuperar elementos específicos según su posición en la lista. Por otro lado, **Set** no admite el acceso por índice ya que no mantiene un orden específico.

Implementaciones comunes

List tiene implementaciones comunes como **ArrayList**, **LinkedList** y **Queue**. Estas implementaciones proporcionan diferentes formas de almacenar y acceder a los elementos de la lista. **Set**, por su parte, tiene implementaciones comunes como **HashSet**, **TreeSet** y **LinkedHashSet**, que ofrecen diferentes características de almacenamiento y operaciones de conjunto.

Interfaz iterable

Tanto **List** como **Set** son interfaces iterables, lo que significa que puedes recorrer sus elementos utilizando un bucle foreach o cualquier bucle de iteración.

Maps

Un Map en Java es una interfaz que representa una colección de pares clave-valor, donde cada clave es única y se utiliza para acceder a su correspondiente valor. En otras palabras, es una estructura de datos que nos permite asociar un valor con una clave específica.

Imagínalo como un diccionario, donde la clave es la palabra y el valor es su definición.

Podemos buscar una palabra en el diccionario utilizando la clave y obtener su significado correspondiente.

En un Map, cada par clave-valor se denomina "entrada". La interfaz Map nos proporciona métodos para agregar, eliminar y buscar entradas en el mapa. Algunos de los métodos más comunes son put(), remove() y get().

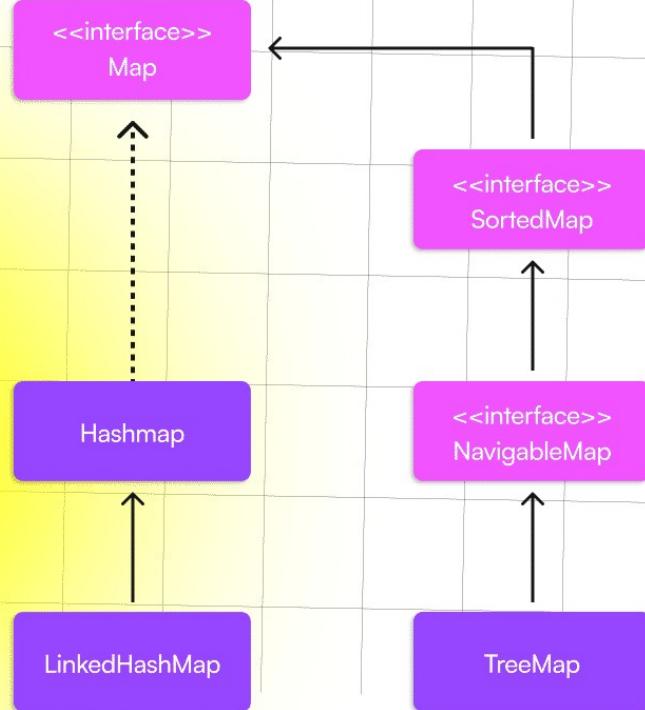


Una
característica
importante de
los Maps en

Java es que las claves deben ser únicas. Esto significa que no podemos tener dos entradas con la misma clave en el mapa. Si intentamos agregar una entrada con una clave que ya existe, el valor correspondiente se actualizará con el nuevo valor proporcionado.

Existen varias implementaciones de la interfaz Map en Java, como HashMap, TreeMap y LinkedHashMap. Cada implementación tiene sus propias características y ventajas, y la elección depende de los requisitos específicos del programa.

Map Interface



.....→ Implements

——→ Extends

HashMap

HashMap es la implementación más común y eficiente en términos de tiempo de ejecución. No garantiza un orden específico de las entradas. TreeMap mantiene las entradas ordenadas en función de las claves, mientras que LinkedHashMap mantiene el orden de inserción de las entradas .

```

import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Crear un objeto HashMap
        Map<String, Integer> hashMap = new HashMap<>();

        // Agregar elementos al HashMap
        hashMap.put("Manzana", 10);
        hashMap.put("Plátano", 5);
        hashMap.put("Naranja", 8);

        // Acceder a un elemento por su clave
        int cantidadManzanas = hashMap.get("Manzana");
        System.out.println("Cantidad de manzanas: " + cantidadManzanas);

        // Verificar si una clave existe en el HashMap
        boolean contienePera = hashMap.containsKey("Pera");
        System.out.println("¿Contiene pera? " + contienePera);
    }
}
  
```

```

// Iterar sobre los elementos del HashMap
for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
    String fruta = entry.getKey();
    int cantidad = entry.getValue();
    System.out.println(fruta + ": " + cantidad);
}

// Eliminar un elemento del HashMap
hashMap.remove("Plátano");

// Verificar el tamaño del HashMap
int tamaño = hashMap.size();
System.out.println("Tamaño del HashMap: " + tamaño);
}
}

```

Los Maps

Los **Maps** en Java son útiles cuando necesitamos **almacenar datos en forma de pares clave-valor y queremos acceder rápidamente a los valores utilizando las claves**. Son eficientes para buscar y actualizar valores utilizando las claves correspondientes.

Recuerda que para utilizar un Map en Java, debemos importar la clase correspondiente y crear una instancia de ella. A partir de ahí, podemos utilizar los métodos proporcionados por la interfaz Map para agregar, eliminar y buscar entradas en el mapa.

Para profundizar en estos conceptos, veamos una clase práctica [VER VIDEO HENRY MODULO 3](#)

Para seguir aprendiendo

En este espacio te compartimos material extra que te será de mucha ayuda para seguir fortaleciendo tu aprendizaje sobre JAVA.

Diferencias entre listas y set

Si quieras seguir aprendiendo sobre listas y set te invitamos a visitar este video

https://www.youtube.com/watch?v=VXrj3pqABzw&ab_channel=ProgramandoenJAVA

Listas

Si quieras seguir aprendiendo sobre listas te invitamos a visitar este video https://www.youtube.com/watch?v=yHFacwFar0A&ab_channel=Pa%C3%BAReyes

¿Qué es un paradigma funcional?

El paradigma funcional es un enfoque de programación que se basa en el uso de funciones puras, la inmutabilidad, la recursividad y el uso de funciones de primera clase.

Veamos brevemente cada uno de estos conceptos.

Funciones puras

Una función pura es aquella que siempre produce el mismo resultado cuando se le pasa la misma entrada y no tiene efectos secundarios. Esto significa que una función pura no modifica el estado de las variables externas ni realiza acciones adicionales más allá de calcular un resultado basado en sus entradas. Al ser predecibles y sin efectos secundarios, las funciones puras son más fáciles de razonar, depurar y probar.

Inmutabilidad

En el paradigma funcional, se evita cambiar el estado de las variables existentes. En lugar de eso, se evalúan expresiones y se crean nuevas estructuras de datos basadas en los datos originales. La inmutabilidad ayuda a evitar problemas de concurrencia y facilita la comprensión del código, ya que los datos no cambian inesperadamente.

Recursividad

En lugar de utilizar bucles iterativos, el paradigma funcional favorece el uso de funciones recursivas. Una función recursiva es aquella que se llama a sí misma con argumentos actualizados en cada iteración. Esto permite resolver problemas de manera elegante y concisa, y puede ser más intuitivo en ciertos casos.

Orden superior y funciones de primera clase:

En el paradigma funcional, las funciones se consideran ciudadanos de primera clase. Esto significa que se pueden pasar como argumentos a otras funciones, devuelven como resultados y se almacenan en variables. Esta capacidad de tratar a las funciones como datos permite una mayor modularidad y flexibilidad en el diseño de programas.

Ventajas

El paradigma funcional tiene varias ventajas. Al enfocarse en funciones puras y evitar el cambio de estado, el código se vuelve más legible, modular y fácil de probar. Además, al utilizar la recursividad y el orden superior, se pueden aplicar técnicas de abstracción más poderosas y expresivas.

En resumen

El paradigma funcional se basa en el uso de funciones puras, la inmutabilidad, la recursividad y las funciones de primera clase. Proporciona ventajas en términos de legibilidad, modularidad y facilidad de prueba. Es una alternativa poderosa al paradigma imperativo tradicional y puede ser especialmente útil en situaciones donde la concurrencia y la claridad del código son importantes.

Para reforzar estos conceptos, te invitamos a ver este video: <https://youtu.be/o-RE0qBGRRA>

La librería Stream API

La librería Stream API en Java nace como una forma de aplicar los principios de la programación funcional en el procesamiento de colecciones de datos. Proporciona una manera elegante, concisa y legible de trabajar con estos conjuntos de datos, permitiendo realizar operaciones como filtrado, mapeo, reducción y ordenamiento de manera más eficiente.

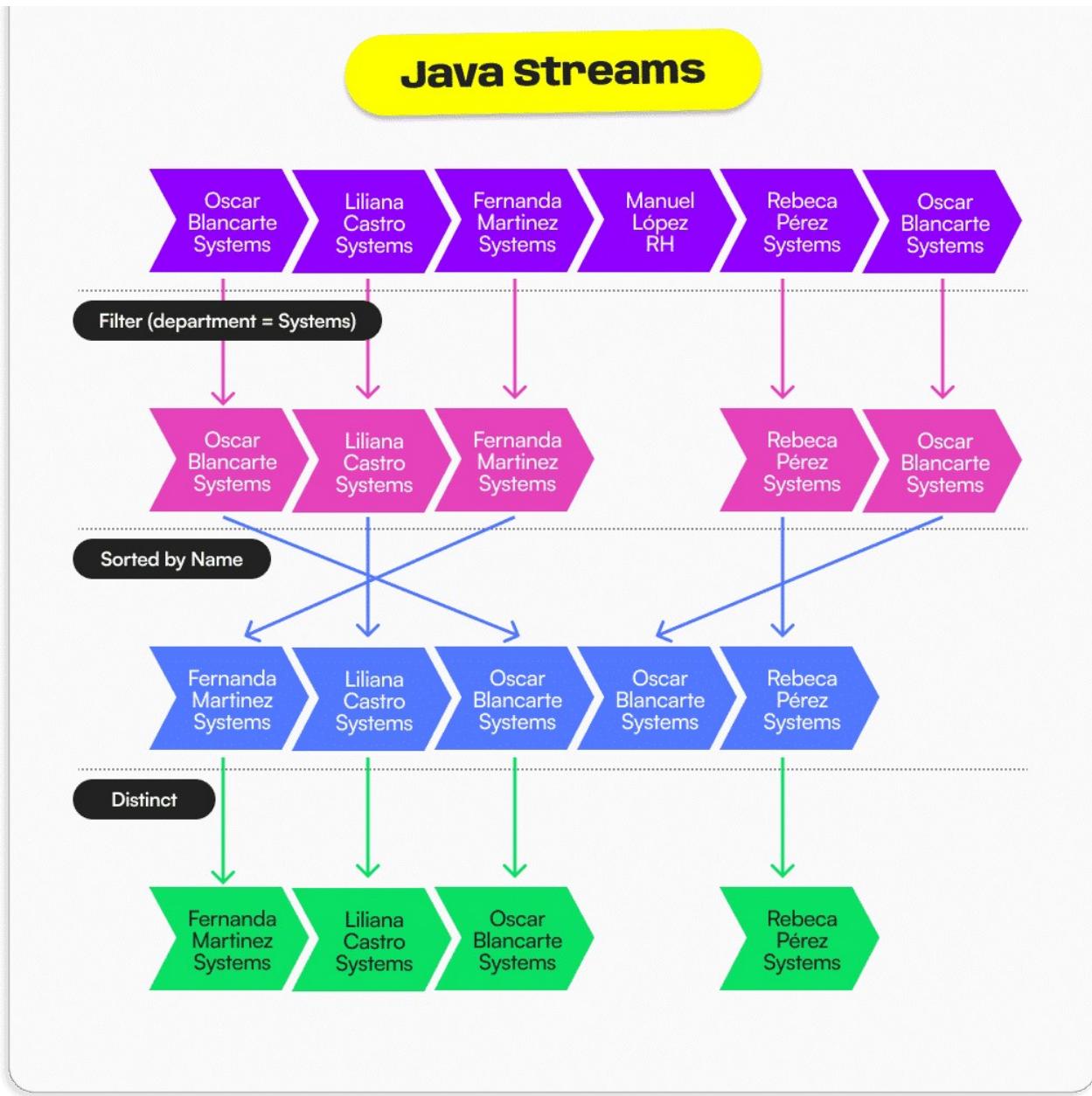
Ventajas de la librería Stream API

Código más legible y conciso

La Stream API permite escribir código de manera más clara y expresiva. Con su sintaxis fluida y funcional, puedes encadenar operaciones en forma de "pipes", lo que facilita la lectura y comprensión del código. Esto ayuda a reducir la cantidad de líneas de código necesarias para realizar operaciones comunes en colecciones de datos, lo que a su vez mejora la mantenibilidad y la legibilidad del código.

Operaciones eficientes y optimizadas

La Stream API de Java está diseñada para realizar operaciones de manera eficiente en grandes volúmenes de datos. Utiliza técnicas de procesamiento en paralelo y diferido (lazy evaluation) para optimizar el rendimiento y minimizar el uso de recursos. Esto se traduce en un mejor rendimiento y tiempos de ejecución más rápidos en comparación con enfoques tradicionales de iteración y manipulación de colecciones.



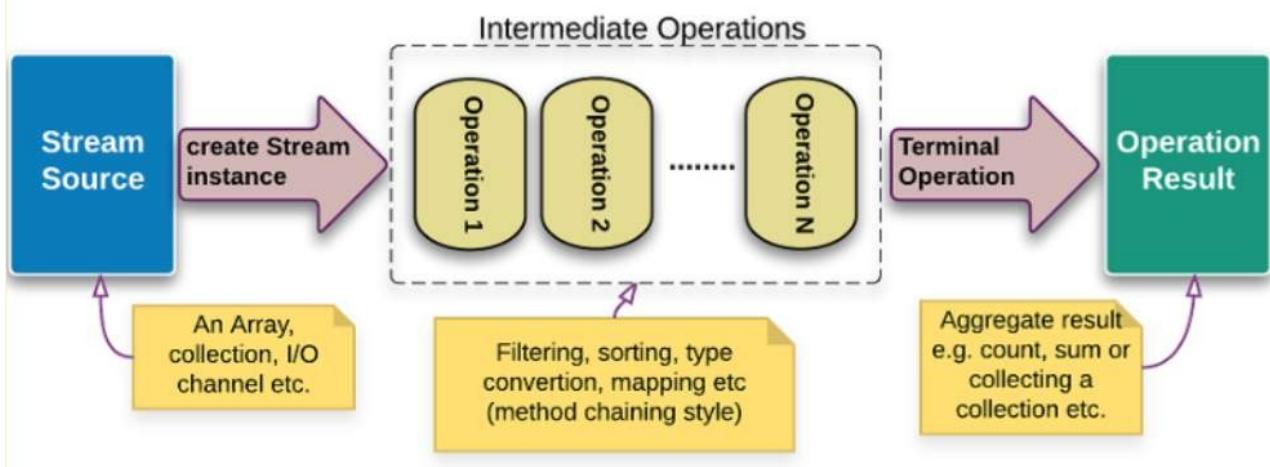
Flexibilidad y adaptabilidad

La Stream API proporciona una amplia gama de operaciones intermedias y terminales que puedes combinar y personalizar según tus necesidades. Puedes aplicar operaciones de filtrado, mapeo, ordenamiento, reducción y más de manera flexible y fácil de entender. Además, puedes utilizar lambdas y referencias a métodos para especificar el comportamiento de las operaciones, lo que te brinda una gran flexibilidad y adaptabilidad en la manipulación de tus datos.

Streams: concepto fundamental

Un concepto fundamental en la Stream API es el stream, que representa una secuencia de elementos que se pueden procesar de manera funcional. Un stream nos permite aplicar operaciones intermedias y terminales para transformar, filtrar y manipular los elementos en el stream.

Java Streams



Las operaciones intermedias son aquellas que se aplican a un stream y devuelven otro stream como resultado. Estas operaciones nos permiten realizar transformaciones y filtrados en los elementos del stream, sin afectar directamente a la fuente de datos original. Algunas operaciones intermedias comunes son el filtrado de elementos basado en una condición, la transformación de elementos mediante mapeo y la eliminación de duplicados.

Por otro lado, las operaciones terminales son aquellas que se aplican a un stream y producen un resultado final. Estas operaciones consumen completamente el stream y pueden devolver un valor concreto o realizar alguna acción final, como recopilar los elementos en una lista, calcular una suma, contar elementos, entre otros.

El uso de la Stream API permite escribir código más conciso y legible al combinar operaciones de manera encadenada, evitando la necesidad de bucles explícitos y reduciendo la cantidad de código necesario para realizar operaciones comunes en las colecciones de datos.

En resumen...

La Stream API en Java nos brinda una forma poderosa y funcional de trabajar con colecciones de datos. Los streams representan secuencias de elementos que se pueden procesar de manera funcional, y podemos aplicar operaciones intermedias y terminales para transformar, filtrar y manipular estos elementos. Esto nos permite escribir código más conciso, legible y eficiente al trabajar con colecciones en Java.

La función `stream()` y la estructura de pipes en una llamada funcional

Funciones intermedias y terminales básicas

El Stream en Java consta de una estructura en forma de "pipes" que encadenan las operaciones, lo que nos permite realizar una serie de transformaciones y filtrados en los elementos de una colección de manera secuencial y funcional.

Para ilustrar esto, podemos considerar un ejemplo básico utilizando un listado de nombres. Supongamos que tenemos la siguiente lista de nombres:

```
List<String> names = Arrays.asList("Juan", "María", "Pedro", "Ana", "Carlos");
```

A partir de esta lista, podemos aplicar diferentes operaciones utilizando la Stream API para realizar transformaciones y filtrados. Veamos algunas de estas operaciones:

Operaciones intermedias

- **`filter`**: Permite filtrar los elementos del stream basado en una condición. Por ejemplo, podemos filtrar los nombres que tienen más de 4 letras:

```
List<String> filteredNames = names.stream()
```

```
.filter(name -> name.length() > 4)
.collect(Collectors.toList());
```

- **`map`**: Permite transformar cada elemento del stream en otro valor. Por ejemplo, podemos convertir todos los nombres a mayúsculas:

```
List<String> uppercasedNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

- **`flatMap`**: Permite realizar una transformación más compleja en los elementos del stream. Por ejemplo, si tenemos una lista de listas de nombres, podemos aplanarla en un solo stream de nombres:

```
List<List<String>> listOfNames = Arrays.asList(Arrays.asList("Juan", "Pedro"),
    Arrays.asList("María", "Ana"));
List<String> flattenedNames = listOfNames.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
```

- **`limit`**: Permite limitar la cantidad de elementos en el stream. Por ejemplo, podemos obtener solo los primeros 3 nombres:

```
List<String> limitedNames = names.stream()
    .limit(3)
    .collect(Collectors.toList());
```

Operaciones terminales:

- **`collect`**: Permite recopilar los elementos del stream en una colección. Por ejemplo, podemos recopilar los nombres en una nueva lista:

```
List<String> collectedNames = names.stream()
    .collect(Collectors.toList());
```

- **`reduce`**: Permite combinar los elementos del stream en un único resultado. Por ejemplo, podemos concatenar todos los nombres en una sola cadena separada por comas:

```
Optional<String> reducedNames = names.stream()
    .reduce((name1, name2) -> name1 + ", " + name2);
```

- **`forEach`**: Permite realizar una acción en cada elemento del stream. Por ejemplo, podemos imprimir cada nombre en la consola:

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

- `anyMatch`: Permite verificar si al menos un elemento del stream cumple con una condición. Por ejemplo, podemos verificar si hay algún nombre que empiece con "A":

```
boolean anyNameStartsWithA = names.stream()
    .anyMatch(name -> name.startsWith("A"));
```

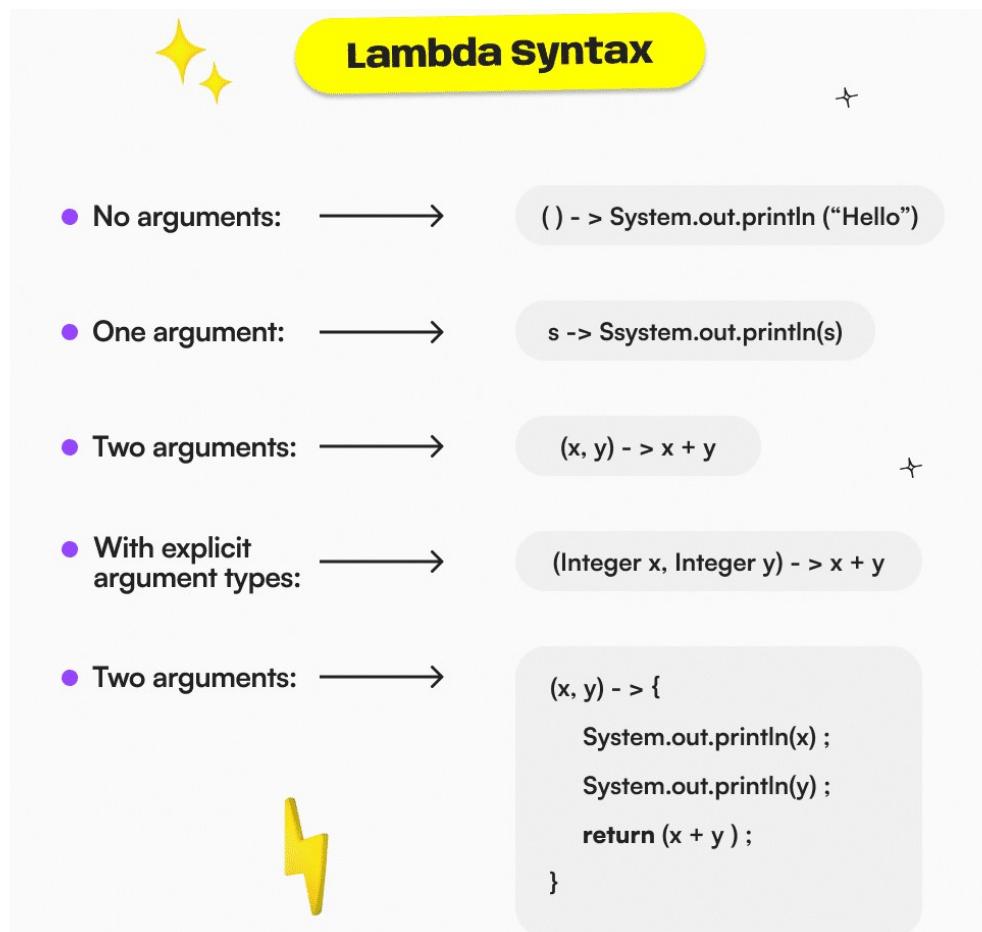
Estos son solo algunos ejemplos de las operaciones intermedias y terminales que podemos utilizar con la Stream API en Java. A medida que avancemos en el proyecto, podremos explorar más operaciones y su funcionamiento.

El siguiente video es un tutorial para seguir profundizando en este tema con más ejercicios:

<https://youtu.be/ACQtz4zpBLE>

Lambdas (Arrow functions)

Las lambdas, también conocidas como funciones flecha o arrow functions, son una característica de Java que nos permite definir funciones de manera concisa y expresiva. Son funciones anónimas que pueden tratarse como expresiones y ser pasadas como argumentos a métodos o almacenadas en variables. En otras palabras, las lambdas son bloques de código que pueden moverse de un lugar a otro en nuestro programa.



Las lambdas son especialmente útiles para implementar interfaces funcionales, como las que vimos en la lección 5 del módulo 1. La sintaxis básica de una lambda es la siguiente: `(parámetro) -> expresión`. En lugar de tener que escribir una función completa con nombre, parámetros y cuerpo, podemos definir una lambda de forma más compacta.

La Stream API se beneficia en gran medida del uso de lambdas, ya que nos permiten especificar de manera concisa el comportamiento de las operaciones intermedias y terminales que aplicamos a los streams.

En los ejemplos anteriores se mencionó cómo podemos resaltar cómo se utiliza la lambda. Por ejemplo, supongamos que queremos filtrar los nombres que tienen más de 4 letras utilizando una lambda:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>title</title>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
  </head>
  <body>
    <!-- page content -->
  </body>
</html>
```

Aquí, la lambda `name -> name.length() > 4` se utiliza para definir el criterio de filtrado, donde el parámetro `name` representa cada elemento del stream.

También podemos agregar otro ejemplo utilizando una lambda para el mapeo de nombres a mayúsculas:

```
List<String> uppercasedNames = names.stream()
  .map(name -> name.toUpperCase())
  .collect(Collectors.toList());
```

En este caso, la lambda `name -> name.toUpperCase()` se utiliza para definir la transformación que se aplicará a cada nombre del stream.

En resumen

Las lambdas nos permiten escribir código más conciso y legible al eliminar la necesidad de crear funciones completas con nombres y cuerpo definidos. Son una poderosa herramienta para trabajar con la Stream API y expresar de forma clara el comportamiento que deseamos aplicar a nuestros datos.

Además, veamos dos videos donde abordamos estos conceptos más en detalle:

- Primer video de la plataforma no esta disponible!!! lo reporte en Slack...
- segundo video: <https://youtu.be/e7zcj9A9yqU>

Homework

- **Crea** una aplicación para realizar operaciones sobre una lista de objetos Persona. Cada objeto Persona debe tener las siguientes propiedades: nombre, edad y hobby, todas de tipo String.
- **Utiliza** la Stream API de Java, realiza las siguientes operaciones sobre la lista de personas:
 - a) Filtrar las personas que sean mayores de 18 años y cuyos hobbies incluyan la palabra "programar". Utiliza la operación filter para realizar este filtrado.
 - b) Obtener una lista con los nombres de todas las personas. Utiliza la operación map para mapear cada objeto Persona a su nombre.
 - c) Limitar la lista de personas a un máximo de 5 personas. Utiliza la operación limit para establecer este límite.
 - d) Imprimir los nombres de todas las personas en la lista utilizando la operación forEach.
- **Crea** una clase principal llamada "PersonaManager" donde implementarás el código necesario para realizar estas operaciones.
- En el método main, crea una lista de objetos Persona con diferentes nombres, edades y hobbies. Puedes utilizar el constructor de la clase Persona para crear cada objeto.

- **Utiliza** los métodos y operaciones de la Stream API para realizar las operaciones mencionadas anteriormente.
- **Ejecuta** la aplicación y verifica que las operaciones se realicen correctamente, obteniendo los resultados esperados.
- **Recuerda** comentar tu código de forma adecuada y organizarlo de manera legible y estructurada.

Resultados en HW5.zip (LINK =>

https://drive.google.com/drive/folders/10CQn5YIeEl6zYotGkoEyYanZWJ2eJuzf?usp=share_link

Streams (Autor Victor orozco): https://www.youtube.com/watch?v=yjTfiO11tX8&ab_channel=V%C3%ADctorOrozco

Lamdas (autor Coding With John): https://www.youtube.com/watch?v=tj5sLSFjVj4&ab_channel=CodingwithJohn

¿Por qué necesitamos genéricos?

Los **genéricos** en **Java** ofrecen la **posibilidad de crear clases, interfaces y métodos** que pueden ser parametrizados con tipos específicos. Esta característica brinda **flexibilidad** y **reutilización de código**, permitiéndonos desarrollar componentes que funcionen con diferentes tipos de datos sin necesidad de duplicar código.

La idea central de los genéricos es permitir a los **desarrolladores crear estructuras de datos y algoritmos que sean independientes** del tipo de datos con el que se vaya a trabajar. Esto significa que podemos escribir una clase o método capaz de manejar diferentes tipos de datos de manera segura y coherente.

Es importante tener en cuenta que los genéricos **no funcionan con tipos de datos primitivos** en Java. Esto se debe a que los genéricos operan en tiempo de compilación y, en el fondo, el motor de Java "**borra**" los parámetros de tipo y los implementa como tipo Object. Por lo tanto, los genéricos en Java solo pueden trabajar con tipos de referencia.

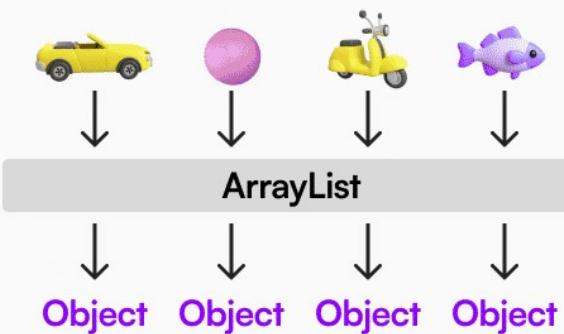
Como se puede observar en la figura anterior, sin el uso de genéricos, es posible colocar cualquier tipo de objeto en un ArrayList sin que el compilador de Java genere un error.

En

without generics

Objets go IN as a reference to Car, Football, Scooter, and Fish objects.

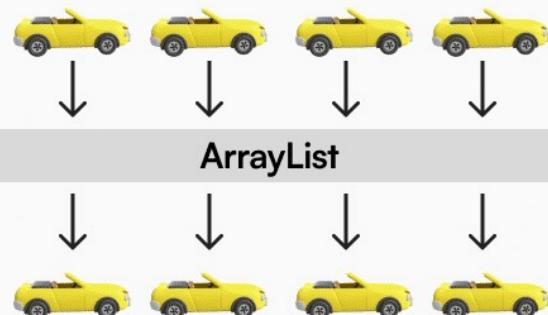
And come out as a reference of type Object.



without generics

Objets go IN as a reference to only Car objects.

And come OUT as a reference of type Car.



contraste, al utilizar genéricos, podemos restringir el tipo de elementos que se pueden añadir a un ArrayList, lo que resulta en una colección de tipo seguro. Esto permite detectar y resolver posibles problemas en tiempo de compilación en lugar de tiempo de ejecución.

Asimismo, el siguiente video te proporcionará más información sobre el uso de genéricos en Java y su importancia en la creación de colecciones seguras y reutilizables.

Ver video https://youtu.be/MFu8a_LpnIc

Tipos de genéricos: clases y métodos

Si no usásemos genéricos, podríamos tener una clase llamada `CajaDeHerramientas` con dos listas: `herramientasHerrero` y `herramientasCarpintero`.

A continuación, se muestra un ejemplo de cómo podríamos instanciar y usar esas listas sin genéricos:

```
public class CajaDeHerramientas {  
    private List<HerramientasHerrero> herramientasHerrero;  
    private List<HerramientasCarpintero> herramientasCarpintero;  
  
    public CajaDeHerramientas() {  
        herramientasHerrero = new ArrayList<>();  
        herramientasCarpintero = new ArrayList<>();  
    }  
  
    public void agregarHerramienta(Herramienta herramienta) {  
        if (herramienta instanceof HerramientasHerrero) {  
            herramientasHerrero.add((HerramientasHerrero) herramienta);  
        } else if (herramienta instanceof HerramientasCarpintero) {  
            herramientasCarpintero.add((HerramientasCarpintero) herramienta);  
        }  
    }  
  
    // Otros métodos de la clase...  
}
```

En este ejemplo, utilizamos el operador `instanceof` para validar el tipo de la herramienta y luego realizar un cast al tipo correspondiente antes de agregarlo a la lista correspondiente.

Ahora, podemos refactorizar este ejemplo utilizando genéricos para lograr el mismo resultado de una manera más elegante y segura. La sintaxis básica de los genéricos se define en la clase y en los métodos dentro de esa clase de la siguiente manera:

```
public class CajaDeHerramientas<T> {  
    private List<T> herramientas;  
  
    public CajaDeHerramientas() {  
        herramientas = new ArrayList<>();  
    }  
  
    public void agregarHerramienta(T herramienta) {  
        herramientas.add(herramienta);  
    }  
  
    // Otros métodos de la clase...  
}
```

En este caso, hemos utilizado el tipo genérico `T` para representar el tipo de herramienta. Esto nos permite crear una instancia de `CajaDeHerramientas` para cualquier tipo de herramienta y agregar herramientas directamente a la lista sin necesidad de validaciones o castings adicionales.

Aquí hay un ejemplo de cómo se usaría la clase `CajaDeHerramientas` con genéricos:

```
CajaDeHerramientas<HerramientasHerrero> cajaHerrero = new  
CajaDeHerramientas<>();  
cajaHerrero.agregarHerramienta(new HerramientasHerrero());  
  
CajaDeHerramientas<HerramientasCarpintero> cajaCarpintero = new  
CajaDeHerramientas<>();  
cajaCarpintero.agregarHerramienta(new HerramientasCarpintero());
```

En este ejemplo, hemos creado dos instancias de `CajaDeHerramientas` para diferentes tipos de herramientas y agregamos herramientas directamente a las listas correspondientes sin necesidad de validar tipos o realizar castings.

Los genéricos nos permiten definir clases y métodos que pueden trabajar con diferentes tipos de datos de manera segura y sin duplicar código.

En este video vas a ver un ejemplo del uso de Generics:

[**VER VIDEO 1 DE CLASE DE HENRY**](#)

Bounded Generics o genéricos restringidos

Al trabajar con **genéricos** en Java, podemos aplicar restricciones a los parámetros de tipo. Estas restricciones nos permiten definir límites en los tipos que pueden ser utilizados en una clase o método genérico. Dos tipos comunes de restricciones son el upper bound (límite superior) y el lower bound (límite inferior).

Upper bound (límite superior)

Al utilizar el upper bound, especificamos que el tipo de parámetro debe ser una clase en particular o cualquier subclase de esa clase. Esto se logra utilizando la palabra clave "extends" seguida del nombre de la clase o interfaz.

Ejemplo:

```
public class MiClase<T extends MiSuperClase> {  
    // Código de la clase  
}
```

En este ejemplo, el tipo de parámetro T se restringe a ser una clase que extienda o implemente MiSuperClase.

Lower bound (límite inferior)

Al utilizar el lower bound, especificamos que el tipo de parámetro debe ser una clase en particular o cualquier superclase de esa clase. Esto se logra utilizando el wildcard "?" junto con la palabra clave "super" seguida del nombre de la clase.

Ejemplo

```
public class MiClase<T super MiSuperClase> {  
    // Código de la clase  
}
```

En este ejemplo, el tipo de parámetro T se restringe a ser una clase que sea MiSuperClase o una superclase de MiSuperClase.

Estas restricciones nos permiten definir de manera más precisa los tipos que pueden ser utilizados en una clase o método genérico, lo que proporciona mayor flexibilidad y seguridad en la programación genérica.

Material extra genericos: <https://youtu.be/GKJl-4oNUWg>

¿Qué son los wildcards?

Los wildcards o comodines en Java se representan con el signo de interrogación '?'. Estos comodines se utilizan cuando queremos trabajar con tipos genéricos pero no conocemos el tipo específico al que se refieren.

El uso de wildcards en Java nos permite escribir código genérico más flexible y reutilizable al trabajar con tipos genéricos desconocidos o con límites superiores o inferiores específicos. Esto facilita la interoperabilidad y la adaptabilidad en el manejo de colecciones y componentes genéricos.

Ejemplo

```
public void miMetodo(List<?> lista) {  
    // Código del método  
}
```

En este ejemplo, el método miMetodo acepta una lista de cualquier tipo, pero el tipo exacto es desconocido. Esto permite que el método sea más genérico y pueda trabajar con diferentes tipos de listas.

Bounded Wildcards (Comodines con restricciones): Se utilizan para establecer restricciones en el tipo referido. Pueden ser de dos tipos: Upper Bounded Wildcards (comodines con límite superior) y Lower Bounded Wildcards (comodines con límite inferior).

Upper Bounded Wildcards: Se representan utilizando el keyword "extends" seguido del tipo límite superior. Esto permite que el tipo referido sea el tipo límite o cualquier subtipo de ese tipo.

Ejemplo

```
public void miMetodo(List<? extends Number> lista) {  
    // Código del método  
}
```

En este ejemplo, el método miMetodo acepta una lista de cualquier tipo que sea una subclase de Number. Esto significa que podemos pasar una lista de Integer, Double, Float, etc.

Lower Bounded Wildcards: Se representan utilizando el keyword "super" seguido del tipo límite inferior. Esto permite que el tipo referido sea el tipo límite o cualquier superclase de ese tipo.

Ejemplo

```
public void miMetodo(List<? super Integer> lista) {  
    // Código del método  
}
```

En este ejemplo, el método miMetodo acepta una lista de cualquier tipo que sea Integer o una superclase de Integer. Esto significa que podemos pasar una lista de Integer, Number, Object, etc.

El uso de wildcards nos brinda flexibilidad al trabajar con tipos genéricos y nos permite escribir código más genérico y reutilizable en situaciones donde no conocemos el tipo exacto al que se refiere.

VER VIDEO 2 DE CLASE DE HENRY

HOMEWORK

Crear una clase llamada Utilidades con 2 métodos para manipular listas:

- Un método será imprimirElementos() -> donde se recorrerá la lista, los elementos pueden ser de cualquier tipo .
- Otro método será copiarElementos() -> donde se deberá usar bounded generic para asegurar que la lista de destino sea de un tipo igual o superclase del tipo de la lista de origen .
- Además, utilizar una wildcard para indicar que la lista de origen puede ser de cualquier tipo.

Para probar esta clase genérica, realizar operaciones en el main, creando listas de diferentes tipos (Integer, String o creando alguna clase personalizada) y utilizando los métodos imprimirElementos() y copiarElementos()

ejercicio resuelto: https://drive.google.com/drive/folders/1akyPCEvOqQzy7ULY-noUXpT9Wm68M9oD?usp=share_link

Continuamos con el proyecto...

Durante el repaso de la última clase del módulo 2, es importante verificar que hayamos implementado correctamente los siguientes elementos en nuestro proyecto integrador:

- **Transición de arrays a colecciones:** Hemos refactorizado todas las manipulaciones de arrays en nuestra aplicación para que utilicen colecciones, ya sea List o Set. Esto nos permite aprovechar las ventajas de las operaciones y funcionalidades proporcionadas por las colecciones en Java.
- **Utilización de Map<K, V>:** Hemos implementado al menos un Map en nuestro proyecto, donde hemos utilizado una clave (K) y un valor (V) para almacenar y recuperar información de forma eficiente.
- **Llamadas funcionales:** Hemos incorporado al menos dos llamadas funcionales en nuestro código. Estas llamadas pueden ser utilizadas para realizar validaciones, iteraciones o mapeos en

nuestras colecciones. La utilización de funciones como filter(), map() o forEach() nos permite realizar estas operaciones de manera más concisa y legible.

- **Uso de clases o interfaces con genéricos:** Hemos implementado al menos una clase o interfaz en nuestro proyecto que utiliza genéricos. Esto nos brinda flexibilidad y reutilización de código al poder trabajar con diferentes tipos de datos sin tener que duplicar la implementación.

Para esta última etapa y cierre de módulo en nuestro Proyecto Integrador, ¡no nos olvidemos de nuestra última lección! vamos a implementar lo aprendido sobre **Generics**.

Debemos generar al menos un método en nuestro proyecto que acepte cualquier tipo de objeto para realizar alguna lógica específica.

Por ejemplo, un método para listar objetos, ya sean gastos o categorías. Esto nos permitirá trabajar con diferentes tipos de datos de manera genérica y adaptable.

A continuación, te dejamos una aproximación a cómo puedes implementar en el Proyecto Integrador todos los temas aprendidos hasta el momento. Pero ten en cuenta que este hito del PI es solo una idea base de cómo lo puedes hacer tú, ¡recuerda intentarlo por tu cuenta antes de ver esta posible solución!

VER VIDEO 3 HENRY CLASE 3

Modelo de Proyecto Integrador M2

Clíckea en "Proyecto" para acceder al archivo modelo de Proyecto Integrador de este primer módulo:

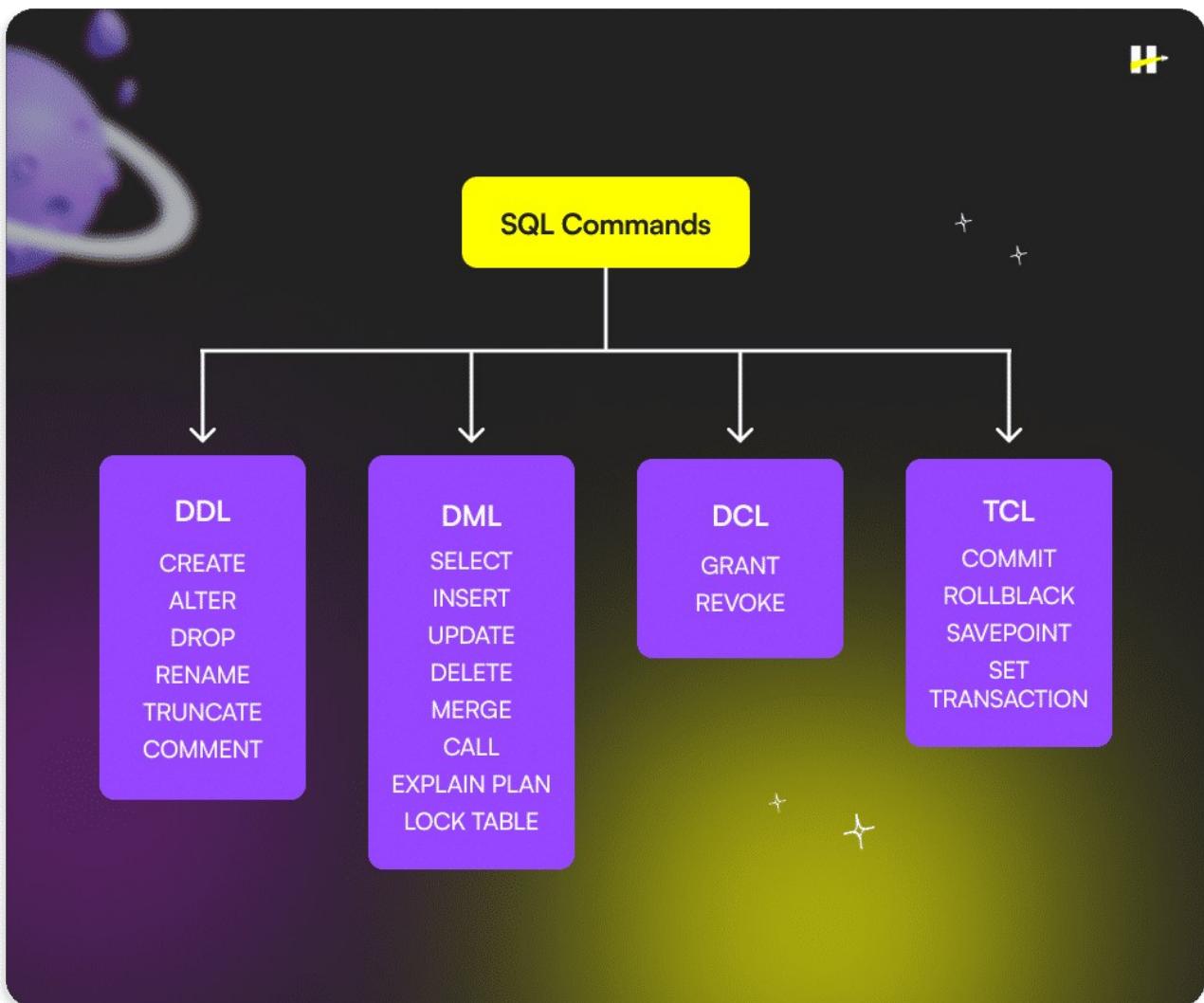
<https://drive.google.com/file/d/1SGRJgcEfBrHSHzG3cSrfhbxn3NEPDeFD/view?usp=sharing>

Material Extra: Generics in java <https://www.youtube.com/watch?v=h7piyWnQbZA>

Resumen de los comandos básicos de SQL

Data Definition Language (DDL) es un conjunto de comandos utilizados para definir y gestionar la estructura de la base de datos.

El grupo de comandos DDL más comunes son **CREATE, ALTER y DROP**, los cuales se utilizan para crear, modificar y eliminar objetos de la base de datos, como tablas, vistas, índices, etc. También se incluyen restricciones, como **PRIMARY KEY** y **FOREIGN KEY**, que permiten definir reglas y relaciones entre las tablas.



CREATE: Se utiliza para crear nuevos objetos en la base de datos. Por ejemplo, CREATE TABLE se utiliza para crear una nueva tabla en la base de dato

```
CREATE TABLE empleados (  
    id INT PRIMARY KEY,  
    nombre VARCHAR(50),  
    edad INT  
) ;
```

ALTER: Permite modificar la estructura de los objetos existentes en la base de datos. Por ejemplo, **ALTER TABLE** se utiliza para agregar, modificar o eliminar columnas de una tabla.

Ejemplo en H2:

```
ALTER TABLE empleados  
ADD COLUMN salario DECIMAL(10,2);
```

DROP: Se utiliza para eliminar objetos de la base de datos. Por ejemplo, DROP TABLE se utiliza para eliminar una tabla de la base de datos. Ejemplo en H2:

```
DROP TABLE empleados;
```

Las restricciones, como PRIMARY KEY y FOREIGN KEY, se utilizan para definir reglas y relaciones entre las tablas. Por ejemplo, PRIMARY KEY se utiliza para especificar una columna o un conjunto de columnas como clave primaria de una tabla, mientras que FOREIGN KEY se utiliza para establecer una relación entre dos tablas basada en una columna común.

Data Manipulation Language (DML)

Data Manipulation Language (DML) es un conjunto de comandos utilizados para manipular los datos en la base de datos. El grupo de comandos DML más comunes son INSERT INTO, UPDATE, DELETE FROM y SELECT.

INSERT INTO: Se utiliza para insertar nuevos registros en una tabla.

Ejemplo en H2:

```
INSERT INTO empleados (id, nombre, edad, salario)
VALUES (1, 'Juan Perez', 30, 5000);
```

UPDATE: Permite actualizar los datos existentes en una tabla.

Ejemplo en H2:

```
UPDATE empleados
SET salario = 6000
WHERE id = 1;
```

DELETE FROM: Se utiliza para eliminar registros de una tabla.

Ejemplo en H2:

```
DELETE FROM empleados
WHERE id = 1;
```

SELECT: Permite realizar consultas para recuperar datos de la base de datos.

Ejemplo en H2:

```
SELECT *
FROM empleados
```

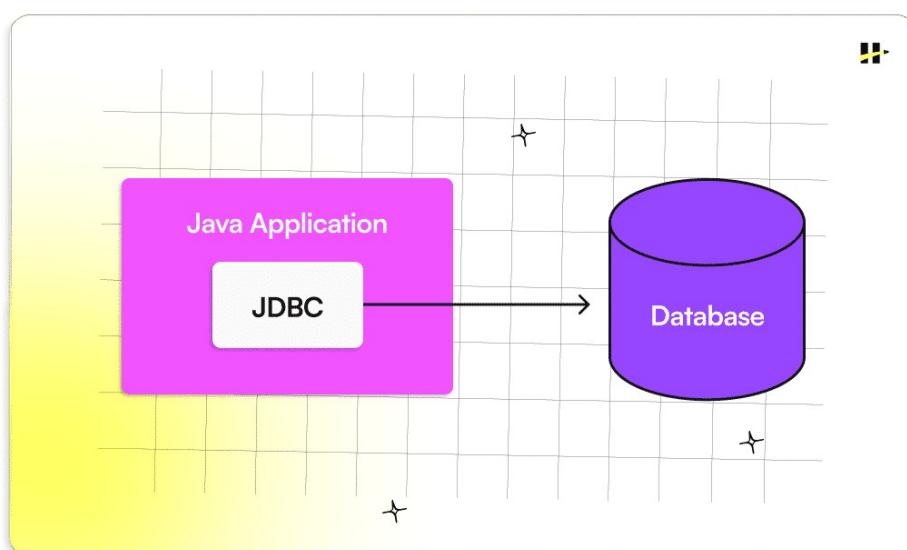
Estos son solo algunos ejemplos básicos de los comandos **DDL** y **DML** utilizados en **H2**. Cabe mencionar que cada sistema de gestión de bases de datos puede tener su propia sintaxis y características específicas, aunque los conceptos generales son similares.

¿Qué es JDBC? Mecanismos de Java para manipular una BD

Ahora que ya hemos repasado ejemplos básicos de comandos DDL y DML, la siguiente pregunta que podemos estar haciéndonos es, ¿cómo hacemos entonces para conectarnos con una base de datos desde nuestra aplicación de Java? Para eso existe la API de JDBC.

JDBC (Java Database Connectivity) es una API (Application Programming Interface) que proporciona un conjunto de interfaces y clases en Java para interactuar con bases de datos relacionales.

JDBC nos va a permitir acceder y manipular los datos almacenados en una base de datos a través de nuestro código Java.



La API JDBC consta de varios componentes clave:

En primer lugar, elementos clave en la **cadena de interacción**. Los drivers JDBC son componentes esenciales que actúan como **intermediarios** entre las aplicaciones programadas en Java y las diversas bases de datos con las que necesitan comunicarse.

En esencia, los drivers JDBC son como **traductores inteligentes** que permiten que las aplicaciones Java hablen el lenguaje de la base de datos, comprendiendo sus comandos y consultas, y luego traduciendo las respuestas de la base de datos en un formato que la aplicación pueda entender. Estos drivers establecen la **conexión**, gestionan las transacciones y facilitan la transferencia de datos entre el mundo de las aplicaciones y el reino de las bases de datos.

Los drivers JDBC desempeñan un papel crucial al proporcionar una **interfaz uniforme y estándar que ahorra a los desarrolladores el esfuerzo de aprender los detalles específicos** de cada base de datos con la que trabajan. Esto a su vez simplifica el proceso de desarrollo y facilita la portabilidad de las aplicaciones entre diferentes sistemas de gestión de bases de datos.

Existen cuatro tipos principales:

1. **Driver de tipo 1 (JDBC-ODBC Bridge):** Utiliza un controlador ODBC (Open Database Connectivity) para acceder a la base de datos. Es un puente entre JDBC y ODBC. Requiere la instalación de un controlador ODBC y no es recomendado para entornos de producción.
2. **Driver de tipo 2 (Controlador nativo parcial):** Utiliza código nativo para comunicarse directamente con la base de datos. Requiere la instalación de software adicional en el cliente y en el servidor de la base de datos.
3. **Driver de tipo 3 (Controlador de middleware):** Utiliza un servidor intermedio para comunicarse con la base de datos. El servidor intermedio traduce las llamadas JDBC a un protocolo específico de la base de datos.
4. **Driver de tipo 4 (Controlador JDBC puro):** Es un controlador completamente escrito en Java que se comunica directamente con la base de datos utilizando el protocolo de red de la base de datos. No requiere software adicional y es el tipo de controlador más comúnmente utilizado en entornos de producción.

Cada tipo de driver tiene sus propias características y requisitos de configuración. Es importante seleccionar el tipo de driver adecuado en función de las necesidades y requisitos del proyecto.

Luego, entramos en los componentes claves de desarrollo, clases e interfaces propias de la librería que nos permitirán codificar la conexión, acceso y manipulación de la base de datos.

Los componentes fundamentales que estaremos viendo son:

DriverManager

Es una clase que coordina la conexión con la base de datos y gestiona los controladores de JDBC. Permite establecer una conexión con una base de datos utilizando una URL de conexión y las credenciales necesarias.

Connection

Es una interfaz que representa una conexión con una base de datos. Se utiliza para establecer la comunicación con la base de datos, realizar consultas y transacciones, y administrar el estado de la conexión.

Statement

Es una interfaz que se utiliza para enviar consultas SQL a la base de datos y recibir los resultados. Permite ejecutar instrucciones SQL estáticas.

PreparedStatement

Es una subinterfaz de Statement que se utiliza para enviar consultas precompiladas a la base de datos. Proporciona mejor rendimiento y seguridad en comparación con las consultas estáticas.

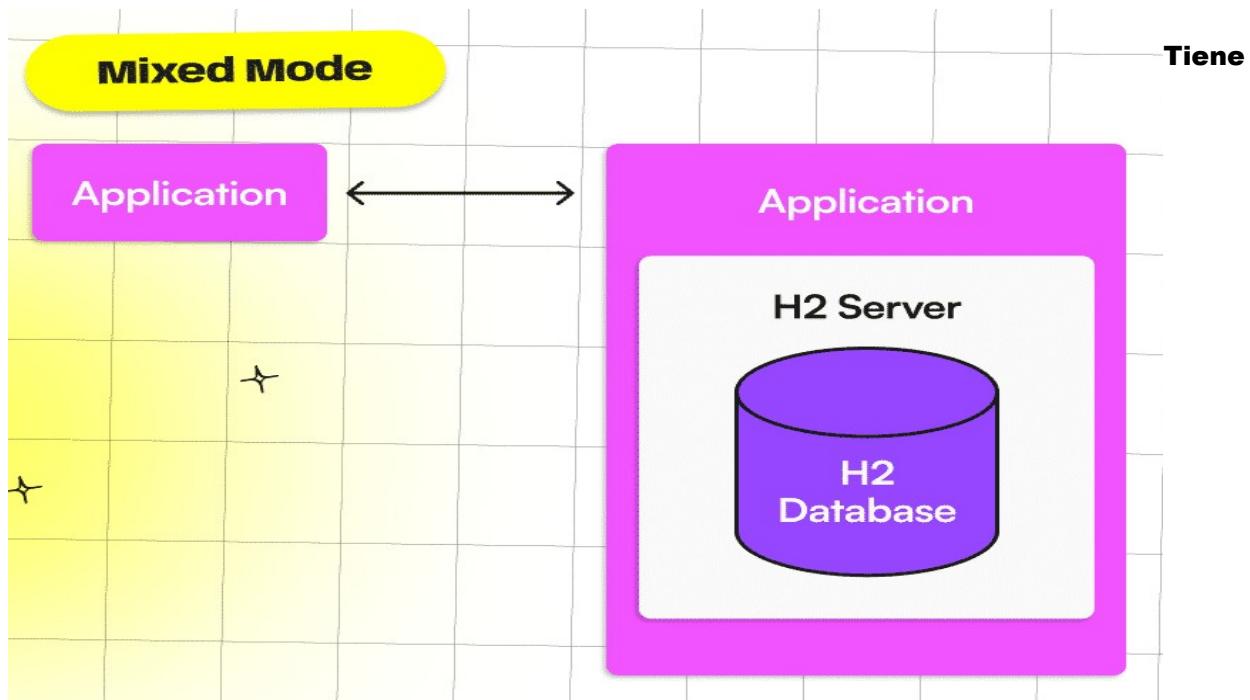
ResultSet

Es una interfaz que representa un conjunto de resultados de una consulta. Permite acceder y manipular los datos devueltos por una consulta.

En el **siguiente video** y los siguientes a ese vas a encontrar una explicación mas profunda sobre estos conceptos: <https://youtu.be/cFLsynl91B0>

H2 - Todo queda en casa: BD en memoria

H2 es una base de datos relacional escrita en Java, diseñada para ser rápida, ligera y fácil de usar.



varias características y ventajas que la hacen popular entre los desarrolladores:

Portabilidad

H2 puede ser fácilmente integrado en una aplicación Java ya que es una base de datos escrita en Java. No requiere una instalación separada y se puede embeber en la aplicación.

Rendimiento

H2 es conocida por su buen rendimiento, ofreciendo tiempos de respuesta rápidos y una alta eficiencia en la ejecución de consultas.

Soporte para múltiples modos de operación

H2 admite diferentes modos de operación, como el modo embebido, el modo cliente-servidor y el modo en memoria. Esto brinda flexibilidad para adaptarse a diferentes escenarios de desarrollo y despliegue.

Compatibilidad con estándares SQL

H2 cumple con los estándares SQL y ofrece soporte para una amplia gama de funciones y comandos SQL.

Ahora, vamos a aprender a configurar y utilizar H2 en una aplicación Java.

Ve la **siguiente clase** para conocer a detalle lo anterior (escanear QR para ir a video de henry!!)

Ahora que ya sabes lo básico, te dejamos también por escrito una guía para que puedas consultar y seguir a tu ritmo.

1. Agregar la dependencia: Asegúrate de tener la dependencia de H2 en tu proyecto.

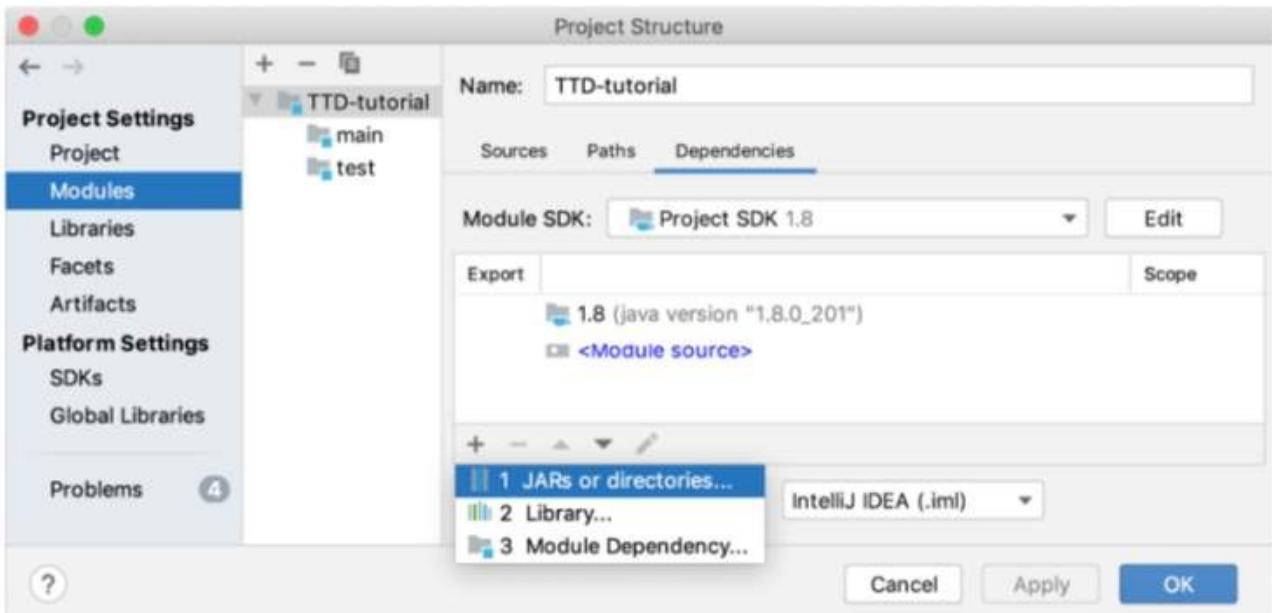
Desde el menú principal, selecciona Archivo | Estructura del Proyecto y haz clic en Módulos | Dependencias.

Haz clic en el botón Agregar y selecciona un tipo de dependencia:

Archivos JAR o directorios: selecciona un archivo Java Archive o un directorio desde los archivos en tu computadora.

Biblioteca: selecciona una biblioteca existente o crea una nueva y luego agrégala a la lista de dependencias.

Dependencia de módulo: selecciona otro módulo en el proyecto.



2. Configurar la conexión: En tu aplicación Java, configura los parámetros de conexión a la base de datos H2, como la URL de conexión, el nombre de usuario y la contraseña.

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Main {
    public static void main(String[] args) {
        // Configurar los parámetros de conexión
        String url = "jdbc:h2:~/test"; // URL de conexión a la base de datos H2
        String username = "usuario"; // Nombre de usuario de la base de datos
        String password = "contraseña"; // Contraseña de la base de datos

        try {
            // Establecer la conexión
            Connection connection = DriverManager.getConnection(url, username,
password);

            // Realizar operaciones en la base de datos

            // Cerrar la conexión
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

3. Crear la tabla e insertar registros: Utiliza la API de JDBC para crear una tabla en la base de datos H2 y luego insertar algunos registros. Puedes utilizar sentencias SQL para ejecutar estas operaciones.

Aquí tienes un ejemplo básico de cómo crear una tabla e insertar registros en H2 utilizando JDBC:

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;

public class H2Example {
    public static void main(String[] args) {
        try {
            // Establecer la conexión con la base de datos

```

```

Connection connection = DriverManager.getConnection("jdbc:h2:~/test", "username", "password");

        // Crear una declaración SQL
        Statement statement = connection.createStatement();

        // Crear la tabla
        String createTableQuery = "CREATE TABLE IF NOT EXISTS usuarios (id
INT PRIMARY KEY, nombre VARCHAR(50))";
        statement.executeUpdate(createTableQuery);
        // Insertar registros
        String insertQuery = "INSERT INTO usuarios VALUES (1, 'John Doe'), (2, 'Jane Smith')";
        statement.executeUpdate(insertQuery);
        // Cerrar la conexión
        statement.close();
        connection.close();
        System.out.println("Registros insertados con éxito en la tabla 'usuarios'.");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Este ejemplo crea una tabla llamada "usuarios" con dos columnas (id y nombre) y luego inserta dos registros en la tabla.

Recordemos que para esto es muy importante tener instalado el jdk. Puedes encontrar el tutorial de instalación aca: https://www.youtube.com/watch?v=kPWezAZGPks&ab_channel=LaGeekipediaDeErnesto

A partir de este punto, puedes continuar realizando operaciones de inserción, actualización, eliminación y consulta en la base de datos H2 utilizando la API de JDBC.

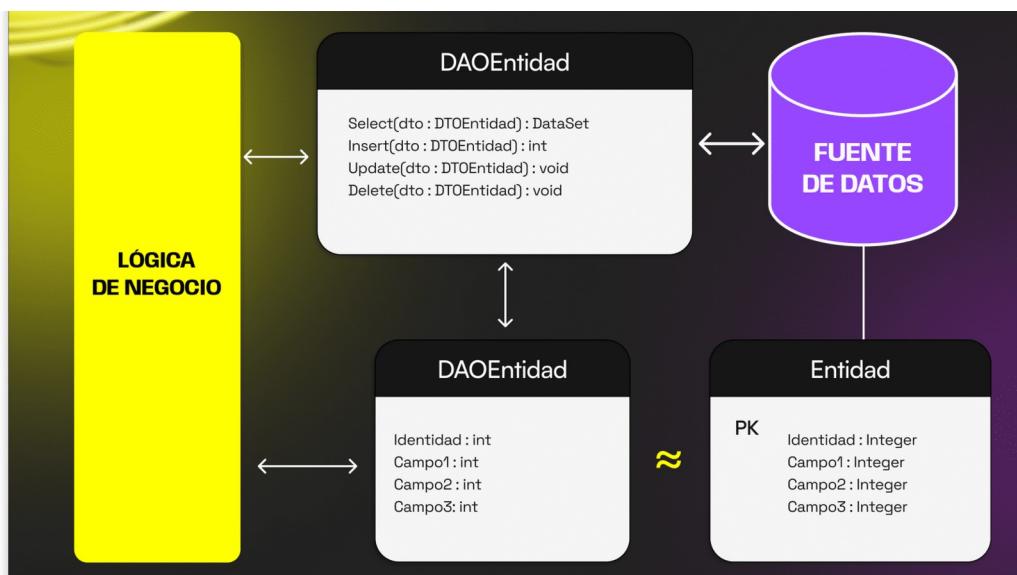
¿Qué es el patrón DAO?

Empezando por el principio...

El patrón DAO (Data Access Object) es un patrón de diseño que se utiliza en el desarrollo de aplicaciones para separar la lógica de acceso a datos de la lógica de negocio. Su intención principal es proporcionar una capa de abstracción entre la aplicación y la fuente de datos subyacente, como una base de datos, un servicio web o cualquier otro mecanismo de almacenamiento de datos.

El patrón DAO se basa en el principio de responsabilidad única, donde cada componente de la aplicación tiene una única responsabilidad. En este caso, el DAO se encarga específicamente de la manipulación y recuperación de datos, mientras que otras partes de la aplicación se centran en la lógica de negocio.

El objetivo del patrón DAO es



proporcionar una interfaz común y consistente para acceder y manipular los datos, independientemente de la fuente de datos subyacente. Esto permite que el resto de la aplicación no esté directamente acoplada a detalles específicos de almacenamiento, lo que facilita el mantenimiento y la flexibilidad de la aplicación.

Al utilizar el **patrón DAO**, se pueden definir interfaces o clases abstractas que representen las operaciones de acceso a datos, como crear, leer, actualizar y eliminar (CRUD). Luego, se implementan estas interfaces o clases abstractas para cada fuente de datos específica, como una implementación para una base de datos relacional y otra implementación para un servicio web.

En resumen, el **patrón DAO** proporciona una abstracción y encapsulación de la lógica de acceso a datos, lo que permite una mayor flexibilidad, mantenibilidad y reutilización del código en aplicaciones que interactúan con múltiples fuentes de datos.

En este video vas a encontrar una explicación más a fondo de patrón DAO:
<https://youtu.be/VVbTSkzhtA8>

Y aquí, un ejemplo: <https://youtu.be/ev-cNqp2bbI>

Para que necesitas al patrón DAO

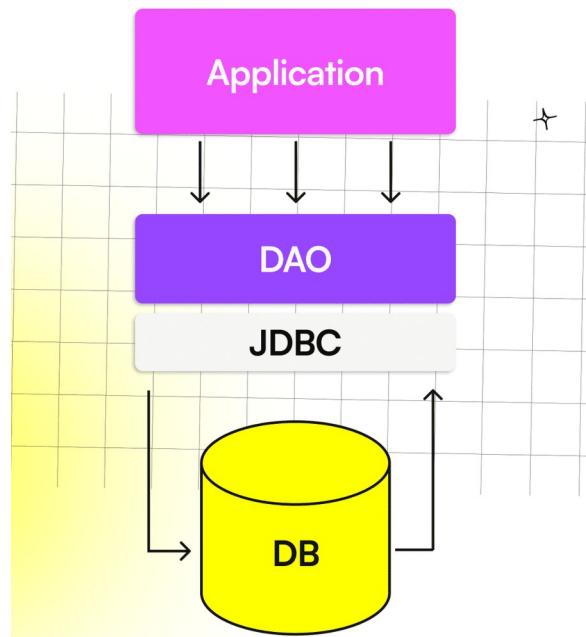
Ahora que sabemos lo que es el patrón dao veamos su utilidad

La **arquitectura en capas** es un enfoque común en el diseño de aplicaciones que busca separar las diferentes responsabilidades y funcionalidades de un sistema en distintas capas. Uno de los ejemplos más conocidos de arquitectura en capas es el patrón MVC (Modelo-Vista-Controlador), aunque existen otros enfoques similares como el patrón de 3 capas.

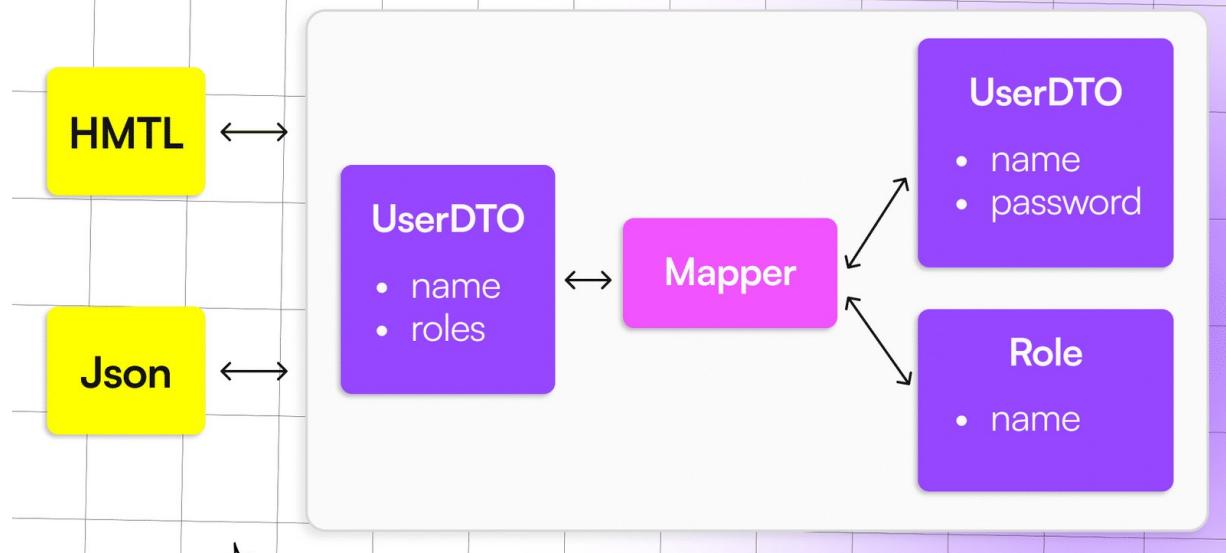
En la **arquitectura en capas**, cada capa tiene una responsabilidad específica y se comunica con las capas adyacentes a través de interfaces bien definidas. Esto permite lograr un mejor mantenimiento, escalabilidad y reutilización del código, ya que cada capa puede ser modificada o reemplazada sin afectar a las demás capas.

En el caso del patrón **DAO**, encaja en la capa de acceso a datos de la arquitectura en capas. El objetivo principal del patrón DAO es abstraer la lógica de acceso a datos de las demás capas de la aplicación. Los DAOs (Data Access Objects) se encargan de interactuar con la capa de almacenamiento de datos, ya sea una base de datos, un servicio web u otro mecanismo de almacenamiento. Proporcionan métodos para realizar operaciones CRUD (crear, leer, actualizar y eliminar) en los datos.

Los **DTOs (Data Transfer Objects)** son objetos utilizados para transferir datos entre las capas de la aplicación. Sirven como una representación de los datos que se intercambian entre las capas, evitando la exposición directa de las entidades de datos. Los DTOs suelen ser simples y contener solo los datos necesarios para la comunicación entre capas.



Presentation Layers



clases de entidad representan los objetos de dominio de la aplicación, es decir, los objetos que representan conceptos del mundo real con los que se trabaja en el sistema. Estas clases contienen la estructura y el comportamiento asociados con los datos de la aplicación.

En resumen, en la arquitectura en capas, el **patrón DAO** se sitúa en la capa de acceso a datos y se utiliza para separar la lógica de acceso a datos del resto de la aplicación. Los DTOs se utilizan para transferir datos entre las capas y las clases de entidad representan los objetos de dominio de la aplicación. Juntos, estos componentes contribuyen a la modularidad y organización de la aplicación, facilitando su mantenimiento y escalabilidad.

¿Cómo implementamos el patrón DAO?

¿CÓMO IMPLEMENTAR?

Para implementar el patrón DAO (Data Access Object) con un mapeo simple entre un DTO y una entidad en Java, puedes seguir los siguientes pasos:

Crea una interfaz DAO que defina los métodos de acceso a los datos. Esta interfaz actuará como una abstracción del acceso a la capa de persistencia. Por ejemplo:

```

```java
public interface UserDao {
 UserDto findById(int id);
 void save(UserDto userDto);
 void update(UserDto userDto);
 void delete(int id);
}
```
  
```

Crea una clase que implemente la interfaz DAO y se encargue de manejar la lógica de persistencia. En esta clase, realiza el mapeo entre el DTO y la entidad antes de realizar las operaciones de acceso a datos. Por ejemplo:

```

```java
public class UserDaoImpl implements UserDao {
 private UserRepository userRepository; // Repositorio que interactúa con la
 // capa de persistencia

 // Constructor u otras formas de inyección de dependencias para obtener el
 // repositorio UserRepository

 @Override
 public UserDto findById(int id) {
 UserEntity userEntity = userRepository.findById(id);
 ...
 }
}
```
  
```

```

        return mapEntityToDto(userEntity);
    }

    @Override
    public void save(UserDto userDto) {
        UserEntity userEntity = mapDtoToEntity(userDto);
        userRepository.save(userEntity);
    }

    @Override
    public void update(UserDto userDto) {
        UserEntity userEntity = mapDtoToEntity(userDto);
        userRepository.update(userEntity);
    }

    @Override
    public void delete(int id) {
        userRepository.delete(id);
    }

    // Métodos auxiliares para el mapeo entre el DTO y la entidad
    private UserDto mapEntityToDto(UserEntity userEntity) {
        // Lógica de mapeo
    }

    private UserEntity mapDtoToEntity(UserDto userDto) {
        // Lógica de mapeo
    }
}
```

```

Utiliza el DAO en tu aplicación para realizar las operaciones de acceso a datos. Por ejemplo:

```

```java
UserDao userDao = new UserDaoImpl();

UserDto userDto = userDao.findById(1);
userDto.setName("John Doe");
userDao.update(userDto);

UserDto newUserDto = new UserDto("Jane Smith", "jane@example.com");
userDao.save(newUserDto);
```

```

**En este video puedes ver un ejemplo en código: <https://youtu.be/NjY-WA-jeJ8>**

El patrón DAO te permite separar la lógica de persistencia de la capa de negocio y proporciona una abstracción para interactuar con los datos a través del DTO. El mapeo entre el DTO y la entidad se realiza en la implementación del DAO, lo que te permite utilizar la entidad internamente para las consultas y operaciones de persistencia, mientras que los DTO se utilizan para comunicarse con otras capas de la aplicación.

### Homework

**Para crear una base de datos para almacenar tarjetas de crédito y débito, puedes seguir los siguientes pasos:**

**1. Crear la base de datos: Genera un script para crear una base de datos y córrelo en H2.**

**Crear la tabla para tarjetas de crédito:** Ejecuta la correspondiente sentencia SQL para crear una tabla "TarjetasCredito" con tres campos: "id" (clave primaria), "numero" y "titular"

**Insertar registros en la tabla de tarjetas de crédito:** Ejecuta las correspondientes sentencias SQL para insertar tres registros de ejemplo en la tabla "TarjetasCredito".

**Crear la tabla para tarjetas de débito:** Ejecuta la correspondiente sentencia SQL para crear una tabla "TarjetasDebito" con tres campos: "id" (clave primaria), "número" y "titular".

**Insertar registros en la tabla de tarjetas de débito:** Ejecuta las correspondientes sentencias SQL para insertar tres registros de ejemplo en la tabla "TarjetasDebito".

**Genera una aplicación de Java y crea una clase de servicio que contenga 2 métodos:** getTarjetaCredito() y getTarjetaDebito(). Esta clase de servicio la deberás usar en el main.

Crea la estructura de interfaces y clases de implementación DAO para generar la lógica de conexión a BD con JDBC y los métodos de consulta (query) de ambas tablas.

Recuerda generar las entidades Java correspondientes a cada tabla y usar DTO's para la transmisión de los datos recuperados de BD.

Para este ejemplo, podemos pensar que el id de las tarjetas, es un dato sensible que no queremos mostrar por consola.

**Práctica de clase resuelta.** Aquí encontrarás los ejercicios resueltos:  
[https://drive.google.com/drive/folders/1yQp4c0kLBI-E0tdi6uxBcjUO1MkM-1wY?usp=share\\_link](https://drive.google.com/drive/folders/1yQp4c0kLBI-E0tdi6uxBcjUO1MkM-1wY?usp=share_link)

## Profundizamos en los pasos fundamentales para comenzar con JDBC

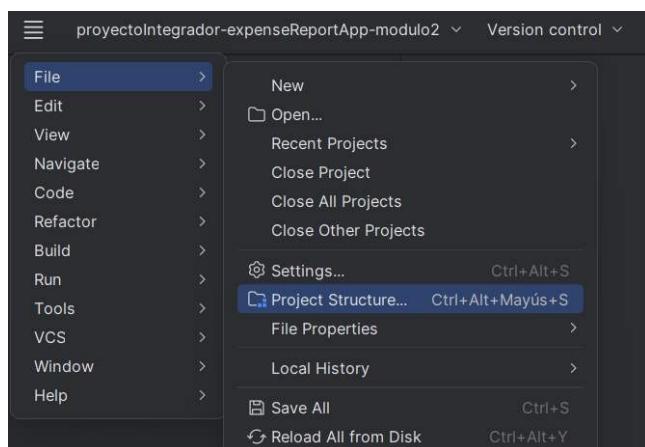
Primero, es importante tener en cuenta que la librería de JDBC forma parte del SDK de Java, por lo cual podremos importar sus clases e interfaces sin problemas y sin importar la versión de Java que estemos utilizando.

El siguiente paso importante es definir el motor de base de datos a utilizar. En nuestro caso, especificaremos el driver para H2. Luego, debemos importar la librería de H2, ya que esta es una API separada del core de Java.

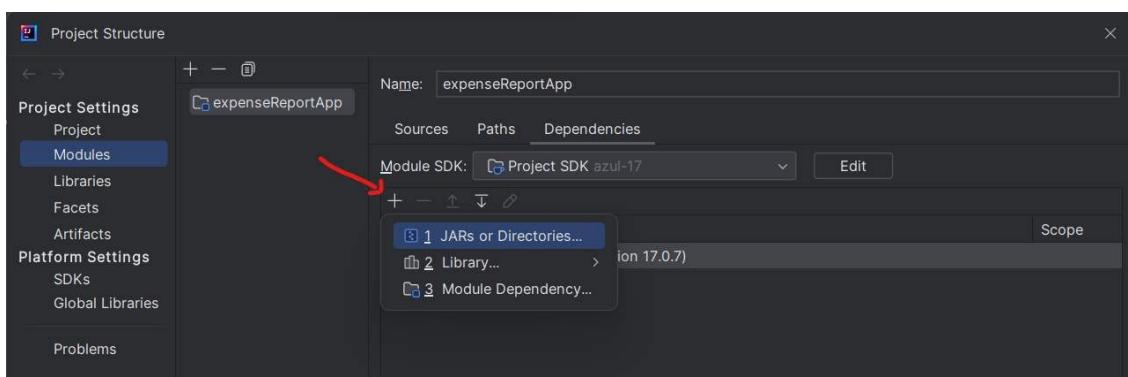
Para utilizar el motor de base de datos H2 es necesario importar el archivo JAR correspondiente que se encuentra en la carpeta "bin" de H2. Esta carpeta se ubica en la raíz de instalación de nuestro sistema operativo cuando descargamos y/o instalamos H2. Este archivo JAR contiene las clases y métodos necesarios para interactuar con la base de datos.

Debemos, por lo tanto, incorporar la librería al classpath de nuestra aplicación, es decir, al repositorio interno de donde la aplicación estará tomando e importando clases de Java. En ese classpath es en donde se encuentra el JDK y donde ubicaremos toda librería de Java que vayamos a utilizar.

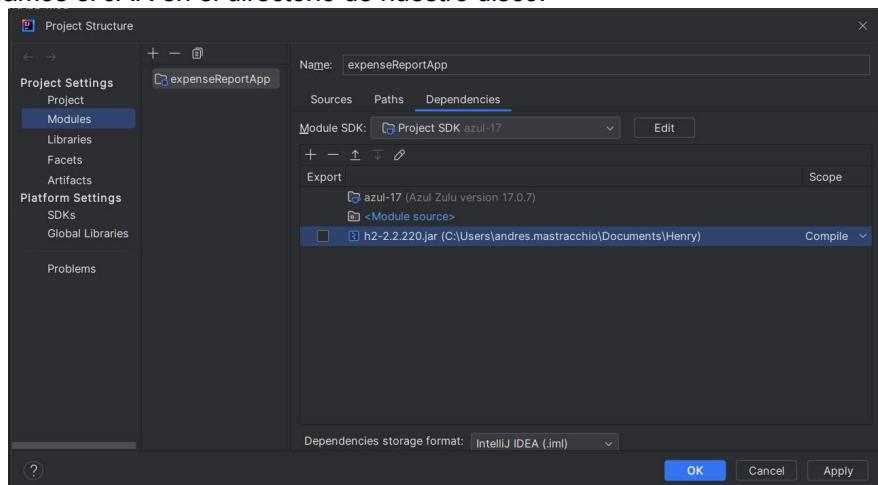
Esto lo hacemos de la siguiente manera:



Debemos dirigirnos a File -> Project Structure



Luego en la pestaña izquierda, ingresamos en Modules, hacemos click en el botón + y elegimos JARs or Directories, buscamos el JAR en el directorio de nuestro disco.



Una vez importado, damos click en Apply y nuestra aplicación podrá ya reconocer las clases propias de la librería de H2.

Ahora veamos un ejemplo de cómo empezar a usar dichas clases e interfaces para establecer una conexión y hacer statements (sentencias) usando JDBC.

Una vez establecida la conexión, se puede crear un objeto Statement. Un Statement es utilizado para ejecutar comandos SQL y enviar consultas a la base de datos. Permite realizar operaciones como la ejecución de consultas **SELECT, INSERT, UPDATE o DELETE**.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class Main {
 public static void main(String[] args) {
```

Establecer la conexión con la base de datos (la url, usuario, password)

La url la podemos obtener de la aplicación de H2, como vimos en la clase correspondiente

```
try {

 Connection connection =
DriverManager.getConnection("jdbc:h2:~/basedeempleados", "username",
"password");
```

Crear el objeto Statement

```
Statement statement = connection.createStatement();
```

Ejecutar una consulta INSERT

```
String query = "INSERT INTO empleados (id, nombre, salario) VALUES (1, 'Juan
Perez', 2000)";
int rowsAffected = statement.executeUpdate(query);
```

Verificar la cantidad de filas afectadas

```
System.out.println("Filas afectadas: " + rowsAffected);
```

Cerrar la conexión y el Statement

```
statement.close();
 connection.close();
} catch (SQLException e) {
 e.printStackTrace();
}
}
```

También se puede refactorizar el uso de Statement en un PreparedStatement. Un PreparedStatement es similar a un Statement, pero permite la ejecución de consultas parametrizadas. Esto proporciona seguridad y evita posibles ataques de inyección SQL.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Main {
 public static void main(String[] args) {
```

Establecer la conexión con la base de datos (la url, usuario, password)

La url la podemos obtener de la aplicación de H2, como vimos en la clase correspondiente

```
try {
 Connection connection =
DriverManager.getConnection("jdbc:h2:~/basedeempleados", "username",
"password");
```

Crear el objeto PreparedStatement

```
String query = "INSERT INTO empleados (id, nombre, salario) VALUES (?, ?, ?);
PreparedStatement preparedStatement =
connection.prepareStatement(query);
```

Asignar valores a los parámetros del PreparedStatement

```
preparedStatement.setInt(1, 1);
```

Valor para el primer parámetro (id)

```
preparedStatement.setString(2, "Juan Perez");
```

Valor para el segundo parámetro (nombre)

```
preparedStatement.setDouble(3, 2000.0);
```

Valor para el tercer parámetro (salario)

Ejecutar la consulta INSERT

```
int rowsAffected = preparedStatement.executeUpdate();
```

Verificar la cantidad de filas afectadas

```
System.out.println("Filas afectadas: " + rowsAffected);
```

Cerrar la conexión y el PreparedStatement

```
preparedStatement.close();
connection.close();
} catch (SQLException e) {
 e.printStackTrace();
}
}
```

Al ejecutar una consulta SELECT, se puede obtener un ResultSet que contiene los resultados de la consulta. Se puede iterar sobre el ResultSet para acceder a cada fila de datos y mostrarlos en la consola.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Main {
 public static void main(String[] args) {
 try {
 Connection connection =
DriverManager.getConnection("jdbc:h2:~/basedeempleados", "username",
"password");
```

Insertar un nuevo empleado (esto está basado en el ejemplo anterior)

```
String insertQuery = "INSERT INTO empleados (id, nombre, salario) VALUES (?, ?, ?);";
```

```

PreparedStatement insertStatement =
connection.prepareStatement(insertQuery);
insertStatement.setInt(1, 2);
insertStatement.setString(2, "Maria Lopez");
insertStatement.setDouble(3, 3000.0);
int rowsInserted = insertStatement.executeUpdate();
System.out.println("Filas insertadas: " + rowsInserted);
insertStatement.close();

```

Sumar los salarios de todos los empleados

```

String sumQuery = "SELECT SUM(salario) AS total_salario FROM empleados";
PreparedStatement sumStatement =
connection.prepareStatement(sumQuery);
ResultSet resultSet = sumStatement.executeQuery();

if (resultSet.next()) {
 double totalSalario = resultSet.getDouble("total_salario");
 System.out.println("Total de salarios: " + totalSalario);
}

resultSet.close();
sumStatement.close();
connection.close();
} catch (SQLException e) {
 e.printStackTrace();
}
}
}
}

```

Finalmente, es importante cerrar las conexiones con la base de datos para liberar los recursos utilizados. Esto se realiza mediante los métodos adecuados para cerrar la conexión, el Statement/PreparedStatement y el ResultSet.

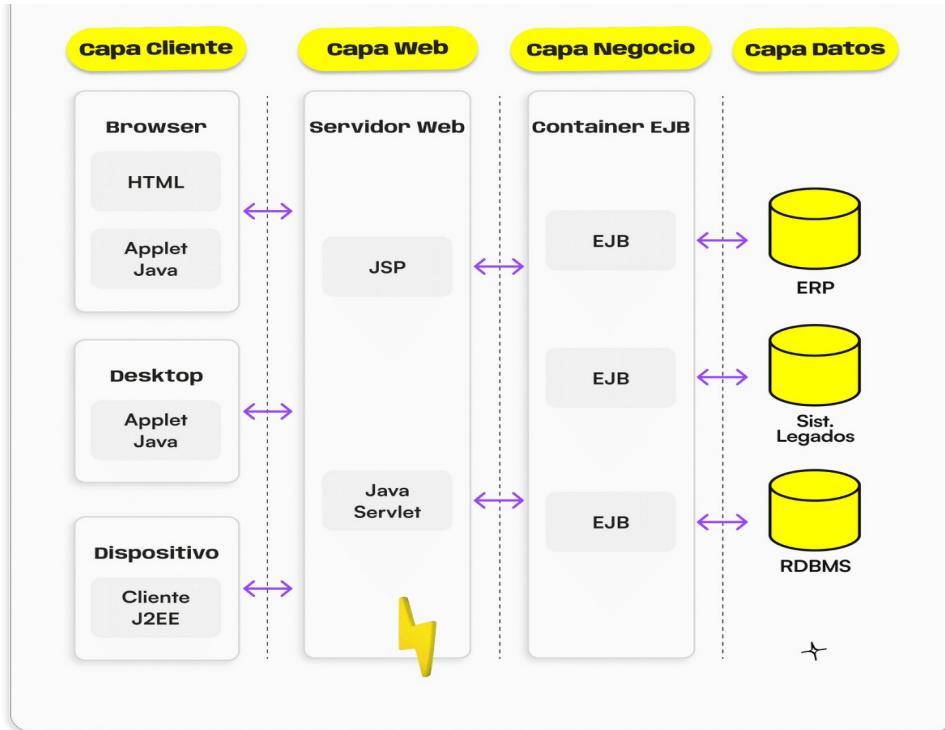
**En resumen, el proceso incluye** la elección de un **driver JDBC** adecuado, la importación del **JAR** correspondiente, la creación de una conexión, el uso de Statement y PreparedStatement para ejecutar **comandos SQL**, la iteración de resultados con ResultSet y, finalmente, el cierre de las conexiones para liberar los recursos.

**Veamos un video para entender mejor el concepto:** poner link cuando se pueda ver!!!!

### **Un poco sobre estructura de capas y... ¡comenzamos con el CRUD!**

En el desarrollo de aplicaciones Java, es común utilizar una arquitectura de capas para organizar y separar las responsabilidades del código. En nuestro proyecto por ejemplo, comenzamos a utilizar una división en capas, una de las cuales estructuramos gracias al patrón DAO, la capa de acceso a datos.

Exploraremos a continuación, un poco de algunas de las estructuras de capas más utilizadas en el desarrollo de aplicaciones Java...



### Capa de presentación (UI)

Esta capa se encarga de la interacción con el usuario. Aquí se implementa la interfaz de usuario y se gestionan las interacciones con los elementos visuales, como botones, formularios, etc.

### Capa de lógica de negocio

En esta capa se encuentra la lógica de negocio de la aplicación. Aquí se definen las reglas y procesos que determinan cómo se manipulan los datos y se realizan las operaciones específicas de la aplicación.

### Capa de acceso a datos

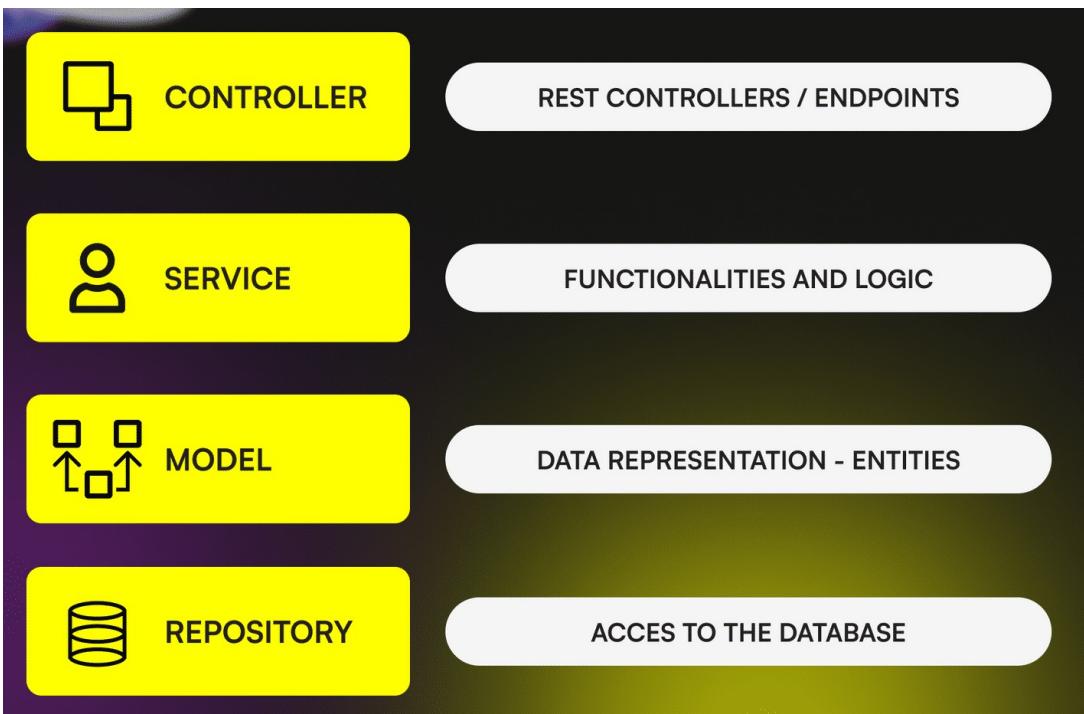
Esta capa se encarga de interactuar con la base de datos o cualquier otra fuente de datos. Aquí se definen las consultas y operaciones para recuperar, almacenar y actualizar los datos en la base de datos.

### Capa de servicios

En algunos casos, se utiliza una capa de servicios que actúa como intermediaria entre la capa de presentación y la capa de lógica de negocio. Esta capa proporciona servicios y funcionalidades específicas para la interacción entre la interfaz de usuario y la lógica de negocio.

### Capa de persistencia

Esta capa se encarga de la persistencia de los datos en la base de datos. Aquí se definen las entidades o clases que representan los datos y se implementa la lógica para almacenar y recuperar estos datos en la base de datos.



Estas estructuras de capas ayudan a organizar el código y separar las responsabilidades, lo que facilita el mantenimiento y la escalabilidad de la aplicación. Cada capa tiene su propio conjunto de clases y funciones específicas, lo que permite un desarrollo más modular y flexible.

### Implementación de CRUD

Un CRUD (Create, Read, Update, Delete) se refiere a las operaciones básicas de persistencia de datos en una base de datos. Estas operaciones permiten crear nuevos registros, leer o consultar registros existentes, actualizar registros existentes y eliminar registros de la base de datos. A continuación, se proporciona un ejemplo de un método en Java que utiliza un PreparedStatement para realizar una operación de inserción (CREATE) en una base de datos.

```
public class EjemploInsert {
 // Método para insertar un nuevo registro en la base de datos
 public void insertarRegistro(String nombre, int edad) {
 // Obtener la conexión a la base de datos (se asume que ya está establecida)
 Connection conn = obtenerConexion();

 String sql = "INSERT INTO tabla_usuarios (nombre, edad) VALUES (?, ?)";

 try {
 // Crear un PreparedStatement a partir de la sentencia SQL
 PreparedStatement stmt = conn.prepareStatement(sql);

 // Establecer los valores en el PreparedStatement
 stmt.setString(1, nombre);
 stmt.setInt(2, edad);

 // Ejecutar la inserción
 stmt.executeUpdate();

 // Cerrar el PreparedStatement
 stmt.close();

 System.out.println("Registro insertado exitosamente.");
 } catch (SQLException e) {
 System.out.println("Error al insertar el registro: " + e.getMessage());
 }
 }
}
```

```

 } finally {
 // Cerrar la conexión a la base de datos
 cerrarConexion(conn);
 }
 }
}

```

En este ejemplo, el método insertarRegistro recibe como parámetros el nombre y la edad del usuario que se desea insertar en la base de datos. Se crea una sentencia SQL con placeholders (?) para los valores. Luego, se crea un PreparedStatement a partir de la sentencia SQL y se establecen los valores en el PreparedStatement utilizando los métodos setString y setInt. Finalmente, se ejecuta la inserción llamando al método executeUpdate del PreparedStatement. Si ocurre algún error durante el proceso, se captura la excepción y se imprime un mensaje de error.

Es importante destacar que este es solo un ejemplo básico para ilustrar el uso de un PreparedStatement en una operación de inserción. En una aplicación real, se deberían considerar aspectos adicionales, como la gestión de excepciones, la validación de datos y la seguridad en la manipulación de la base de datos.

Veamos un ejemplo... link video 2 que tapoco anda

### **Read - obteniendo información de la BD**

Veamos a continuación, un ejemplo de un método en Java que utiliza JDBC para realizar una operación de lectura (READ) en una base de datos utilizando una sentencia SELECT.

```

public class EjemploRead {

 // Método para obtener un registro por su ID
 public void obtenerRegistroPorId(int id) {
 // Obtener la conexión a la base de datos (se asume que ya está
 establecida)
 Connection conn = obtenerConexion();

 // Definir la sentencia SQL con un placeholder (?) para el ID
 String sql = "SELECT * FROM tabla_usuarios WHERE id = ?";

 try {
 // Crear un PreparedStatement a partir de la sentencia SQL
 PreparedStatement stmt = conn.prepareStatement(sql);

 // Establecer el valor del ID en el PreparedStatement
 stmt.setInt(1, id);

 // Ejecutar la consulta y obtener el resultado en un ResultSet
 ResultSet rs = stmt.executeQuery();

 // Iterar sobre los registros devueltos por el ResultSet
 while (rs.next()) {
 int registroId = rs.getInt("id");
 String nombre = rs.getString("nombre");
 int edad = rs.getInt("edad");

 System.out.println("ID: " + registroId);
 System.out.println("Nombre: " + nombre);
 System.out.println("Edad: " + edad);
 System.out.println("-----");
 }

 // Cerrar el ResultSet y el PreparedStatement
 rs.close();
 stmt.close();
 } catch (SQLException e) {
 System.out.println("Error al obtener el registro: " +
e.getMessage());
 }
 }
}

```

```

 } finally {
 // Cerrar la conexión a la base de datos
 cerrarConexion(conn);
 }
 }
}

```

En este ejemplo, el método obtenerRegistroPorId recibe como parámetro el ID del registro que se desea obtener de la base de datos. Se crea una sentencia SQL con un placeholder (?) para el ID. Luego, se crea un PreparedStatement a partir de la sentencia SQL y se establece el valor del ID en el PreparedStatement utilizando el método setInt. Después, se ejecuta la consulta llamando al método executeQuery del PreparedStatement y se obtiene el resultado en un ResultSet. Se itera sobre los registros devueltos por el ResultSet utilizando el método next y se obtienen los valores de cada columna utilizando los métodos getInt y getString. Finalmente, se imprime la información de cada registro y se cierran el ResultSet y el PreparedStatement.

Nuevamente, es importante mencionar que este es solo un ejemplo básico para ilustrar el uso de un PreparedStatement en una operación de lectura. En una aplicación real, se deberían considerar aspectos adicionales, como la gestión de excepciones, la validación de datos y la seguridad en la manipulación de la base de datos.

**Veamos un ejemplo...** video 3 que no anda

### Update - modificando y actualizando nuestra BD

Ahora veamos un ejemplo de un método en Java que utiliza JDBC para realizar una operación de actualización (UPDATE) en una base de datos utilizando una sentencia UPDATE.

```

java
Copy code
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class EjemploUpdate {

 // Método para actualizar un registro por su ID
 public void actualizarRegistroPorId(int id, String nuevoNombre, int
nuevaEdad) {
 // Obtener la conexión a la base de datos (se asume que ya está
establecida)
 Connection conn = obtenerConexion();

 // Definir la sentencia SQL con los placeholders (?) para los nuevos
valores
 String sql = "UPDATE tabla_usuarios SET nombre = ?, edad = ? WHERE id =
?";

 try {
 // Crear un PreparedStatement a partir de la sentencia SQL
 PreparedStatement stmt = conn.prepareStatement(sql);

 // Establecer los nuevos valores en el PreparedStatement
 stmt.setString(1, nuevoNombre);
 stmt.setInt(2, nuevaEdad);
 stmt.setInt(3, id);

 // Ejecutar la actualización
 int filasAfectadas = stmt.executeUpdate();

 // Verificar si la actualización fue exitosa
 if (filasAfectadas > 0) {
 System.out.println("El registro se actualizó exitosamente.");
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 }
}

```

```

 } else {
 System.out.println("No se encontró el registro especificado.");
 }

 // Cerrar el PreparedStatement
 stmt.close();
 } catch (SQLException e) {
 System.out.println("Error al actualizar el registro: " +
e.getMessage());
 } finally {
 // Cerrar la conexión a la base de datos
 cerrarConexion(conn);
 }
}

// Método para obtener la conexión a la base de datos
private Connection obtenerConexion() {
 // Implementación para obtener la conexión (omitida en el ejemplo)
 // ...
}

// Método para cerrar la conexión a la base de datos
private void cerrarConexion(Connection conn) {
 // Implementación para cerrar la conexión (omitida en el ejemplo)
 // ...
}
}

```

En este ejemplo, el método actualizarRegistroPorId recibe como parámetros el ID del registro que se desea actualizar, el nuevo nombre y la nueva edad. Se crea una sentencia SQL con placeholders (?) para los nuevos valores. Luego, se crea un PreparedStatement a partir de la sentencia SQL y se establecen los nuevos valores utilizando los métodos setString y setInt. Después, se ejecuta la actualización llamando al método executeUpdate del PreparedStatement y se obtiene el número de filas afectadas. Se verifica si la actualización fue exitosa comparando el número de filas afectadas con cero. Finalmente, se cierra el PreparedStatement.

Nuevamente, es importante mencionar que este es solo un ejemplo básico para ilustrar el uso de un PreparedStatement en una operación de lectura. En una aplicación real, se deberían considerar aspectos adicionales, como la gestión de excepciones, la validación de datos y la seguridad en la manipulación de la base de datos.

Veamos un ejemplo: video 4 que no anda

### **Delete - Eliminando registros y tablas de la BD**

A continuación, te mostraré un ejemplo de un método en Java que utiliza JDBC para realizar una operación de eliminación (DELETE) en una base de datos utilizando una sentencia DELETE.

```

java
Copy code
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class EjemploRemove {

 // Método para eliminar un registro por su ID
 public void eliminarRegistroPorId(int id) {
 // Obtener la conexión a la base de datos (se asume que ya está
establecida)
 Connection conn = obtenerConexion();

 // Definir la sentencia SQL con el placeholder (?) para el ID
 String sql = "DELETE FROM tabla_usuarios WHERE id = ?";
 }
}

```

```

try {
 // Crear un PreparedStatement a partir de la sentencia SQL
 PreparedStatement stmt = conn.prepareStatement(sql);

 // Establecer el ID en el PreparedStatement
 stmt.setInt(1, id);

 // Ejecutar la eliminación
 int filasAfectadas = stmt.executeUpdate();

 // Verificar si la eliminación fue exitosa
 if (filasAfectadas > 0) {
 System.out.println("El registro se eliminó exitosamente.");
 } else {
 System.out.println("No se encontró el registro especificado.");
 }

 // Cerrar el PreparedStatement
 stmt.close();
} catch (SQLException e) {
 System.out.println("Error al eliminar el registro: " +
e.getMessage());
} finally {
 // Cerrar la conexión a la base de datos
 cerrarConexion(conn);
}
}

// Método para obtener la conexión a la base de datos
private Connection obtenerConexion() {
 // Implementación para obtener la conexión (omitida en el ejemplo)
 // ...
}

// Método para cerrar la conexión a la base de datos
private void cerrarConexion(Connection conn) {
 // Implementación para cerrar la conexión (omitida en el ejemplo)
 // ...
}
}

```

En este ejemplo, el método `eliminarRegistroPorId` recibe como parámetro el ID del registro que se desea eliminar. Se crea una sentencia SQL con un placeholder (?) para el ID. Luego, se crea un `PreparedStatement` a partir de la sentencia SQL y se establece el ID utilizando el método `setInt`. Después, se ejecuta la eliminación llamando al método `executeUpdate` del `PreparedStatement` y se obtiene el número de filas afectadas. Se verifica si la eliminación fue exitosa comparando el número de filas afectadas con cero. Finalmente, se cierra el `PreparedStatement`.

Al igual que en los ejemplos anteriores, es importante mencionar que este es solo un ejemplo básico para ilustrar el uso de un `PreparedStatement` en una operación de eliminación. En una aplicación real, se deben considerar aspectos adicionales, como la gestión de excepciones, la validación de datos y la seguridad en la manipulación de la base de datos.

Video 5 que no anda

## Homework

**La tarea consiste en crear una tabla llamada "Empleados" en la base de datos H2 con tres campos: "id" (entero, clave primaria), "nombre" (cadena de texto) y "salario" (decimal).**

Deberás implementar una clase Java llamada "Empleado" que represente a un empleado y contenga propiedades correspondientes a los campos de la tabla. Además, deben crear otra clase que contenga los métodos necesarios para interactuar con la base de datos utilizando JDBC. Estos métodos incluyen establecer la conexión con la base de datos, insertar un nuevo empleado, actualizar la información de un empleado existente, eliminar un empleado y obtener y mostrar todos los empleados en la consola.

En la clase principal (Main), se debe crear una instancia de la clase de interacción con la base de datos y utilizar los métodos para realizar acciones como insertar empleados, actualizar el salario de un empleado, eliminar un empleado, obtener y mostrar todos los empleados en la consola.

Como tarea adicional (bonus), se pide refactorizar la tarea de la clase anterior para acceder a los datos realizando el CRUD correspondiente y conectándolo con la base de datos creada en la clase anterior. Esto implica realizar la configuración y refactorización necesarias.

### **Práctica de clase resuelta**

Aquí encontrarás los ejercicios resueltos:

[https://drive.google.com/file/d/1A\\_zfgSGr8NipT7GI7ox5NXmklsFVe\\_gJ/view?usp=share\\_link](https://drive.google.com/file/d/1A_zfgSGr8NipT7GI7ox5NXmklsFVe_gJ/view?usp=share_link)