Clase 5

martes, 16 de septiembre de 2025 15:20

Interfaqces funcionales, lambdas y referencias a métodos se usan para facilitar crear funciones.
Son una forma de crear objetos función más simple y menos larga.

```
class Addition implements MathOperation {

    public double operation(double a, double b) {
        return a + b;
    }
}
```

Es muy largo!
Creamos una clase para algo muy simple

```
MathOperation addition = new MathOperation() {
    public double operation(double a, double b) {
        return a + b;
    }
};
```

Clases anónimas, también demasiado detalle!

```
MathOperation addition = (int a, int b) -> a + b;
```

Código conciso: expresión lambda

Te guardas un lambda que es la implementación de una función en una variable y podes usarla después
Son como una clase anónima pero más corta
Es una función que tima parámetros de entrada y devuelve un valor
No tiene nombre ni pertenece a una clase, se puede pasar como parámetro y ejecutarse bajo demanda

```
Collections.sort(words,
        (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Parámetros (podría ser vacío)

Operador Flecha

Cuerpo del Lambda (implementación de la interface)

```
public interface MyPrinter {
    public void print(String s);
}
```

Expresión Lambda que solo invoca a un método con parámetros:

```
MyPrinter myPrinter = (s) -> { System.out.println(s); };
```

Referencias a métodos:

```
MyPrinter myPrinter = System.out::println;

myPrinter.print("Hola");
```

Los dos puntos dobles le indican al compilador que es **una referencia a un método** y el método al que se hace referencia es el que viene después de : :

Los dos puntitos se usan para avisarle que lo que viene es un método.

comparingInt es un método de clase de la **interface Comparator** que se usa para construir **comparadores personalizados** para **enteros primitivos**.

```
Collections.sort(words, Comparator.comparingInt(String::length));
words.sort(comparingInt(String::length));
```

Referencias a métodos

```
class Persona {
public int getEdad() {
    return edad;
}
}
```

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona("Lucía", 30));
personas.add(new Persona("Diego", 25));
personas.add(new Persona("Juana", 35));
personas.sort(Comparator.comparingInt(Persona::getEdad));
```

Referencias a métodos

Crea un comparador que compara objetos de tipo **Persona** en función de su edad.
La lista de personas se ordena utilizando dicho comparador resultando en una lista ordenada por edad.

Lo de arriba crea un comparador que invoca a la longitu de las strings y compara en base a eso
Lo de abajo le dice: creame un comparador de personas en base a la edad y después el comparator usa eso para ordenar. Crea ordenadores de entero sy lo que le paso es la función.

```
package enumerativos;
import java.util.function.DoubleBinaryOperator;
public enum Operation {
    PLUS("+", (x, y) -> x + y),
    MINUS("-", (x, y) -> x - y),
    TIMES("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);
    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    public String toString() {
        return symbol;
    }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}
```

```
@FunctionalInterface
public interface DoubleBinaryOperator {
   double applyAsDouble(double left, double right);
}
```

Es una interface funcional del paquete **java.util.function** que representa a una función que toma 2 valores double como argumento y retorna un valor double como resultado.

Tipo enumerativo basado en lambdas

Es posible implementar el **comportamiento específico de cada constante enum** pasándole al constructor una expresión Lambda con dicho comportamiento.

El constructor guarda la expresión Lambda en la variable de instancia **op.**

El método **apply()** de **Operation** se usa para invocar al Lambda. No están más las sobreescrituras del método **apply()**.

Despues se usa como operation.plus.apply(3,4), igual que antes pero la implementación ta wena, es más simple.

Colecciones--> esctructura de datos.
Agrupa elementos u objetos.

Un **framework de colecciones** es una arquitectura que permite representar y manipular colecciones de datos de manera estándar. Todos los *frameworks de colecciones* están compuestos por:

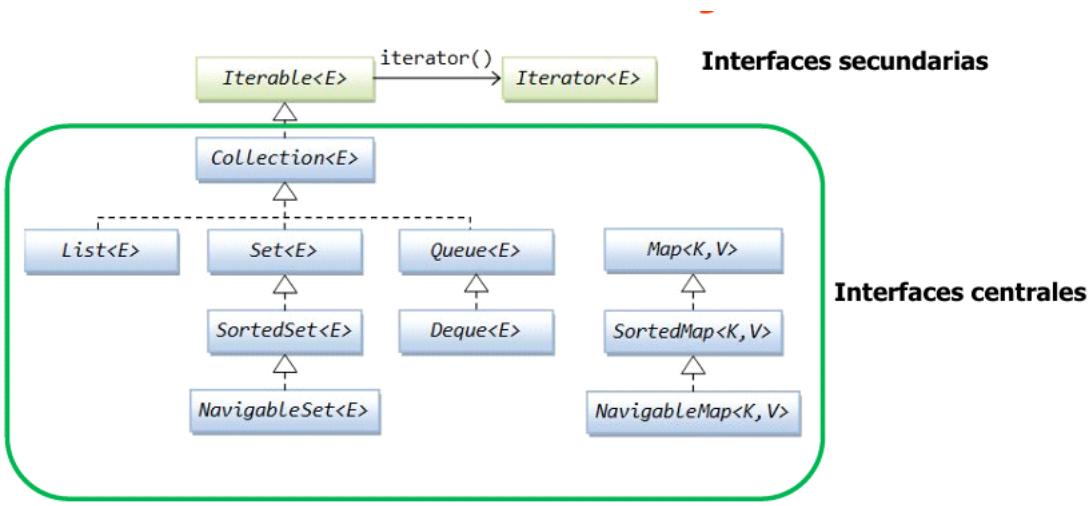
Interfaces: son **tipos de datos abstractos** que representan colecciones. Las interfaces permiten que las colecciones sean manipuladas independientemente de los detalles de implementación. Forman una jerarquía.

Implementaciones: son las implementaciones concretas de las interfaces. Son estructuras de datos reusables.

Algoritmos: son **métodos** que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos que implementan alguna de las interfaces de colecciones. Son métodos **polimórficos** es decir el mismo método se usa sobre diferentes implementaciones de las interfaces de colecciones. Son unidades funcionales reusables.

El **framework de colecciones** forma parte de JAVA a partir de la versión 1.2

Ej interfaz:list, impelmentación: arraylist



Las interfaces centrales especifican los múltiples contenedores de elementos y las interfaces secundarias especifican las formas de recorrido de las colecciones. Las interfaces centrales permiten a las colecciones ser manipuladas independientemente de los detalles de implementación

```
import java.util.*;
public class ColeccionSimple {
  public static void main(String [] args){
    Collection<Integer> c=new ArrayList<>();
     for (int i=0; i < 10; i++)
        c.add(i);
     for(int i: c)
        System.out.println(i);
  }
}
```

Auto-Boxing

Las collections usab objetos, pero yo puedo pasarle una int y sabe que tiene que ponerlo en un objeto de tipo integer.