Programación Imperativa

La programación imperativa (del latín imperare = ordenar) es el paradigma de programación más antiguo. De acuerdo con este paradigma, un programa consiste en una secuencia claramente definida de instrucciones para un ordenador.

El código fuente de los lenguajes imperativos encadena instrucciones una detrás de otra que determinan lo que debe hacer el ordenador en cada momento para alcanzar un resultado deseado. Los valores utilizados en las variables se modifican durante la ejecución del programa. Para gestionar las instrucciones, se integran estructuras de control como bucles o estructuras anidadas en el código.

Ventajas	Desventajas
Fácilmente legible.	El código se convierte rápidamente en demasiado amplio y difícil de abarcar.
Fácil de aprender en lo relativo a comportamientos.	Mayor riesgo durante la edición.
Un modelo fácilmente comprensible para los principiantes (vía de solución).	El mantenimiento bloquea el desarrollo de la aplicación, ya que la programación funciona estrechamente con el sistema.
Se pueden tener en cuenta características de casos especiales de la aplicación.	La optimización y la ampliación son más difíciles.

Programación Declarativa

La programación imperativa se centra en el "cómo", y la declarativa, en el "qué".

En la programación declarativa se describe directamente el resultado final deseado (el qué). Pongamos un ejemplo culinario para entenderlo mejor: los lenguajes imperativos proporcionan la receta, mientras que los declarativos, fotos del plato preparado.

En los lenguajes declarativos, el código fuente permanece muy abstracto en relación al procedimiento concreto. Para llegar a la solución, se utiliza un algoritmo que encuentra y utiliza automáticamente los métodos adecuados. Este procedimiento tiene numerosas ventajas: de esta forma, los programas no solo se pueden escribir considerablemente más rápido, sino que las aplicaciones se pueden optimizar también de forma muy sencilla, ya que, si en el futuro se desarrolla un nuevo método, el algoritmo puede acceder fácilmente al método nuevo gracias a la aplicación abstracta del código fuente.

Funciones como ciudadanos de primera clase

- Esto quiere decir que las funciones van a poder ser guardadas en una variable, pasadas como un parámetro o retornadas como un resultado.

Cuando decimos que algo es *ciudadano de primera clase* en un lenguaje, quiere decir que el lenguaje reconoce que se pueden usar estos elementos como un componente mas del lenguaje. Sabe que tienen su propia sintaxis o que cumple al menos con estas caracteristicas:
Un ciudadano de primera clase puede:

- Ser pasado como argumento
- Ser el retorno de una funcion
- Ser definido y modificado
- Ser asignado a variables

Por ejemplo, otros ciudadanos de primera clase en java son:

- Clases
- Interfaces
- Datos primitivos
- Anotaciones
- Generics

Funciones puras

- Son aquellas que producen el **mismo resultado para el mismo parámetro**, por ejemplo, una función que suma que recibe como parámetro 5 y 3, siempre va a retornar 8.
- Son funciones que no tienen ningún tipo de dependencias(BD, archivos, etc).
- No dependen ni las afecta el contexto (Funcionan en aislamiento).
- No genera valores aleatorios.
- Son predecibles lo que ayuda a la hora de hacer pruebas.
- No tienen efectos laterales, es decir, cambios en la base de datos, creación de archivos, cambios en el sistema, etc.
- Una función pura puede invocar a otra pura ya que su resultado es **predecible**, de lo contrario se considera **impura**.

Efectos secundarios

- Todo cambio observable desde fuera del sistema se considera un efecto secundario. por ejemplo si tenemos una función que al ejecutarla cambia el color del banner en nuestra app, a este cambio de color se le considera un efecto secundario el cual es observable desde fuera del sistema. otras acciones que podemos considerar efectos secundarios son:
    - Leer, crear y modificar archivos.
    - Leer y escribir en una base de datos.
    - Enviar y recibir una llamada a una red.
    - Alterar un objeto o variable utilizada por otras funciones.
- Reducir el impacto de los cambios secundarios nos ayuda a tener una mejor estructura, a tener más funciones puras y a tener separadas las responsabilidades.
- Debemos buscar que las funciones impuras sólo sean puntos de entrada de información, una vez la información está adentro debemos manejarla con funciones puras.
- Mayores funciones puras, mayor es la capacidad de testear nuestro sistema.
- En comparación sería como la Clean Architecture.

Funciones de orden mayor

- Toma otra función como parámetro o retorna una función como resultado.
- Las principales ventajas de estas funciones son:

    - Se pueden pasar comportamientos
    - pueden compartir un medio de comunicación (Callbacks).
    - Pueden compartir lógica y reglas del negocio.

Funciones lambda

- Son funciones anónimas de un solo uso, no tienen nombre, se usan en un solo lugar y son demasiado simples.

Inmutabilidad

- Inmutabilidad es algo que nunca cambia.
  Ventaja:
    - Una vez creado el dato (objeto, variable) no se puede modificar.
    - Esto facilita la creación de funciones puras.
    - Facilita la concurrencia cuando se quiera usar en distintos hilos de proceso.

  Desventajas:
    - Cada modificación crea una nueva instancia del objeto o variable..
    - Habrá objetos mutables fuera de nuestro alcance, pero debemos de generar alguna manera de evitar que estos muten.
    - Requiere especial atención al diseño.

  Para tener en cuenta a nivel de código (JAVA):
    - Usar la palabra reservada **final** para convertir una clase inmutable, esto quiere decir que la clase no se va a poder extender.
    - Usar la palabra reservada **final** para convertir una variable final, esto evitará que muten en algún momento de la ejecución, también hace que sea requerida en el constructor del objeto y no permitirá modificaciones mediante setters.
    - Como precaución extra para asegurar que una variable sea inmutable en los métodos getter debemos devolver una copia de la variable.

Repositorio de los ejercicios

- Link de los ejercicios
- Para ejecutar cada ejemplo de cada clase se ubica en la rama de **job-search** y en la carpeta **modules** se crea el método main a la clase que se quiera ejecutar.

java.util.function: Function

- Interfaz funcional que recibe un valor y regresa un resultado a través del método **apply( )**.
- Al ser un tipo permite que esta sea almacenada en variables, pasada como parámetros o retornada como resultado.
- se puede aplicar mediante una lambda ya que Function en una Functional interfaz,
- **Ejemplo:**

```
Function<Integer, String> squareFunction = number → String.valueOf(number*number);
squareFunction.apply( t: 5); //25
```

Dónde **Integer** es el tipo de dato de entrada y **String** el tipo de retorno.

java.util.function: Predicate

- Interfaz funcional que recibe un valor y devuelve un **boolean**.
- Se puede evaluar su lógica con una lambda (Ya que también es una interfaz funcional) y su forma de ejecutarlo es a través del método **test( )**.
  Ejemplo;

```
Predicate<Integer> isOdd = number → number % 2 == 1;
isOdd.test( t: 5); //true
```

Donde **Integer** es el tipo de dato que le vamos a pasar como parámetro (x).

java.util.function: Consumer y Supplier

- La interfaz funcional **Consumer** permite realizar operaciones con una lambda sobre un **tipo** de dato en una lista, por cada dato en la lista vamos consumiendo y operando sobre ese dato.
  - Su ejecución se hace a través del método **accept( )**.

Consumer: Es una expresion lambda que acepta un solo valor y no devuelven valor alguno.
Ejemplo: Una funcion que reciba una lista de archivos y borre cada uno de ellos, sin devolver nada.
Supplier: Es una expresion que no tienen parámetros pero devuelven un resultado.
Ejemplo: Se crea un supplier de tipo CLIArguments llamado generator que no recibe ni un parametro pero que crea un nuevo objeto CLIArguments y retorna generator, Se pueden crear archivos bajo demanda.

- Si, puedes "*traducir*" los nombres y entender un poco mas que hacen:
  - Consumer -> Consumidor
  - Supplier -> Proveedor

  Entonces, puedes ver que un `Consumer` se encarga de "consumir" los datos que le pases.

```
Consumer<Student> saveProgressInDataBase = student -> db.updateStudent(student);

Supplier<String> randomPasswordGenerator = () -> complexAlgorithm.generate() ;
```

- Puedes tambien tener en cuenta que el `Consumer` de alguien que **RECIBE** datos y el `Supplier` es alguien que **SUMINISTRA** datos

Ejemplo:

```
static void showHelp(CLIArguments cliArguments) {  Complexity is 3 Everything is cool!
    Consumer<CLIArguments> consumerHelper = cliArguments1 → {
        if (cliArguments1.isHelp()) {
            System.out.println("Manual solicitado");
        }
    };

    consumerHelper.accept(cliArguments);
}
```

Donde **<CLIArguments>** es el tipo de dato de entrada y su ejecución es así:

```
consumerHelper.accept(cliArguments);
```

- La interfaz funcional **Supplier** se encarga de proveer datos. Una de sus utilidades es generar configuraciones o archivos bajo demanda.
    - Su ejecución se hace a través del método **get( )**.

  Ejemplo:

```
Supplier<CLIArguments> generator = () → new CLIArguments();
```

Donde **<CLIArguments>** es el tipo de dato de entrada y su ejecución es así:

```
CLIArguments newCLIArguments = generator.get();
```

java.util.function: Operators y BiFunction

- **Unary Operator<T>:** Es una interfaz funcional que recibe un parámetro de un tipo y devuelve un resultado de ese mismo tipo.
    - Este método se usa más que todo para hacer una única operación a un dato, se opera a través de una lambda.
    - Su ejecución es a través del método **apply( )**.

  Ejemplo:.

```
UnaryOperator<String> bold = text → "**" + text + "**";
bold.apply( t: "Hola");//**Hola**
```

Donde **<String>** es el tipo de dato de entrada y de salida

- **Bifunction<T, U, R>:** Es una interfaz funcional que recibe 2 argumentos y devuelve un resultado no necesariamente del mismo tipo de dato:
    - T: Es el tipo de dato del primer argumento.
    - U: Es el tipo de dato del segundo argumento.
    - R: Tipo de dato del resultado de la función.
- Se ejecuta mediante el método **apply()**

  Ejemplo:

```
BiFunction<String, Integer, String> leftPad =
        (text, numberOfSpaces) → String.format("%" + numberOfSpaces + "s", text);

leftPad.apply( t: "Java",  u: 5);//     Java
```

Dónde recibe como parámetros de entrada un **String** y un **Integer**, y devuelve un **String**.

- **BinaryOperator<T>:** Es una interfaz funcional que
    - Recibe 2 parámetros del mismo tipo y devuelve un resultado de ese mismo tipo.
    - Se ejecuta con el método **apply().**

Ejemplo:

```
BinaryOperator<Integer> multiply = (integer1, integer2) → integer1 * integer2;
multiply.apply( t: 5,  u: 6);//30
```

Donde **<Integer>** es el tipo de dato de los parámetros de entrada **(integer1, integer2)** y también el tipo de dato del parámetro de salida (Resultado).

En resumen

- ***T tipo genérico**
  - `Consumer<T>`: recibe un dato de tipo `T` y no genera ningún resultado
  - `Function<T,R>`: toma un dato de tipo `T` y genera un resultado de tipo `R`
  - `Predicate<T>`: toma un dato de tipo `T` y evalúa si el dato cumple una condición
  - `Supplier<T>`: no recibe ningún dato, pero genera un dato de tipo `T` cada vez que es invocado
  - `UnaryOperator<T>` recibe un dato de tipo `T` y genera un resultado de tipo `T`

SAM y FunctionalInterface

- **Single Abstract Method (SAM)**: Es una interfaz que solo tiene un método sin definir.
- Con la etiqueta **@FuntionalInterface** defino que la interfaz va a ser una interfaz funcional.

Ejemplo interfaz funcional personalizada:

```
@FunctionalInterface
interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

Donde: **T,U,V** son los tipos datos de los parámetros de entrada (En este caso son 3 parámetros de entrada) y **R** es el tipo de dato del parámetro de salida.

Ejemplo:

```java
Function<Integer, String> addZeros = number -> number < 10 ? "0" + number : String.valueOf(number);

TriFunction<Integer, Integer, Integer, LocalDate> parseDate =
        (day, month, year) -> LocalDate.parse(year + "-" + addZeros.apply(month) + "-" + addZeros.apply(day));

TriFunction<Integer, Integer, Integer, Integer> calculateAge =
        (day, month, year) -> Period.between(parseDate.apply(day, month, year), LocalDate.now()).getYears();

calculateAge.apply( t: 31,  u: 10,  v: 1997); //22
```

En este ejemplo **calculateAge** recibe 3 **Integers,** convierte los valores a un formato **LocalDate**, calcula los años entre el periodo de tiempo actual y el ingresado y retorna la cantidad de años.

Operador de Referencia

- Es el que simplifica la forma de ejecutar un método que no tiene retorno y que recibe la cantidad de argumentos de la función que lo está llamando.

```java
profesores.forEach(profesor -> System.out.println(profesor));

/**
 *  La lambda de arriba se reemplaza por el argumento del forEach de abajo.
 *  */
profesores.forEach(System.out::println);
```

Method References

Es una sintaxis alternativa para las expresiones lambda.

Ejemplo 1:

Function<Person, Integer> f = person-> person.getAge(); **=> lambda normal**

Function<Person, Integer> f = Person::getAge; **=> lambda method reference**

Ejemplo 2:

BinaryOperator<Integer> sum= (i1, i2) -> i1 +i2;

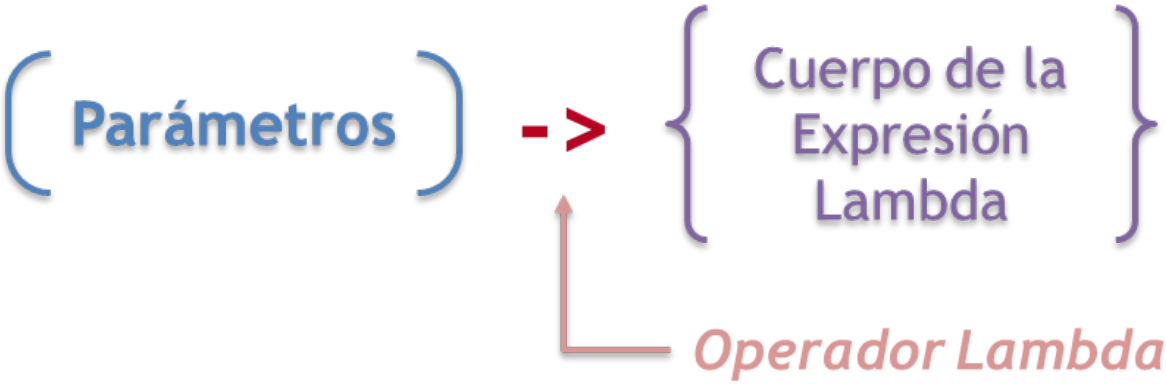
= (i1, i2) -> Integer.sum(i1, i2) ;

BinaryOperator<Integer> sum= Integer::sum;

Inferencia de tipos

- Java puede identificar de una función anónima que tipo de parámetro y retorno tiene y con ello darnos acceso a ese tipo en la siguiente función.
- La inferencia del tipo, Java lo hace es que en tiempo de ejecución valida el valor que se pasa sea del tipo que se requiere.

Sintaxis de las funciones lambda



Métodos default en las interfaces

- La palabra reservada **default** nos permite definir un cuerpo para un método dentro de una interfaz.
- Se suelen utilizar en interfaces funcionales donde solo pueden tener un método abstracto.
- No se de debe de abusar de su uso.

Chaining

- Encadenar el resultado de una ejecución con respecto a otra ejecución.
- Cada ejecución devuelve el valor de la instancia para que pueda ser tomada por la siguiente ejecución del encadenamiento.
- Chaining es la estrategia de retornar siempre un objeto, tal que puedas invocar métodos con cada invocación.
- Es la práctica de llamar a diferentes métodos en una sola línea en lugar de llamar a diferentes métodos con la misma referencia de objeto por separado. Bajo este procedimiento, tenemos que escribir la referencia del objeto una vez y luego llamar a los métodos separándolos con un (punto).

  Ejemplo 1:

```java
static class Numbers {  Complexity is 3 Everything is cool!

    private int a;
    private float b;

    Numbers() {
        System.out.println("Calling The Constructor");
    }

    public Numbers setInt(int a) {
        this.a = a;
        return this;
    }

    public Numbers setFloat(float b) {
        this.b = b;
        return this;
    }

    void display() {
        System.out.println("Display= int(" + a + ") float(" + b + ")");
    }
}
```

```java
new ChainingExample
        .Numbers()
        .setInt(10)
        .setFloat(20)
        .display();//Display= int(10) float(20.0)
```

Ejemplo 2:

Método que recibe una palabra y un número que dice cuántos caracteres de derecha a izquierda remover. La palabra se convierte a *uppercase*, remueve la cantidad de caracteres ingresados, reemplaza la letra **'A'** por **'x'** e imprime el resultado.

```java
wordUtilExample( word: "Maracuya",  charsToRemove: 4);
```

```java
static void wordUtilExample(String word, int charsToRemove) {  Complexity is 7 It's time to do something...

    Function<String, String> uppercaser = String::toUpperCase;

    BiFunction<Function<String, String>, Integer, Function<String, String>> rightRemover =
            (txt, num) -> txt.andThen(x -> {
                int removePosition = (x.length() - num);
                return x.substring(0, removePosition);
            });

    System.out.println(
            rightRemover
                    .apply(uppercaser, charsToRemove)
                    .andThen(t -> t.replaceAll( regex: "A",  replacement: "x"))
                    .apply(word)
    );//MxRx
}
```

**Yo se que muchas responsabilidades para un método, pero lo hice para efectos demostrativos.*

Composición

- El método **compose** toma un función (**lambda**), ejecuta esa función primero y después ejecuta la función desde dónde se mandó a llamar (**funcion**).
    - **funcion.compose(lambda)**

Ejemplo:

```java
Function<Integer, Integer> multiplyBy3 = x -> {
    System.out.println("Multiplicando x: " + x + " por 3, ejecutando multiplyBy3");
    return x * 3;
};

Function<Integer, Integer> add1MultiplyBy3 =
        multiplyBy3.compose( y -> {
            System.out.println("Le agregare 1 a: " + y + ", ejecutando add1MultiplyBy3");
            return y+1;
        });

Function<Integer, Integer> addSquare =
        add1MultiplyBy3.andThen(x -> {
            System.out.println("Estoy elevando " + x + " al cuadrado, ejecutando addSquare");
            return x * x;
        });
```

```
addSquare.apply( t: 3);

/**
 * Le agregare 1 a: 3, ejecutando add1MultiplyBy3
 * Multiplicando x: 4 por 3, ejecutando multiplyBy3
 * Estoy elevando 12 al cuadrado, ejecutando addSquare
 * 144*/
```

Optionals (https://www.baeldung.com/java-optional)

- El objetivo de la clase **Optional** es que a partir de las funciones que tengamos no nos debemos de preocupar por el valor del retorno.
- Las clase **Optional** es la forma de meter un dato y operar sobre este estando o no presente, con eso podemos generar valores por **default**.
- Tu código debería usar únicamente **Optional** como resultado de una función, nunca como entrada.
- **Optional.ofNullable()** le dice a **Optional** que tenemos un dato del cual desconocemos si es nulo o no. **Optional** lo mantiene como escondido para evitar un error **NullPonierException**. Otra opción sería hacer un **try catch** que devuelva un **Optional.of()** o sino un **Optional.empty()**.

Ejemplo:

```
static Optional<List<String>> getOptionalNames() {
    List<String> nameList = new LinkedList<>();

    return Optional.of(nameList);
}

static Optional<String> optionalValuePlayer() {  Complexity is 4 Everything is cool!
    //return Optional.ofNullable(); o

    try {
        //Obtencion de datos
        return Optional.of("David");
    } catch (Exception e) {
        e.printStackTrace();
    }
    return Optional.empty();
}
```

```java
Optional<List<String>> optionalNames = getOptionalNames();

if (optionalNames.isPresent()) {}

optionalNames.ifPresent(namesValue → namesValue.forEach(System.out::println));

Optional<String> valuablePlayer = optionalValuePlayer();

String valuablePlayerName = valuablePlayer.orElseGet(() → "No hay jugador valioso");
```

Currying

- Reducir la complejidad de una función partiendola en subfunciones, se le conoce como currying.
- Currying es una manera de crear funciones más dinámicas basados en la reducción de parámetros.
-
```java
Function<Integer, Function<String, Function<Double, String>>> curried = curryThree(threeFunction);

curried.apply( t: 1)
        .apply( t: "")
        .apply( t: 0.0);
```
-
```java
static <F, S, T, R> Function<F, Function<S, Function<T, R>>> curryThree(ThreeFunction<F, S, T, R> threeFunction) {
    return first ->
            second ->
                    third -> threeFunction.apply(first, second, third);
}
```

Stream

- Un **Stream** es como una lista que tiene elementos que se pueden iterar y ejecutar un conjunto de funciones de manera anidada.
- El **Stream** es auto-iterable.
- Un **Stream** solo puede ser consumido una única vez.
- Podemos ver o imaginar a un stream como un flujo de datos, un río de datos. Donde los datos van moviéndose sin esperar a que alguien los mueva.
- Podemos generar **Streams** de cualquier fuente.

Ejemplo:

```
List<String> courseList = StreamsExample.getList(
        …elements: "Java",
        "Inteligencia Artificial",
        "Desarrollo movil",
        "Scala"
);

Stream<List<String>> courseStream = Stream.of(courseList);
Stream<String> markCourse = courseStream.map(course → course + "!");
Stream<String> justScalaCourse = markCourse.filter(course → course.contains("Scala"));
```

Stream listeners

- Los Stream listeners nos permiten operar sobre el mismo **stream** usando el **chaining** sin tener que esta almacenando los flujos en variables.

Ejemplo:

```
static <T> Stream<T> addOperator(Stream<T> stream) {  Complexity is 4 Everything is cool!
    return stream.peek(data → System.out.println("Dato: " + data));
}
```

```
Stream<String> coursesStream2 = courseList.stream();

addOperator(
        coursesStream2.map(course → course + "!")
        .filter(course → course.contains("Java"))
).forEach(System.out::println);//Java!
```

[Operaciones y Collectors]

Streams de tipo específico y Paralelismo

- Existen streams que son de algún tipo en específico, como lo es el **IntStream** que desde un iterador genera un stream de números enteros.
- **.parallel()** permite que la ejecución de un **stream** sea de manera concurrente, es decir, que los procesos se van a ejecutar de manera paralela según la cantidad de núcleos del procesador y va devolviendo los datos de manera **desorganizada**. Al final **stream** recolecta los resultados en un solo elemento.
- **.parallel()** funciona muy bien y es recomendable usarlo cuando se manejan muchos datos, de lo contrario, lo mejor es no usarlo.

Ejemplo:

```
IntStream infiniteStream = IntStream.iterate( seed: 0, x → x + 1);
infiniteStream.limit(1000)
        .parallel()
        .filter(number → number % 2 == 0)
        .forEach(System.out::println);
```

Operaciones Terminales (Esta documentación también se encuentra en *FInalOperations.class* dentro de los archivos del curso)

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

Operaciones Intermedias (Esta documentación también se encuentra en *IntermediateOperations.class* dentro de los archivos del curso)

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

Collectors

- Cuando necesitemos retornar el resultado de un stream cualquier otro tipo de de dato usamos los **Collectors**.
- **.boxed()** convierte los elementos a un **stream** de datos (**Stream<T>**) para que luego puedan ser devueltos a través de los **Collectors**.

Ejemplo:

```
IntStream infiniteStream = IntStream.iterate(seed: 0, x → x + 1);

List<Integer> numbersList = infiniteStream.limit(1000)
        .filter(number → number % 2 == 0)
        .boxed()
        .collect(Collectors.toList());
```

Librerías para el proyecto

Enlaces de las librerías:

- JCommander: Esta librería toma los argumentos que se le pasan por consola y los convierte a objetos de Java.
- Feign Core: Esta librería hace peticiones web.

- Feign Gson: Esta librería convierte los JSON que recibe de la petición a objetos y clases en Java.