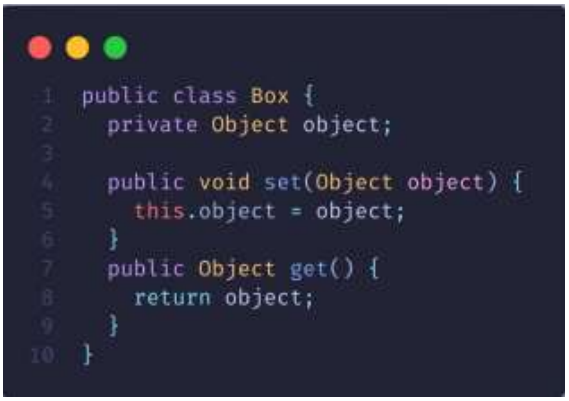
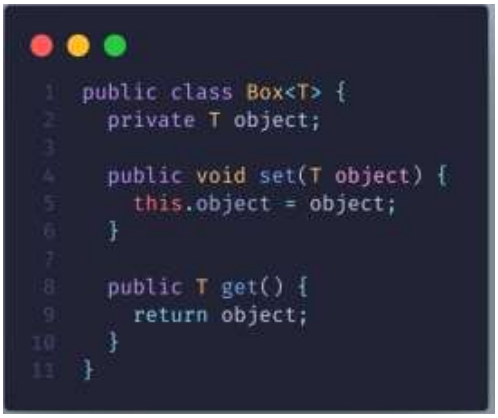
UD 01 - Streams, Ficheros y Expresiones Regulares

Clases genéricas

Se trata de una clase parametrizada sobre uno o más tipos. El tipo se asigna en tiempo de compilación



```
public class Box {
  private Object object;

  public void set(Object object) {
    this.object = object;
  }
  public Object get() {
    return object;
  }
}
```



```
public class Box<T> {
  private T object;

  public void set(T object) {
    this.object = object;
  }

  public T get() {
    return object;
  }
}
```

El código de la izquierda es más propenso a producir errores.

Otro ejemplo:

```
public class Par<T, S> {
  private T obj1;
  private S obj2;

  // Resto de la clase

}
```

Los nombres de tipos de parámetros más usados son:

- E (element, elemento)
- K (key, clave)
- N (number, número)
- T (type, tipo)
- V (value, valor)
- S, U, V, … (2º, 3º, 4º, … tipo)

Para instanciar un objeto genérico, tenemos que indicar los tipos dos veces.

```
Par<String, String> pareja2 = new Par<String, String>("Hola",
"Mundo");
```

Este estilo es muy verboso. Desde Java SE 7 tenemos el operador <>

```
Par<String, String> pareja2 = new Par<>("Hola", "Mundo");
```

Podemos indicar que el tipo parametrizado sea uno en particular (o sus derivados).

```
public class NumericBox<T extends Number>
public class StrangeBox <T extends A>
```

¿Qué es un tipo comodín <?> en Java?

Los tipos comodín (wildcard), aparecen con el concepto de programación genérica. En castellano también los puedes encontrar como tipo salvaje.

Imagina que tienes una caja. Esa caja puede contener diferentes tipos de objetos: manzanas, naranjas, libros, etc. Sin embargo, no sabemos exactamente qué tipo de objeto hay dentro hasta que abrimos la caja.

En Java, los tipos comodín son como esa caja. Representan un tipo desconocido, pero que cumple con ciertas restricciones. Se utilizan cuando no sabemos el tipo exacto de un objeto en tiempo de compilación, pero necesitamos realizar algunas operaciones con él.

¿Para qué sirven los tipos comodín?

Mayor flexibilidad: Permiten escribir código más genérico y reutilizable, ya que no están limitados a un tipo de dato específico.

Mayor seguridad: Ayudan a evitar errores de tiempo de ejecución relacionados con tipos incompatibles.

Colecciones: Son especialmente útiles al trabajar con colecciones como List, Set y Map, donde los elementos pueden ser de diferentes tipos.

```
List<?> myList; // Una lista que puede contener cualquier tipo de
objeto
```

En este ejemplo, myList puede contener una lista de números enteros, una lista de cadenas o incluso una lista de objetos de una clase personalizada. Sin embargo, no podemos agregar elementos directamente a myList porque no sabemos su tipo exacto.

Tipos de tipos comodín:

<?>: Tipo comodín sin restricciones. Puede representar cualquier tipo.

<? extends T>: Tipo comodín con límite superior. Solo puede representar tipos que sean subtipos de T.

<? super T>: Tipo comodín con límite inferior. Solo puede representar tipos que sean supertipos de T.

```
List<? extends Number> numbersList; // Solo puede contener números
List<? super Integer> integerList; // Puede contener Integer y sus
supertipos (como Number o Object)
```

Comparable y Comparator

Comparable y Comparator son dos interfaces en Java utilizadas para comparar objetos. Aquí están las principales diferencias:

Comparable

- La interfaz Comparable se encuentra en el paquete java.lang y tiene un único método llamado compareTo().

- Si quieres que los objetos de una clase sean comparables por defecto, entonces esa clase debe implementar Comparable.

- El método compareTo() compara el objeto actual con el objeto especificado y devuelve un entero negativo, cero o un entero positivo si el objeto actual es menor que, igual a, o mayor que el objeto especificado.

- Comparable es útil cuando tienes una única fuente de ordenamiento.

Comparator

- La interfaz Comparator se encuentra en el paquete java.util y tiene dos métodos: compare() y equals().

- Puedes crear una clase separada que implemente “Comparator”, y luego usar un objeto de esa clase para comparar objetos de otra clase, lo que significa que no necesitas modificar la clase que quieres comparar.

- El método compare() compara sus dos argumentos y devuelve un entero negativo, cero o un entero positivo si el primer argumento es menor que, igual a, o mayor que el segundo.

- Comparator es útil cuando tienes múltiples fuentes de ordenamiento. Por ejemplo, puedes tener un Comparator que ordene por nombre y otro que ordene por edad.

En resumen, Comparable es para una ordenación "natural" y por defecto de los objetos de una clase, mientras que Comparator es para ordenaciones personalizadas.

Otra explicación:

Muchas operaciones entre objetos nos obligan a compararlos: buscar, ordenar, ... Los tipos primitivos y algunas clases ya implementan su orden (natural, lexicográfico). Para nuestras clases (modelo) tenemos que especificar el orden con el que las vamos a tratar

Comparable:

Se trata de un interfaz sencillo. Recibe un objeto del mismo tipo. Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor. Nos sirve para indicar el orden principal de una clase.

```
public interface Compararable<T> {
// 0 si es igual, 1 si es mayor, -1 si es menor.
public int compareTo(T o);
```

```
}
```

Comparator:

Se trata de un interfaz sencillo. Recibe dos argumentos. Devuelve 0 si son iguales, un valor negativo si es menor, y uno positivo si es mayor. Nos sirve para indicar un orden puntual, diferente al orden principal de una clase.

```
public interface Comparator<T> {
public int compare(T o1, T o2);
}
```

Ejemplos:

Diferencia entre Comparable y Comparator en Java.

Comparable

```
import java.util.*;

class Student implements Comparable<Student> {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int compareTo(Student s) {
        return this.id - s.id;
    }

    public String toString() {
        return this.id + " " + this.name;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Alice"));
        list.add(new Student(1, "Bob"));
        list.add(new Student(2, "Charlie"));

        Collections.sort(list);

        System.out.println(list);
    }
}
```

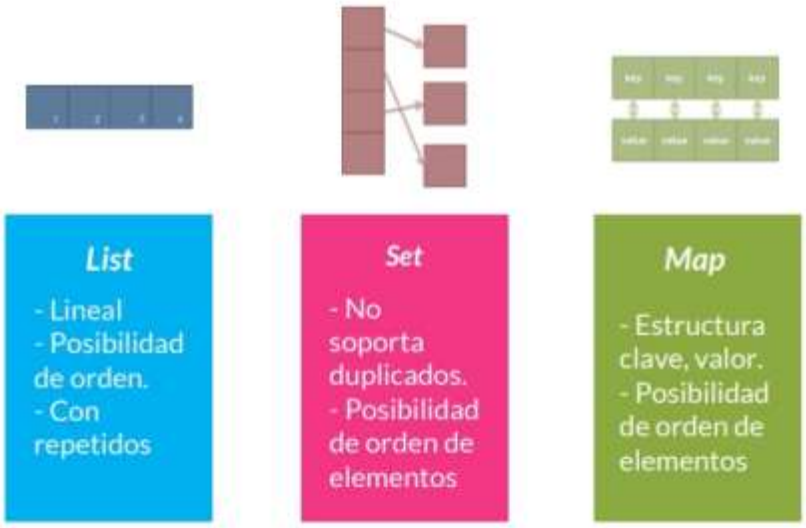
En este ejemplo, la clase Student implementa Comparable<Student>. Cuando llamamos a Collections.sort(list), los objetos Student en la lista se ordenan por su id porque eso es lo que especificamos en el método compareTo().

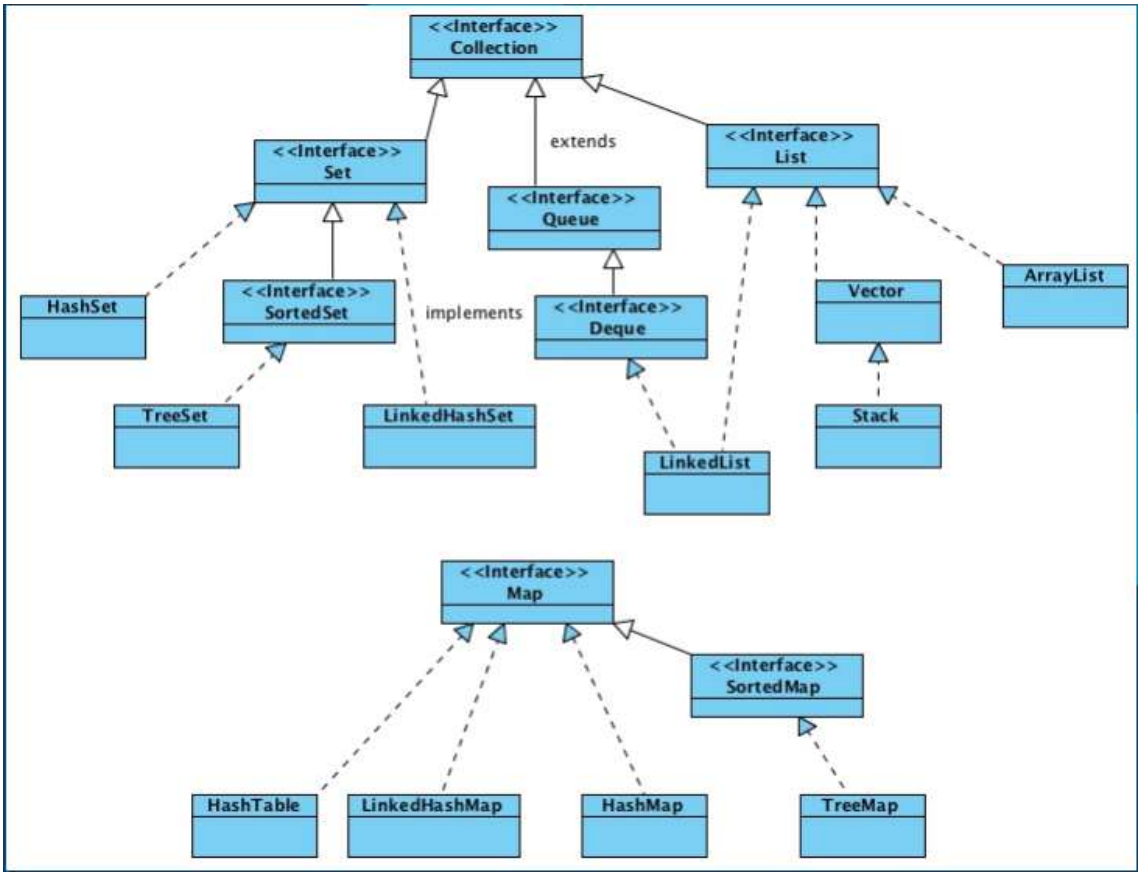
Comparator

```java
import java.util.*;

class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public String toString() {
        return this.id + " " + this.name;
    }
}

class StudentNameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Alice"));
        list.add(new Student(1, "Bob"));
        list.add(new Student(2, "Charlie"));

        Collections.sort(list, new StudentNameComparator());

        System.out.println(list);
    }
}
```

En este ejemplo, la clase Student no implementa Comparable<Student>. En su lugar, creamos una clase separada StudentNameComparator que implementa Comparator<Student>. Cuando llamamos a Collections.sort(list, new StudentNameComparator()), los objetos Student en la lista se ordenan por su name porque eso es lo que especificamos en el método compare().

Colecciones





Ejercicios Básicos:

1 List

Ej1. Crear una lista de Strings y añadir elementos a ella. Luego, imprimir todos los elementos de la lista usando un bucle for-each.

Ej2. Dada una lista de números enteros, escribir una función que devuelva una nueva lista que contenga solo los números pares de la lista original.

Ej3. Dada una lista de Strings, escribir una función que devuelva la longitud del string más largo en la lista.

2 Set

Ej1. Crear un Set de Strings y añadir elementos a él. Luego, imprimir todos los elementos del Set. ¿Qué observas acerca del orden de los elementos?

Ej2. Dada una lista de números enteros, escribir una función que devuelva un Set que contenga solo los números únicos de la lista original.

Ej3. Dada una lista de Strings, escribir una función que devuelva un Set que contenga solo los Strings únicos de la lista original.

3 Map

Ej1. Crear un Map que asocie nombres de países con sus capitales. Luego, imprimir todos los pares de clave-valor del Map.

Ej2. Dada una lista de Strings, escribir una función que devuelva un Map donde las claves son los Strings únicos de la lista y los valores son el número de veces que cada String aparece en la lista.

Ej3. Dada una lista de estudiantes (donde cada estudiante es un objeto con propiedades como nombre, edad, grado, etc.), escribir una función que devuelva un Map donde las claves son los nombres de los estudiantes y los valores son los objetos de los estudiantes.

Funciones Lambda

Trabajando funcionalmente y de forma fluida.

Antes de nada debemos introducir la programación funcional en Java

En la programación Estructurada nos centramos en el QUÉ y el CÓMO, en la funcional nos centramos en el QUÉ

Expresiones Lambda: Las expresiones lambda son funciones anónimas, es decir, funciones que no necesitan una clase. Su sintaxis básica se detalla a continuación:

(parámetros) -> { cuerpo-lambda}

El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

- ✓ Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
- ✓ Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.

Cuerpo de lambda:

- ✓ Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la clausula return en el caso de que deban devolver valores.
- ✓ Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la clausula return en el caso de que la función deba devolver un valor.

Ejemplos:

Ejemplo 1: Ordenar una lista de Strings por longitud

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Dog", "Elephant", "Cat",
"Rabbit", "Butterfly");

        Collections.sort(list, (s1, s2) -> s1.length() - s2.length());

        System.out.println(list);
    }
}
```

En este ejemplo, la función lambda (s1, s2) -> s1.length() - s2.length() se utiliza como argumento para el método Collections.sort(). Esta función lambda compara dos strings por su longitud.

Ejemplo 2: utiliza una función lambda para implementar una interfaz funcional personalizada:

```
public class Main {
    interface Greeting {
        void sayHello(String name);
    }

    public static void main(String[] args) {
        Greeting greeting = (name) -> System.out.println("Hello, " +
name);
        greeting.sayHello("Alice");
    }
}
```

En este ejemplo, definimos una interfaz funcional “Greeting” con un método “sayHello()”. Luego, en el método “main()”, creamos una instancia de “Greeting” utilizando una función lambda “(name) -> System.out.println("Hello, " + name)”. Esta función lambda toma un nombre e imprime un saludo personalizado.

Ejercicios:

1. Calculadora con funciones lambda: Crea una interfaz funcional “Calculator” con un método “calculate()”. Este método debe tomar dos números enteros y devolver un número entero. Luego, en tu método “main()”, crea varias instancias de “Calculator” utilizando funciones lambda para implementar operaciones como suma, resta, multiplicación y división. Finalmente, prueba tus calculadoras con algunos números.

2. Filtrado de lista con funciones lambda: Crea una lista de Strings que contenga varios nombres. Luego, utiliza una función lambda para filtrar esta lista y crear una nueva lista que solo contenga los nombres que comienzan con una letra específica (por ejemplo, "A"). Para hacer esto, puedes utilizar el método “removeIf()” de “ArrayList”, que toma un “Predicate” (que es una interfaz funcional que puedes implementar con una función lambda).

Referencia a métodos

En Java, una referencia a método es una forma concisa de representar un método sin invocarlo inmediatamente. Es como una "etiqueta" que apunta a un método específico. Esta característica, introducida en Java 8, facilita la escritura de código más funcional y expresivo, especialmente cuando se trabaja con lambdas y streams.

Las referencia a métodos se representan con ::

System.out.println(“Hola”); → System.out::println

```
import java.util.Arrays;
import java.util.List;
```

```java
public class EjemploReferenciaAmetodo {
    public static void main(String[] args) {
        List<String> mensajes = Arrays.asList("Hola","Adios", "Buenos
días", "Buenas", "Chao");

        mensajes.forEach(System.out::println);
    }
}
```

API Stream

La API Stream en Java es una característica introducida en Java 8 que permite realizar operaciones de procesamiento de datos en secuencias de elementos, como listas y arrays, de una manera declarativa.

Aquí están algunos puntos clave sobre la API Stream:

1.- **Flujo de datos**: Un Stream proporciona un conjunto de elementos de un tipo específico en una secuencia. Puedes obtener un Stream de colecciones, listas, conjuntos, enteros, caracteres de una cadena, etc.
2.- **Operaciones en cadena**: Los Streams están diseñados para trabajar con funciones lambda y permiten operaciones en cadena. Puedes combinar varias operaciones en una sola línea de código.
3.- **Operaciones de agregación**: Los Streams soportan métodos como "filter", "map", "limit", "reduce", "find", "match", y así sucesivamente. Estos métodos se pueden dividir en dos categorías: operaciones intermedias (que pueden ser encadenadas) y operaciones terminales (que producen un resultado o un efecto secundario).
4.- **Paralelismo**: Una de las características más importantes de la API Stream es que permite procesamiento paralelo. Esto significa que puedes dividir tus datos en múltiples chunks y procesarlos en paralelo, lo cual puede ser muy útil cuando trabajas con grandes cantidades de datos.

Veamos un ejemplo de cómo se puede usar la API Stream para filtrar y transformar datos en una lista:

```
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9, 10);

        List<Integer> evenSquares = numbers.stream()
                                           .filter(n -> n % 2 == 0)
                                           .map(n -> n * n)

.collect(Collectors.toList());

        System.out.println(evenSquares);
    }
}
```

En este ejemplo, creamos un Stream de la lista numbers, luego usamos el método filter para filtrar los números pares y el método map para transformar cada número en su cuadrado. Finalmente, recogemos los resultados en una nueva lista.

Interfaces Java para trabajar con Streams:

1. Predicate: Esta es una interfaz funcional que toma un argumento y devuelve un valor booleano. Se utiliza comúnmente en métodos de filtrado. Por ejemplo, puedes usar un "Predicate" para filtrar todos los números pares en una lista.

```
Predicate<Integer> isEven = n -> n % 2 == 0;
```

2. Consumer: Esta es una interfaz funcional que toma un argumento y no devuelve nada. Se utiliza comúnmente en métodos que realizan alguna acción en cada elemento de una colección, como imprimir cada elemento.

```
Consumer<Integer> print = n -> System.out.println(n);
```

3. Function: Esta es una interfaz funcional que toma un argumento y devuelve un resultado. Se utiliza comúnmente en métodos que transforman elementos de una colección, como convertir todos los números en una lista a sus cuadrados.

```
Function<Integer, Integer> square = n -> n * n;
```

4. Supplier: Esta es una interfaz funcional que no toma ningún argumento y devuelve un resultado. Se utiliza comúnmente en métodos que generan valores, como generar una secuencia de números aleatorios.

```
Supplier<Integer> random = () -> new Random().nextInt();
```

Estas interfaces funcionales son muy útiles cuando trabajas con la API Stream y las expresiones lambda en Java, ya que te permiten pasar comportamientos como argumentos a los métodos.

Otra explicación:

Predicate<T>: Comprueba si se cumple o no una condición. Se utiliza mucho junto a expresiones lambda a la hora de filtrar:

```
.filter((p) -> p.getEdad() >= 35
```

Consumer<T>: Sirve para consumir objetos. Uno de los ejemplos más claros es imprimir.

```
.forEach(System.out::println)
```

Adicionalmente, tiene el método andThen, que permite componer consumidores, para encadenar una secuencia de operacion

Function<T, R>: Sirve para aplicar una transformación a un objeto. El ejemplo más claro es el mapeo de objetos en otros.

```
.map((p) -> p.getNombre())
```

Adicionalmente, tiene otros métodos:

- andThen, que permite componer funciones.
- compose, que compone dos funciones, a la inversa de la anterior.
- identity, una función que siempre devuelve

Supplier<T>: Sirve para devolver un valor.

Tiene algunos interfaces especializados para tipos básicos:

- IntSupplier
- LongSupplier
- DoubleSupplier
- BooleanSupplier

Ejemplos:

Veamos unos ejemplos de cómo se pueden utilizar estas interfaces funcionales en Java:

Ejemplo de Predicate

```
import java.util.*;
import java.util.function.*;

public class Main {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;

        List<Integer> numbers = = new ArrayList<>(

Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
        numbers.removeIf(isEven);

        System.out.println(numbers);  // Imprime: [1, 3, 5, 7, 9]
    }
}
```

En este ejemplo, creamos un “Predicate” que verifica si un número es par. Luego, utilizamos este “Predicate” con el método “removeIf()” para eliminar todos los números pares de la lista.

Ejemplo de Consumer

```
import java.util.*;
import java.util.function.*;

public class Main {
    public static void main(String[] args) {
        Consumer<String> print = s -> System.out.println(s);

        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
        names.forEach(print);  // Imprime: Alice, Bob, Charlie
    }
}
```

En este ejemplo, creamos un “Consumer” que imprime una cadena. Luego, utilizamos este “Consumer” con el método “forEach()” para imprimir todos los nombres en la lista.

Ejemplo de Function

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Function<Integer, Integer> square = n -> n * n;

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        List<Integer> squares = numbers.stream()
                                       .map(square)
                                       .collect(Collectors.toList());

        System.out.println(squares);  // Imprime: [1, 4, 9, 16, 25]
    }
}
```

En este ejemplo, creamos una “Function” que calcula el cuadrado de un número. Luego, utilizamos esta “Function” con el método “map()” para transformar todos los números en la lista a sus cuadrados.

Ejemplo de Supplier

```java
import java.util.function.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Supplier<Integer> random = () -> new Random().nextInt(10);

        for (int i = 0; i < 5; i++) {
            System.out.println(random.get());
        }
    }
}
```

En este ejemplo, creamos un “Supplier” que genera un número aleatorio entre 0 y 9. Luego, utilizamos este “Supplier” para imprimir cinco números aleatorios.

Ejercicios:

1) Filtrar números impares: Crea una lista de números enteros. Utiliza un “Predicate” para filtrar la lista y eliminar todos los números impares.
2) Imprimir todos los elementos de una lista: Crea una lista de cadenas. Utiliza un “Consumer” para imprimir cada cadena en la lista.
3) Transformar una lista de números: Crea una lista de números enteros. Utiliza una “Function” para transformar cada número en la lista a su cubo.
4) Generar una secuencia de números aleatorios: Utiliza un “Supplier” para generar e imprimir una secuencia de diez números aleatorios.
5) Filtrar nombres que comienzan con una letra específica: Crea una lista de nombres. Utiliza un “Predicate” para filtrar la lista y mantener solo los nombres que comienzan con una letra específica (por ejemplo, "A").
6) Aplicar una operación a cada elemento de una lista: Crea una lista de números enteros. Utiliza un “Consumer” para aplicar una operación a cada número en la lista (por ejemplo, multiplicar cada número por 2).

Api Stream: Métodos de búsqueda:

Los métodos de búsqueda son operaciones terminales en la API Stream de Java que nos permiten identificar elementos que cumplen con ciertas condiciones y, si existen, obtener estos elementos específicos. Aquí te presento algunos de los métodos de búsqueda más utilizados:

- “allMatch(Predicate<T>)”: Este método verifica si todos los elementos en el stream satisfacen una condición especificada por el predicado. Si todos los elementos cumplen con la condición, el método devuelve “true”; de lo contrario, devuelve “false”.
- “anyMatch(Predicate<T>)”: Este método verifica si al menos un elemento en el stream satisface una condición especificada por el predicado. Si al menos un elemento cumple con la condición, el método devuelve “true”; de lo contrario, devuelve “false”.

- “noneMatch(Predicate<T>)”: Este método es el opuesto de “allMatch()”. Verifica si ninguno de los elementos en el stream satisface una condición especificada por el predicado. Si ningún elemento cumple con la condición, el método devuelve “true”; de lo contrario, devuelve “false”.
- “findAny()”: Este método devuelve un “Optional<T>“ que contiene un elemento arbitrario del stream si el stream tiene elementos; de lo contrario, devuelve un “Optional” vacío. Este método es útil cuando trabajas con streams paralelos, ya que puede devolver cualquier elemento que cumpla con la condición, no necesariamente el primero.
- “findFirst()”: Este método devuelve un “Optional<T>“ que contiene el primer elemento del stream si el stream tiene elementos; de lo contrario, devuelve un “Optional” vacío. Este método es útil cuando necesitas obtener el primer elemento que cumple con una condición específica.

Estos métodos de búsqueda son herramientas poderosas que puedes utilizar para trabajar con streams en Java. Te permiten realizar operaciones complejas de búsqueda y filtrado de manera eficiente y concisa.

API Stream - Métodos de datos, cálculo y ordenación

- Métodos de Datos y Cálculo: Los streams proporcionan varios métodos terminales para realizar operaciones y cálculos sobre los datos. Durante el curso, exploraremos tres tipos principales: Reducción y Resumen, Agrupamiento y Particionamiento.
- Métodos de Reducción: Estos métodos reducen el stream a un solo valor.
  - ✓ “reduce(BinaryOperator<T>):Optional<T>“: Realiza la reducción del Stream utilizando una función asociativa y devuelve un “Optional”.
  - ✓ “reduce(T, BinaryOperator<T>):T”: Realiza la reducción utilizando un valor inicial y una función asociativa. Devuelve un valor como resultado.
- Métodos de Resumen: Estos métodos resumen todos los elementos de un stream en un solo valor.
  - ✓ “count()”: Devuelve el número de elementos en el stream.
  - ✓ “min(...)”, “max(...)”: Devuelven el valor mínimo o máximo respectivamente. Se puede utilizar un “Comparator” para modificar el orden natural.
- Métodos de Ordenación: Estas son operaciones intermedias que devuelven un stream con sus elementos ordenados.
  - ✓ “sorted()”: Ordena el stream según el orden natural de sus elementos.
  - ✓ “sorted(Comparator<T>)”: Ordena el stream según el orden especificado por el “Comparator” proporcionado.

Estos métodos de la API Stream de Java permiten realizar operaciones complejas de datos y cálculos de manera eficiente y concisa. Te permiten manipular y transformar los datos de formas que serían difíciles o tediosas con las estructuras de datos y los bucles tradicionales.

API Stream - Map y FlapMap

- **Uso de "map"**: "map" es una de las operaciones intermedias más utilizadas en la API Stream. Permite transformar un objeto en otro mediante una "Function<T, R>". Se invoca sobre un "Stream<T>" y retorna un "Stream<R>". Es común realizar transformaciones sucesivas con "map". Por ejemplo:

```
lista.stream().map(Persona::getNombre).map(String::toUpperCase).forEac
h(System.out::println);
```

- **Uso de "flatMap"**: Los streams sobre colecciones de un solo nivel permiten transformaciones a través de "map". Pero, ¿qué sucede si tenemos una colección de dos niveles (o una colección dentro de objetos de otro tipo)? Por ejemplo, considera la siguiente clase "Persona" que tiene una lista de "Viaje":

Para trabajar con la colección interna, necesitamos un método que nos permita "aplanar" un "Stream<Stream<T>>" en un solo "Stream<T>". Ese es el propósito de "flatMap". Por ejemplo:

```
public class Persona {
    private String nombre;
    private List<Viaje> viajes = new ArrayList<>();
}
```

Para trabajar con la colección interna, necesitamos un método que nos permita "aplanar" un Stream<Stream<T>> en un solo Stream<T>. Ese es el propósito de flatMap. Por ejemplo:

```
lista.stream()
     .map(Persona::getViajes)
     .flatMap(viajes -> viajes.stream())
     .map(Viaje::getPais)
     .forEach(System.out::println);
```

En este ejemplo, "flatMap" toma cada lista de viajes (un "Stream<Viaje>") y los combina en un solo "Stream<Viaje>". Luego, "map" se utiliza para transformar cada "Viaje" en su país correspondiente.

API Stream – Collectors

- **Collectors**: Los "Collectors" permiten construir una colección mutable como resultado de las operaciones sobre un stream en una operación terminal.
- **Colectores Básicos**: Permiten operaciones que recolectan todos los valores en uno solo. Algunas de estas operaciones se solapan con operaciones terminales ya estudiadas, pero están presentes porque se pueden combinar con otros colectores más potentes.
    - "counting()": Cuenta el número de elementos.

- “minBy(...)”, “maxBy(...)”: Obtiene el valor mínimo o máximo según un comparador.
- “summingInt”, “summingLong”, “summingDouble”: Calcula la suma de los elementos (según el tipo).
- “averagingInt”, “averagingLong”, “averagingDouble”: Calcula la media (según el tipo).
- “summarizingInt”, “summarizingLong”, “summarizingDouble”: Agrupa los valores anteriores en un objeto (según el tipo).
- “joining()”: Une los elementos en una cadena.

- **Colectores "groupingBy"**: Realizan una función similar a la cláusula “GROUP BY” de SQL, permitiendo agrupar los elementos de un stream por uno o varios valores. Retornan un “Map”.

Los “Collectors” son una herramienta poderosa en la API Stream de Java, que permiten realizar operaciones complejas de recolección y agrupación de datos de manera eficiente y concisa.

API Stream - “filter”

• **“filter”**: Es una operación intermedia que permite eliminar del stream aquellos elementos que no cumplen con una determinada condición, especificada por un “Predicate<T>“. Por ejemplo, el siguiente código filtra una lista de personas para incluir solo aquellas cuya edad está entre 18 y 65 años:

```
personas.stream()
        .filter(p -> p.getEdad() >= 18 && p.getEdad() <= 65)
        .forEach(persona -> System.out.printf("%s (%d años)%n",
persona.getNombre(), persona.getEdad()));
```

• “**filter**” es muy combinable con algunos métodos como “findAny” o “findFirst”. Por ejemplo, el siguiente código busca a una persona llamada "Pepe" en la lista de personas. Si no se encuentra a "Pepe", se devuelve una nueva “Persona”:

```
Persona p1 = personas.stream()
                     .filter(p ->
p.getNombre().equalsIgnoreCase("Pepe"))
                     .findAny()
                     .orElse(new Persona());
```

El método “filter” es una herramienta poderosa en la API Stream de Java, que permite realizar operaciones de filtrado de datos de manera eficiente y concisa.

API Stream - Referencias a Métodos

• **Referencias a Métodos**: Las referencias a métodos son una característica de Java que permite hacer el código aún más conciso. Existen cuatro tipos principales de referencias a métodos:

→ "Clase::metodoEstatico": Referencia a un método estático de una clase. Por ejemplo, "Math::sqrt" es una referencia al método estático "sqrt" de la clase "Math".

→ "objeto::metodoInstancia": Referencia a un método de instancia de un objeto concreto. Por ejemplo, si tienes un objeto "List<String> lista", entonces "lista::size" es una referencia al método "size" de esa lista específica.

→ "Tipo::nombreMetodo": Referencia a un método de instancia de un objeto arbitrario de un tipo en particular. Por ejemplo, "String::length" es una referencia al método "length" que puede ser llamado en cualquier objeto de tipo "String".

→ "Clase::new": Referencia a un constructor de una clase. Por ejemplo, "ArrayList::new" es una referencia al constructor de la clase "ArrayList".

Las referencias a métodos son una forma poderosa y concisa de referirse a métodos existentes en Java, y son especialmente útiles cuando se trabaja con la API Stream y otras APIs funcionales en Java.

Api Stream: Similitudes a SQL:

SQL	API Stream
from	stream()
select	map()
where	filter() (antes de un collecting)
order by	sorted()
distinct	distinct()
having	filter() (después de un collecting)
join	flatMap()
union	concat().distinct()
offset	skip()
limit	limit()
group by	collect(groupingBy())
count	count()

Stream Resumen:

Stream en Java representa una secuencia de elementos sobre la cual se pueden realizar diversas operaciones:

- “filter”: Permite filtrar los elementos de un Stream según un predicado proporcionado.
- “map”: Transforma los elementos de un Stream, similar a la cláusula SELECT en SQL.
- “sorted”: Ordena los elementos de un Stream según un Comparator proporcionado.
- “min”, “max”, “count”: Permiten obtener el valor mínimo, máximo y el conteo de elementos en un Stream, respectivamente.
- “collect”: Define funciones de agregación, como agrupaciones, particiones, etc.
- “Comparator”: Proporciona métodos para realizar ordenamientos, se utiliza en conjunto con el método “sorted”.
- “Collectors”: Ofrece métodos para realizar operaciones como agrupar, sumar, promediar, obtener estadísticas, etc. Se utiliza en conjunto con el método “collect”.

En resumen, la API Stream en Java proporciona una forma eficiente y concisa de manipular y procesar secuencias de elementos, ofreciendo una amplia gama de operaciones para realizar tareas comunes de programación.

Ejercicios Streams (búsquedas):

Ejercicio 1:

Crea una clase Fruta con al menos dos atributos, nombre y color.

Crea una lista con diferentes frutas. Llámala frutería

Obtén una lista de Strings, que contenga el nombre de las frutas que contenía la frutería.

Ejercicio 2:

Utiliza la misma estructura del ejercicio 1, y ahora imprime por pantalla los colores de las diferentes frutas. (No pueden aparecer colores repetidos.)

Ejercicio 3:

Crea una lista de números.

Usando stream, calcula la suma de los cuadrados de todos los números de la lista.

Ejercicio 4:

Crea la clase persona con al menos los atributos nombre y edad.

- Ordenar una lista de personas por edad de forma descendente

- Imprimimos la lista de personas ordenada por nombre de forma ascendente

Ejercicio 5:

Crea la clase empleado con al menos los atributos: nombre, departamento.

- Crea varios empleados y agrupar la lista de empleados por departamento.

- Agrupamos los empleados por departamento y contamos cuantos empleados hay en cada departamento.

- Dado un departamento mostraremos sus empleados. Por ejemplo, muestra los empleados de ventas.

- Dado un nombre de empleado, mostraremos su departamento.