PROGRAMACIÓN ORIENTADA A OBJETOS

CON JAVA 17

ACADEMIA CONSULTEC

www.consultec-ti.com

VALENTÍA | AGILIDAD | EVOLUCIÓN | OPTIMIZACIÓN

Agenda **del día**

3

- Colecciones
- Programación funcional
- Serialización
- Anotaciones

ACADEMIA CONSULTEC

Colecciones

ACADEMIA CONSULTEC

Arrays vs Colecciones

Los Arrays y Collections se utilizan para almacenar un grupo de objetos del mismo o diferente tipo, pero la **diferencia** fundamental entre ambas, es que, mientras que los arrays tienen un tamaño fijo, las colecciones en java son totalmente dinámicas

Ventajas de las *colecciones* respecto a los *arrays*.

1. Cambiar el tamaño dinámicamente, y pueden estar ordenadas y desordenadas.
2. Mayor rendimiento.
3. Puede contener elementos primitivos, wrapper y objetos.
4. Reservan almacenamiento adicional para nuevos elementos.
5. Ofrece varios métodos de utilidad para facilitar las operaciones en su estructura de datos subyacente.
6. No tiene concepto de dimensiones.
7. Pueden almacenar datos homogéneos y heterogéneos, admiten genéricos para garantizar la seguridad de tipos.
8. Algunas colecciones permiten elementos duplicados y otras no.

ACADEMIA CONSULTEC

Arrays

Un array es una estructura de datos que nos permite almacenar una gran cantidad de datos de un mismo tipo. El tamaño de los arrays se declara en un primer momento y no puede cambiar en tiempo de ejecución como puede producirse en otros lenguajes.

Los arrays se numeran desde el elemento cero, que sería el primer elemento, hasta el tamaño-1 que sería el último elemento. Es decir, si tenemos un array de diez elementos, el primer elemento sería el cero y el último elemento sería el nueve.

Métodos de Array:

1. length: devuelve el número de elementos en el arreglo.
2. sort: ordena los elementos del arreglo.
3. fill: rellena el arreglo con un valor específico.
4. toString: devuelve una representación en cadena del arreglo.

Arrays

Ejemplos de Array:

```
//Arreglo unidirecional
String[] daysWeek = {"Lunes", "Martes", "Miercoles",
"Jueves", "Viernes", "Sabado"};
//for normal
for (int i = 0; i < daysWeek.length; i++) {
    System.out.println(daysWeek[i]);
}
//utilizamos un for-each
for (String day : daysWeek) {
    System.out.println(day);
}
```

En consola aparece:

```
Lunes
Martes
Miercoles
Jueves
Viernes
Sabado
```

ACADEMIA CONSULTEC

Arrays

Ejemplos de Array:

```
//Arreglo Bidirecional
String[][] cities = new String[4][2]; //4 * 2 = 8
cities[0][0] = "Colombia";
cities[0][1] = "Medellín";
cities[1][0] = "Colombia";
cities[1][1] = "Bogotá";
cities[2][0] = "México";
cities[2][1] = "Guadalajara";
cities[3][0] = "México";
cities[3][1] = "CDMX";
for (String[] row : cities) {
    for ( String col :row) {
        System.out.println(col);
    }
}
```

En consola aparece:

```
Colombia
Medellín
Colombia
Bogotá
México
Guadalajara
México
CDMX
```

ACADEMIA CONSULTEC

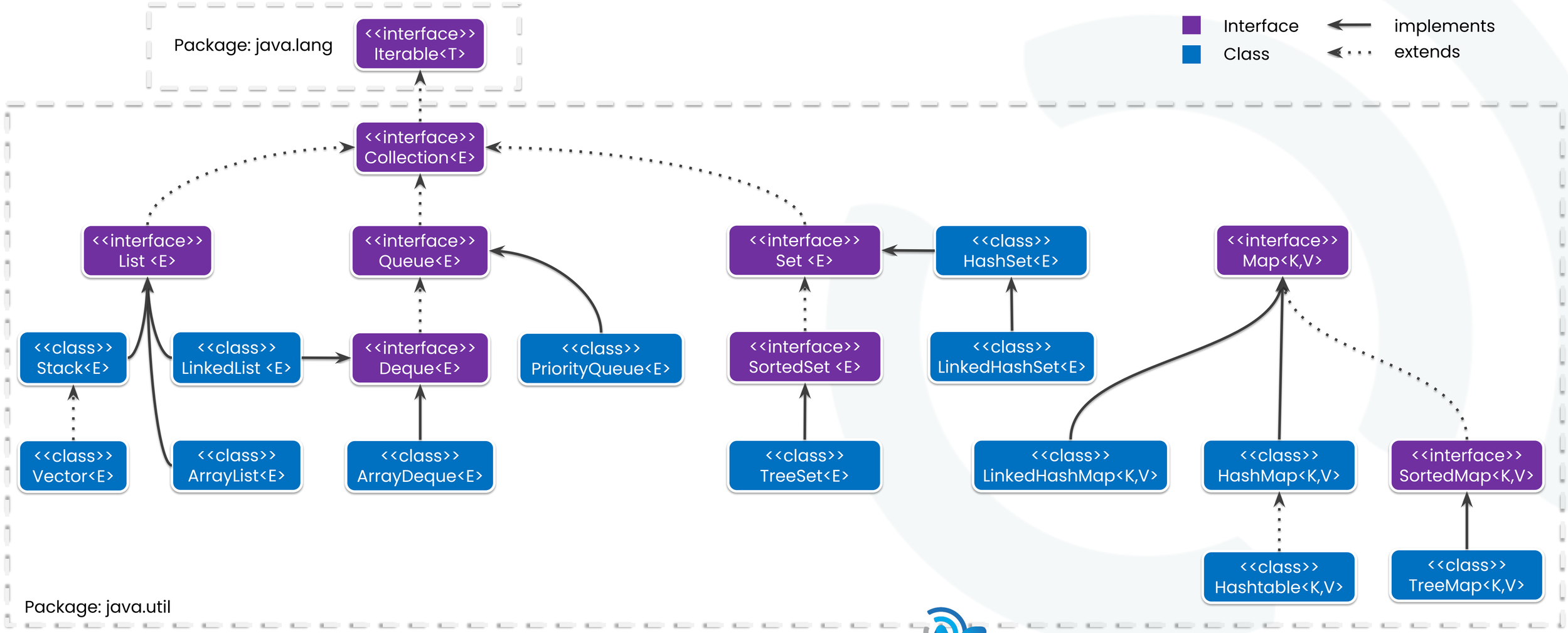
Colecciones

Java 8 introdujo las expresiones *Lambda* y los *Streams* que fueron un salto muy importante en Java. Sin embargo como una tecnología nueva quedaron muchas cosas por añadir y fue únicamente un primer paso, ahora desde Java 9 hace uso del Framework de colecciones de una forma mucho más natural.

¿Cómo funcionan las colecciones?

- Una colección representa un grupo de objetos.
- Estos objetos son conocidos como elementos.
- Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos.
- En Java, se emplea la interfaz genérica Collection para este propósito.
- Con la interfaz Collection podemos almacenar cualquier tipo de objeto.
- Podemos usar una serie de métodos comunes como añadir, eliminar, obtener el tamaño de la colección.

Jerarquía del Collection Framework



ACADEMIA CONSULTEC

Set

Define el concepto de conjunto, y es una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos.

Clases que implementan Set:

- **HashSet:** esta implementación almacena los elementos en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.
- **TreeSet:** esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento que **HashSet**. Los elementos almacenados deben implementar la interfaz **Comparable**.
- **LinkedHashSet:** esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más costosa que **HashSet**.

Ninguna de estas implementaciones son sincronizadas.

Set

```
Set<Integer> intTreeSet = new TreeSet<Integer>();

intTreeSet.add(5); intTreeSet.add(4);
intTreeSet.add(3); intTreeSet.add(2);
intTreeSet.add(1);

for (int number : intTreeSet) {
    System.out.println(number);
}
```

En consola aparece:

```
1
2
3
4
5
```

ACADEMIA CONSULTEC

Set

```
HashSet<String> stringHashSet =new HashSet();
stringHashSet.add("Alberto"); stringHashSet.add("Pancho");
stringHashSet.add("Zuniga"); stringHashSet.add("Humberto");
stringHashSet.add("Eduardo");
System.out.println("Hashset desordenada: ");
for (String cadena : stringHashSet) {
    System.out.println(cadena);
}

System.out.println("Hashset ordenada: ");
HashSet<String> stringHashSetSorted =
stringHashSet.stream().sorted().collect(Collectors.toCollection(LinkedHashSet::n
ew));
for (String cadena : stringHashSetSorted) {
    System.out.println(cadena);
}
```

En consola aparece:

```
Hashset desordenada:
Pancho
Eduardo
Zuniga
Humberto
Alberto
Hashset ordenada:
Alberto
Eduardo
Humberto
Pancho
Zuniga
```

ACADEMIA CONSULTEC

List

Define una sucesión de elementos, y a diferencia de la interfaz Set, esta interfaz sí admite elementos duplicados. Puede almacenar objetos repetidos, y cada uno de los objetos o elementos se encuentran indexados a través de un valor numérico, por tanto este tipo de colección, permite acceder de forma aleatoria a un elemento.

A parte de los métodos heredados de *Collection*, añade métodos que permiten mejorar: *Acceso posicional a elementos*, *Búsqueda de elementos, Iteración sobre elementos y Rango de operación.*

Clases que implementan List:

- **ArrayList:** esta es la implementación típica. Se basa en un array *redimensionable* que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.
- **LinkedList:** esta implementación se basa en una *lista doblemente enlazada* de los elementos, teniendo cada uno de los elementos un *puntero al anterior* y al *siguiente elemento*.
- **Vector:** Es la implementación más antigua y existe desde los inicios de Java. Se usa muy poco porque se trata de una estructura de *Lista* a la que siempre se accede de forma *sincronizada* el cual reduce el rendimiento.

List

```
// // Creo un Array tipo String
ArrayList<String> mascotaList = new ArrayList<>();
mascotaList.add("Gato");
mascotaList.add("Perro");
// Agrego un valor por el índice
mascotaList.add(1, "Loro");
System.out.println("ArrayList inicial " + mascotaList);

mascotaList.remove(2);
System.out.println("ArrayList actualizado " + mascotaList);

// Modifico el elemento
mascotaList.set(0, "Mapache");
System.out.println("ArrayList actualizado " + mascotaList);
```

En consola aparece:

```
ArrayList inicial [Gato, Loro, Perro]
ArrayList actualizado [Gato, Loro]
ArrayList actualizado [Mapache, Loro]
```

ACADEMIA CONSULTEC

List

```
LinkedList<String> linkedList = new LinkedList<String>();
linkedList.add("A");
linkedList.add("B");
linkedList.push("F");
linkedList.push("G");
System.out.println("linkedList inicial: " + linkedList);

linkedList.addLast("C");
linkedList.addFirst("D");
linkedList.add(2, "E");
System.out.println("linkedList actualizado" + linkedList);

linkedList.remove("B");
linkedList.remove(3);
linkedList.removeFirst();
linkedList.removeLast();
for (String i : linkedList) {
    System.out.println(i);
}
```

En consola aparece:

```
linkedList inicial: [G, F, A, B]
linkedList actualizado[D, G, E, F, A, B, C]
G
E
A
```

ACADEMIA CONSULTEC

Queue

También denominadas colas, por lo que no se puede acceder a un elemento de manera aleatoria. Es decir, solo podemos acceder a un objeto que se encuentre o bien al principio, bien al final. Se trata de una lista ordenada de objetos cuyo uso se limita a la inserción de elementos al final de la lista y a la eliminación de elementos desde el principio de la lista (es decir, sigue el principio FIFO o de "primero en entrar, primero en salir").

Clases que implementan Map:

- **LinkedList**: es una estructura de datos lineal en la que los elementos no se almacenan en ubicaciones contiguas y cada elemento es un objeto independiente con una parte de datos y una parte de direcciones. Los elementos se enlazan mediante punteros y direcciones. Cada elemento se conoce como nodo.
- **PriorityQueue:** se sabe que una cola sigue el algoritmo Primero en entrar, primero en salir, pero a veces es necesario que los elementos de la cola se procesen según la prioridad.
- **PriorityBlockingQueue**: es una cola de bloqueo no limitada que utiliza las mismas reglas de ordenación que la clase *PriorityQueue* y proporciona operaciones de recuperación de bloqueo. Dado que es ilimitada, la adición de elementos a veces puede fallar debido al agotamiento de los recursos que resulta en *OutOfMemoryError*.

Queue

```
Queue<Integer> q = new LinkedList<>();
for (int i = 0; i < 5; i++)
    q.add(i);
System.out.println("Elementos de la cola: "  + q);

int removedele = q.remove();
System.out.println("Element removido: " + removedele);
System.out.println("Cola actual: "  + q);

// Para ver el primero de la cola
int head = q.peek();
System.out.println("El primero de la cola: " + head);

//Retiro un elemento de la cola
int elemt = q.poll();
System.out.println("Elemento que se retira: " + elemt);
int size = q.size();
System.out.println("Tamano actual de la cola: " + size);
System.out.println("Cola actual: "  + q);
```

En consola aparece:

```
Elementos de la cola: [0, 1, 2, 3, 4]
Element removido: 0
Cola actual: [1, 2, 3, 4]
El primero de la cola: 1
Elemento que se retira: 1
Tamano actual de la cola: 3
Cola actual: [2, 3, 4]
```

ACADEMIA CONSULTEC

Queue

```java
Queue<Integer> pbq = new PriorityBlockingQueue<Integer>();
pbq.add(10);
pbq.add(20);
pbq.add(15);
System.out.println("Cola actual: " + pbq);

// Se ubica quien es el primer elemento
System.out.println(pbq.peek());
System.out.println("Cola actualizada: " + pbq);

// Se retira el primer elemento
System.out.println(pbq.poll());
System.out.println("Cola actualizada: " + pbq);
```

En consola aparece:

```
Cola actual: [10, 20, 15]
10
Cola actualizada: [10, 20, 15]
10
Cola actualizada: [15, 20]
```

ACADEMIA CONSULTEC

Map

Esta interfaz emplea el concepto de Mapa o diccionario, por tanto, define los conceptos de clave y valor. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.

Clases que implementan Map:

- **HashMap:** esta implementación almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeMap:** esta implementación almacena las claves ordenándolas en función de sus valores. Es bastante más lento que HashMap. Las claves almacenadas deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashMap:** esta implementación almacena las claves en función del orden de inserción, es un poco más costosa que HashMap.

Map

```
HashMap<Integer, String> hm = new HashMap<>();
hm.put(1, "C");
hm.put(2, "C++");
hm.put(3, "Java");
hm.put(4, "JavaScript");
hm.put(5, "Python");
hm.put(6, "PHP");
System.out.println("HashMap inicial: " + hm);

hm.remove(4);
System.out.println("HashMap remuevo clave 4: " + hm);
hm.put(6, "Ruby");
System.out.println("HashMap actualizo clave 6: " + hm);

// Creando un nuevo mapa a partir de otro
HashMap<Integer, String> language = new HashMap<Integer, String>(hm);
System.out.println("Valor de Lenguajes: " + language);
for (Map.Entry<Integer, String> e : hm.entrySet())
    System.out.println("Key: " + e.getKey() + " Value: " + e.getValue());
```

En consola aparece:

```
HashMap inicial: {1=C, 2=C++, 3=Java, 4=JavaScript, 5=Python, 6=PHP}
HashMap remuevo clave 4: {1=C, 2=C++, 3=Java, 5=Python, 6=PHP}
HashMap actualizo clave 6: {1=C, 2=C++, 3=Java, 5=Python, 6=Ruby}
Valor de Lenguajes: {1=C, 2=C++, 3=Java, 5=Python, 6=Ruby}
Key: 1 Value: C
Key: 2 Value: C++
Key: 3 Value: Java
Key: 5 Value: Python
Key: 6 Value: Ruby
```

ACADEMIA CONSULTEC

Map

```
SortedMap<Integer, String> tm = new TreeMap<Integer, String>();
tm.put(3, "Miercoles");
tm.put(2, "Martes");
tm.put(1, "Lunes");
tm.put(5, "Viernes");
tm.put(4, "Jueves");
System.out.println("SortedMap inicial: " + hm);

for (Integer key : tm.keySet()) {
    System.out.println( "Values: " + tm.get(key));
}

for (Map.Entry mapElement : tm.entrySet()) {
    int key = (int)mapElement.getKey();
    // Finding the value
    String value = (String)mapElement.getValue();
    System.out.println(key + " : " + value);
}
```

En consola aparece:

```
Values: Lunes
Values: Martes
Values: Miercoles
Values: Jueves
Values: Viernes
1 : Lunes
2 : Martes
3 : Miercoles
4 : Jueves
5 : Viernes
```

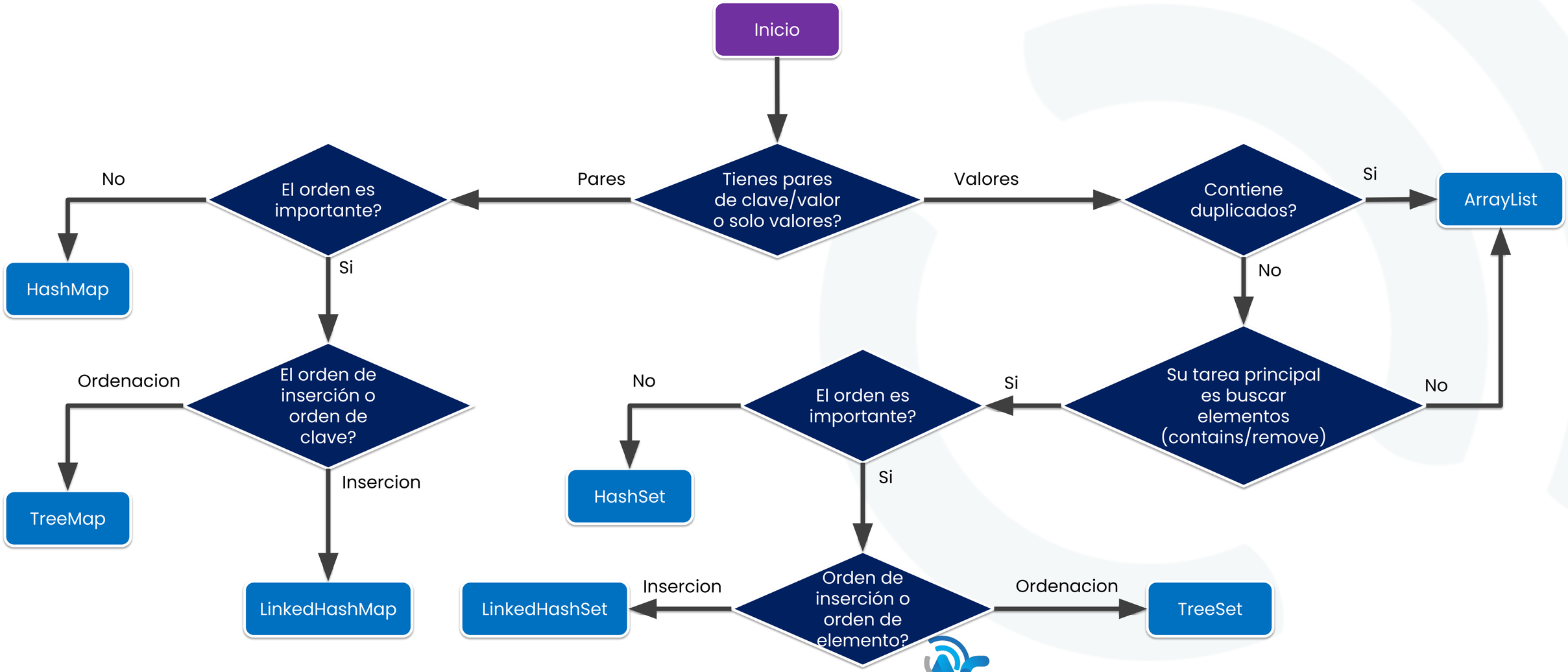
ACADEMIA CONSULTEC

Diferencia entre List/Set/Map

Collection	Ordering	Random Access	Key-Value	Duplicate Element	Null Element	Thread Safety
ArrayList	YES	YES	NO	YES	YES	NO
LinkedList	YES	NO	NO	YES	YES	NO
HashSet	NO	NO	NO	NO	YES	NO
TreeSet	YES	NO	NO	NO	NO	NO
HashMap	NO	YES	YES	NO	YES	NO
TreeMap	YES	YES	YES	NO	NO	NO
Vector	YES	YES		YES	YES	YES
HashTable	NO	YES	YES	NO	YES	YES
Properties	NO	YES	YES	NO	YES	YES
Stack	YES	NO	NO	YES	YES	YES
CopyOnWriteArrayList	YES	YES	NO	YES	YES	YES
ConCurrentHashMap	NO	YES	YES	NO	NO	YES
CopyOnWriteArraySet	NO	NO	NO	NO	YES	YES

ACADEMIA CONSULTEC

Que colección usar?



ACADEMIA CONSULTEC

UnmodifiableMap

Los mapas que no admiten operaciones de modificación se denominan unmodifiable, y fue introducido en Java 8 como un método de fabrica Collectors.unmodifiableMap(). Los mapas no modificables suelen ser vistas *read-only (wrappers)* de otros mapas modificables. No podemos añadirlos, eliminarlos o borrarlos, pero si cambiamos el mapa subyacente, la vista de este mapa también cambia.

Los mapas no modificables (unmodifiable) garantizan que ningún cambio en el objeto mapa subyacente sea visible, siempre y cuando no se tenga referencia del mapa original.

El método estático de fábrica toma un Map como parámetro y devuelve una vista no modificable de tipo java.util.Collections$UnmodifiableMap.

UnmodifiableMap

```
Map<String, String> mutableMap = new HashMap<>();
mutableMap.put("USA", "North America");
mutableMap.put("Panama", "Ciudad de Panama");
System.out.println("Lista Pais: " + mutableMap);

Map<String, String> unmodifiableMap = Collections.unmodifiableMap(mutableMap);
unmodifiableMap.put("Colombia", "Bogota");
```

En consola aparece:

```
Lista Pais: {USA=North America, Panama=Ciudad de Panama}
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.base/java.util.Collections$UnmodifiableMap.put(Collections.java:1505)
    at collections.InmutableMapTest.main(InmutableMapTest.java:25)
```

Factory Methods

Uno de los cambios más importantes es el uso de *Factory Methods* en Java 9. Siendo normalmente una rutina tener que instanciar una clase, construir la lista y recorrerla para manipularla, en la versión 8 de Java proveyeron para solventar este problema el uso de la clase Arrays y su método asList() que nos permite simplificar el código. Sin embargo esto solo nos sirve para las listas y existen otros tipos de colecciones que se usan muy habitualmente como son *Set* y *Map*.

Para solventar estos problemas existen los *Factory Methods*, el cual nos ayudará a trabajar de una forma más cómoda con la inicialización de colecciones, para esto hacemos uso de los métodos estáticos of().

```
List<String> listaList   = List.of("list1","list2","list3","list4");
Set<String> listaSet     = Set.of("set1","set2","set3","set4");
Map<Integer,String> mapa = Map.of(1,"Java",2,"JavaScript",3,"Kotlin");
// En caso de no tener JDK superior a la version 8, puede hacer uso de Stream.of()
List<String> lista= Stream.of("hola","que","tal","estas").collect(Collectors.toList());
```

ACADEMIA CONSULTEC

Factory Methods (ImmutableMap)

Immutable Map como sugiere su nombre, es un tipo de mapa inmutable, esto significa que el contenido del mapa es fijo o constante después de la declaración, es decir, son de sólo lectura. Estas configuraciones fueron agregadas al método of(), por tanto toda instancia creada es inmutable, utilizando este método, podemos crear un mapa inmutable que contenga cero o hasta 10 pares clave-valor.

Excepciones de ImmutableMap :

- Si se intenta añadir, borrar o actualizar elementos en el Mapa, se lanza una ***UnsupportedOperationException***.
- No se permiten elementos nulos, por tanto, si se intenta crear un *ImmutableMap* con un elemento nulo, se lanza una **NullPointerException.**
- Si se intenta añadir un elemento nulo al mapa, se produce una excepción de tipo **UnsupportedOperationException**.

Ventajas de ImmutableMap:

- Son seguros para los hilos.
- Son eficientes en memoria.
- Como son inmutables, pueden ser pasados a librerías de terceros sin ningún problema.

Iterable

La reseña más importante en JDK 8 es el añadir el interface *Iterable* por encima del interface *Collection*, el cual es el que nos permite integrar de forma más natural la programación funcional dentro del lenguaje Java.

Métodos del interface Iterable:

- **iterator()**: Este método devuelve un iterador y nos permite recorrer la colección de elementos que tengamos.

```
Iterator<E> iteratorVar = [Collection].iterator();
```

- **forEach(Consumer<? super T> action)**: El método más importante a nivel de colecciones ya que permite recorrer la colección como si se tratara de un Java Stream. A veces nos provoca equívocos pero su utilidad es grande.

```
listaIterable.forEach([action]);
```

ACADEMIA CONSULTEC

Iterable

Método iterator():

```
ArrayList<String> lista= new ArrayList<String>();
lista.add("prueba");
lista.add("de");
lista.add("iterador");
lista.add("sencillo");

Iterator<String> it = lista.iterator();
// con un iterador
while(it.hasNext()) {
    System.out.println(it.next());
}

// sintacticamente es menos limpio
for (int i=0 ; i<lista.size() ; i++) {
    System.out.println(lista.get(i));
}
```

En consola aparece:

```
prueba
de
iterador
Sencillo

prueba
de
iterador
sencillo
```

ACADEMIA CONSULTEC

Iterable

Método forEach ():

```
Iterable<String> conjunto = Set.of("tal", "que", "hola");
conjunto.forEach(System.out::println);

//Transformo a List para acceder por indice y ordenar
List<String> nuevaLista= new ArrayList<>();
nuevaLista.add("Zuca");
conjunto.forEach(nuevaLista::add);
System.out.println("Elemento 3 sin ordenar: " + nuevaLista.get(3));

//Ordeno la Lista
Collections.sort(nuevaLista);
System.out.println("Elemento 3 ordenado: " + nuevaLista.get(3));

//Al ser una lista ya puedo acceder por indice
for (int i=0;i<nuevaLista.size();i++) {
    System.out.println(nuevaLista.get(i));
}
```

En consola aparece:

```
tal
que
Hola

Elemento 3 sin ordenar: hola
Elemento 3 ordenado: tal

Zuca
hola
que
tal
```

ACADEMIA CONSULTEC

Varargs y Colecciones

Argumentos Variables (varargs) son métodos que permiten recibir un número variable de parámetros, permitiendo trabajar de una forma más cómoda y simple, la aplicación de esta estructura fue introducido desde JDK 5.

Internamente, el método **varargs** se implementa utilizando el concepto de arrays unidimensionales. Por lo tanto, en el método **varargs**, podemos diferenciar argumentos utilizando **Index**, un argumento de longitud variable se especifica mediante tres puntos o puntos suspensivos **(...)**.

```
public static void methodName(datatype... arg)
{
    // usar un loop para obtener el contenido
    for (datatype i : a)
        // ...
}
```

ACADEMIA CONSULTEC

Varargs y Colleciones

```
static void varargsMethod(int... a){
    System.out.println("Numero de argumentos: " + a.length);
    // recorro la lista
    for (int i : a)
        System.out.print(i + " ");
    System.out.println();
}

public static void main(String[] args) {
    // Se envia un solo argumento
    varargsMethod(100);
    // 4 parameters
    varargsMethod(1, 2, 3, 4);
    // no se envian argumentos
    varargsMethod();
}
```

En consola aparece:

```
Number of arguments: 1
100
Number of arguments: 4
1 2 3 4
Number of arguments: 0
```

ACADEMIA CONSULTEC

Receso

15 min

Programación

Funcional

ACADEMIA CONSULTEC

Programación Imperativa vs Declarativa

Las características de un lenguaje de programación dictaminan a que paradigma de programación pertenecen. Por tanto, hay que diferenciar entre 2 paradigmas utilizados en java, el cuales son:

- **Programación Imperativa:** este tipo de programación detalla las instrucciones necesarias para llegar a la solución. Es la más habitual, se centra en decirle al lenguaje de programación cada uno de los pasos a seguir (instrucción a instrucción), con la finalidad de que estos algoritmos, cambien el estado del programa.
  - Pregunta **¿Qué hacer? ¿Cómo hacerlo?**

- **Programación Declarativa:** este tipo de programación describe el problema que se quiere solucionar, pero no las instrucciones necesarias para hacerlo. Algunos de los beneficios de este tipo de programación son que es compacto, otorga más fiabilidad (picamos menos, fallamos menos), reduce los costes y es reutilizable.
  - Pregunta únicamente **¿Qué hacer? y no como hacerlo.**

ACADEMIA CONSULTEC

Programación Funcional

Programación funcional es un paradigma de programación, específicamente de tipo declarativo pero de enfoque imperativo, que tiene como objetivo simplificar, reducir y dividir la mayor cantidad de tareas y funciones en pequeñas partes.

Este modelo está enfocado en que vas a resolver y no tanto en como resolverlo, por tanto, nosotros expresaremos nuestra lógica sin describir controles de flujo ni usaremos ciclos o condicionales, aunque se trabaja principalmente con funciones, evitaremos los datos mutables, así como el hecho de compartir estados entre funciones.

Conceptos aplicados en programación funcional:

- Funciones puras.
- Composición de funciones.
- Estados compartidos.
- Mutabilidad.
- Efecto secundario.

ACADEMIA CONSULTEC

Funciones Puras

Las funciones puras, como indica su nombre son funciones, el cual dando el mismo input siempre retornan el mismo output, además de no tener efectos secundarios. Y tienen las siguientes características:

- Devuelven un resultado
- No tienen efectos secundarios
- Al menos tienen un parámetro de entrada
- Siempre devuelven el mismo resultado si se usan los mismos parámetros

Estas funciones son independientes del estado externo, por lo que resultan muy fáciles de refactorizar o cambiar su lógica sin introducir errores en otras partes del código. Ejemplo:

```
public static int sum(int x, int y) {
    return x + y;
}
```

ACADEMIA CONSULTEC

Funciones Puras

```
public static boolean disponeFondos(double funds) {
    return funds > 0.0;
}

static class Cliente {
    private String nombre;
    private double balance;
    public Cliente(String nombre, double balance) {
        this.nombre = nombre;
        this.balance = balance;
    }
    public String getNombre() { return nombre;  }
    public double getBalance() { return balance; }
}
```

```
public static void main(String[] args) {
    Cliente albert = new Cliente("Albert",-20.00);
    System.out.println(albert.getNombre() + " dispone de fondo? "
            + disponeFondos(albert.getBalance()));

    Cliente ricardo = new Cliente("Ricardo",1300.00);
    System.out.println(ricardo.getNombre() + " dispone de fondo? "
            + disponeFondos(ricardo.getBalance()));
}
```

En consola aparece:

```
Albert dispone de fondo? false
Ricardo dispone de fondo? true
```

ACADEMIA CONSULTEC

Composición de funciones

La composición de funciones es el proceso de combinar dos o más funciones, teniendo como finalidad ejecutar cada una de estas funciones en secuencia para obtener un resultado en concreto.

```
public static double x2(double number){
        return (number * 2);
}
public static double add2(double number){
    return (number + 2);
}
public static void main(String[] args) {
     double number = add2(x2(4));
     System.out.println("El resultado de la funcion compuesta es: " + number);
}
```

En consola aparece:

```
El resultado de la funcion compuesta es: 10.0
```

ACADEMIA CONSULTEC

Estado compartido

El estado compartido es cualquier variable, objeto o espacio de memoria que exista en un ámbito compartido. Un ámbito compartido puede incluir el alcance global o ámbitos de cierre. A menudo, en la programación orientada a objetos, los objetos se comparten entre ámbitos al agregar propiedades a otros objetos.

```
class Estudiante {
    String nombre = "";
    int edad = 0;
    List<Materia> materias = null;
    public Estudiante(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
class Materia {
    public enum CURSO { INGLES, MATEMATICA, CASTELLANO, CIENCIAS, HISTORIA };
    private final CURSO curso;
    int duracion = 0;
    public Materia(CURSO curso, int duracion) {
        this.curso = curso;
        this.duracion = duracion;
    }
}
```

```
Estudiante estudiante = new Estudiante("Pedro", 21);
List<Materia> materias = new ArrayList<>();
Materia castellano = new Materia(Materia.CURSO.CASTELLANO, 60);
Materia ciencias = new Materia(Materia.CURSO.CIENCIAS, 60);
materias.add(castellano);
materias.add(ciencias);
estudiante.materias = materias;
```

ACADEMIA CONSULTEC

Efectos Secundarios

Los efectos secundarios se producen cuando una función altera el estado del programa fuera de su ámbito (cualquier cambio que no sea su valor de retorno). Entre los efectos secundarios se incluyen:

- Modificar una variable externa.
- Escribir, leer o modificar un fichero.
- Ejecutar otra función que tiene efectos secundarios.
- Habilitar una conexión.
- Retorna valores por defecto o nulos.

En la programación funcional se intenta minimizar la cantidad de funciones que tienen efectos secundarios. De forma que las tengamos bien localizadas y a ser posible aisladas.

```
static void helloWorld() {
    System.out.println("Hello World!");
}
```

ACADEMIA CONSULTEC

Efectos Secundarios

```
static boolean containsMexico(File file) {
    try (BufferedReader bfReader = new BufferedReader(new FileReader(file))) {
        String line;
        while ((line = bfReader.readLine()) != null) {
            if (line.contains("Mexico")) {
                return true;
            }
        }
    } catch (IOException ignored) {
        return false;
    }
    return false;
}
```

ACADEMIA CONSULTEC

Inmutabilidad

La idea fundamental detrás de la inmutabilidad es simple: si queremos cambiar una estructura de datos, necesitamos crear una nueva copia de ella con los cambios, en lugar de mutar la estructura de datos original. Por lo tanto las ventajas de la inmutabilidad son las siguientes:

- Los datos no cambiarán a nuestras espaldas.
- Una vez verificado, será válido indefinidamente.
- Sin efectos secundarios ocultos.
- No hay objetos medio inicializados, moviéndose a través de diferentes métodos hasta que esté completo. Esto desacoplará los métodos y, con suerte, los convertirá en funciones puras.
- Seguridad del hilo: no más condiciones de carrera. Si una estructura de datos nunca cambia, podemos usarla de forma segura en varios subprocesos.
- Mejor capacidad de almacenamiento en caché.
- Posibles técnicas de optimización, como transparencia referencial y memorización.

ACADEMIA CONSULTEC

Inmutabilidad

```java
public class UserMutable {
    private String email;
    public UserMutable(){
        super();
    }
    public UserMutable(String email) {
        this.email = email;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

```java
public final class UserInmutable {
    private final String email;
    public UserInmutable(String email) {
        this.email = email;
    }
    public String getEmail() {
        return email;
    }
}
```

ACADEMIA CONSULTEC

Tipos Inmutables Incorporados

Hay varios tipos inmutables disponibles en el JDK. Aquí hay algunos que probablemente ya conoces:

- Contenedores primitivos (java.lang.Integer, java.lang.Boolean, etc.).
- java.lang.String (excepto un código hash calculado de forma diferida).
- Tipos matemáticos (java.math.BigInteger, java.math.BigDecimal).
- Enumeraciones.
- java.util.Locale.
- java.util.UUID.
- API de fecha / hora de Java 8.

Palabra clave “final”:

La palabra clave final se utiliza para definir una variable que solo se puede asignar una vez. Puede parecer inmutabilidad al principio, pero en realidad no lo es en un sentido más amplio.

En lugar de crear una estructura de datos inmutable, sólo la referencia a ella será inmutable.

ACADEMIA CONSULTEC

Funciones de Orden Mayor

Llamamos a una determinada operación de orden superior o mayor si la misma recibe otra operación (comportamiento) por parámetro, siendo capaz de ejecutarla internamente, por tanto:

- Acepta una o más funciones como entrada.
- Retorna o tiene como parámetro de salida una función.

Ventajas:

- Pasar comportamientos (pasar funciones como parámetros).
- Compartir medio de comunicación (callbacks).
- Compartir lógica/reglas que se puedan repetir a lo largo del proyecto.

ACADEMIA CONSULTEC

Funciones de Orden Mayor

```
static Function<Integer, Boolean> esPar = a -> a % 2 == 0;

private static Integer TotalPares(Integer[] listaNumeros, Function<Integer, Boolean>
esPar) {
    int total = 0;
    for (Integer numero: listaNumeros ) {
        if (esPar.apply((java.lang.Integer) numero)) total++;
    }
    return total;
}

public static void main(String[] args) {
    Integer[] listaNumeros = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
    System.out.println("El total de listaNumeros pares entre 1 y 9 es: " +
TotalPares(listaNumeros, esPar));
}
```

En consola aparece:

```
El total de numeros pares entre 1 y 9 es: 4
```

ACADEMIA CONSULTEC

Funciones Lambda

Una expresión lambda, también denominada función lambda, función literal o función anónima; es una subrutina definida que no está enlazada a un identificador, es decir, un expresión Lambda es una función la cual no tiene un nombre.

Las expresiones lambda suelen utilizarse para lo siguiente:

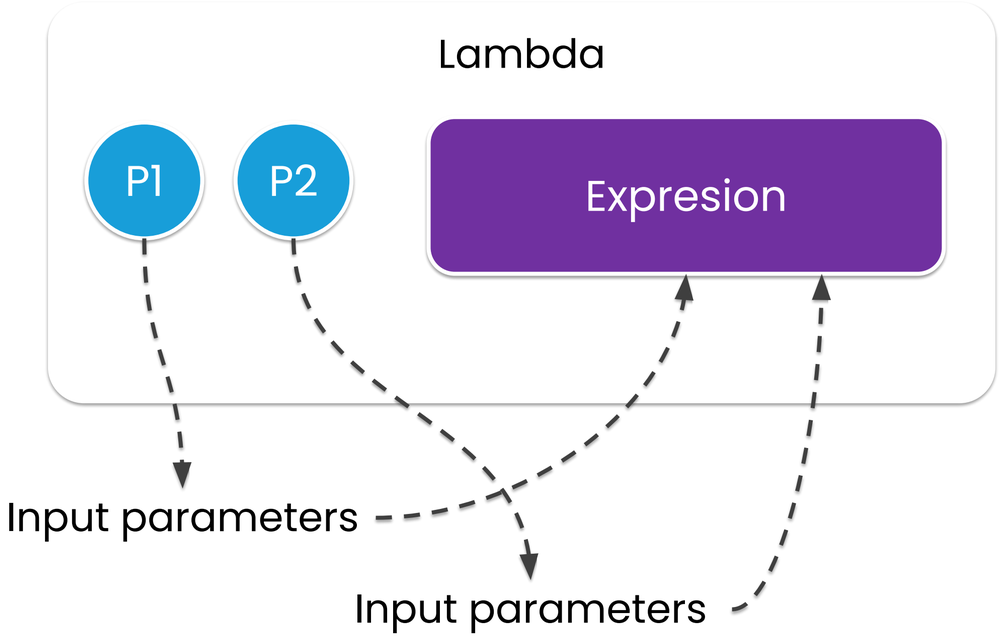
- Como argumentos que son pasados a otras funciones de orden superior.
- Para construir el resultado de una función de orden superior que necesita retornar una función.

Una de las finalidades de las expresiones Lambdas es convertir funciones en un tipo de programación imperativa, gracias a la programación funcional (que se basa en un estilo declarativo) a una programación declarativa.

ACADEMIA CONSULTEC

Expresiones Lambda

Una expresión lambda se compone de dos elementos. En primer lugar de un conjunto de parámetros y en segundo lugar de una expresión que opera con los parámetros indicados.




```
(int x, int y) ->  x+y
() -> 55
(String message) ->{ System.out.println(message); }
```

ACADEMIA CONSULTEC

Formas de las expresiones

Lambda de expresión:

Una *lambda de expresión* que tiene una expresión como cuerpo:

```
(input-parameters) => expresión
```

Una *expresión lambda* con una expresión en el lado derecho del operador se denomina lambda de expresión. Las lambdas de expresión se usan ampliamente en la construcción de *árboles de expresión*.

Lambda de instrucción:

Una lambda de instrucción que tiene un bloque de instrucciones como cuerpo:

```
(input-parameters) => { <sequence-of-statements> }
```

Como ves, las lambdas de instrucción son similares a las lambdas de expresión, salvo que las instrucciones se encierran entre llaves

Formas de las expresiones

```
interface AgregableReturnKeyword{
    int add(int a,int b);
}
public class ExpressionReturnKeyword  {
    public static void main(String[] args) {
        // Lambda expression sin la palabra clave return.
        // Lambda Expression
        AgregableReturnKeyword ad1 = (a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Lambda expression con la palabra clave return.
        // Lambda Instruction
        AgregableReturnKeyword ad2 = (int a,int b)->{
            return (a+b);
        };
        System.out.println(ad2.add(100,200));
    }
}
```

En consola aparece:

```
30
300
```

ACADEMIA CONSULTEC

Almuerzo

45 min

Programación

Funcional

ACADEMIA CONSULTEC

Lambda y forEach

Una funcionalidad clave en lambda es el uso de iteradores forEach en colecciones. El bucle forEach proporciona a los programadores una forma nueva, concisa e interesante de iterar sobre una colección.

Iteración sobre una colección:

```
List<String> names = Arrays.asList("Larry", "Steve", "James");
names.forEach(System.out::println);

Set<String> uniqueNames = new HashSet<>(names);
uniqueNames.forEach(System.out::println);

Queue<String> namesQueue = new ArrayDeque<>(names);
namesQueue.forEach(System.out::println);
```

Iteración utilizando forEach de Map:

```
Map<Integer, String> namesMap = new HashMap<>();
namesMap.put(1, "Larry");
namesMap.put(2, "Steve");
namesMap.put(3, "James");
namesMap.forEach((key, value) ->
        System.out.println(key + " " + value));

// Iterar usando entrySet
namesMap.entrySet().forEach(entry -> System.out.println(
        entry.getKey() + " " + entry.getValue()));
```

ACADEMIA CONSULTEC

Package java.util.function

El paquete *java.util.function* proporciona un conjunto de interfaces funcionales comunes reutilizables (y sus correspondientes lambda) definiciones que pueden ser utilizadas por los programadores en su código en lugar de crear interfaces funcionales completamente nuevas.

Las interfaces funcionales proporcionan tipos de destino para expresiones lambda y referencias a métodos. Cada interfaz funcional tiene un único método abstracto, denominado método funcional para esa interfaz funcional.

ACADEMIA CONSULTEC

Interface Supplier

La interfaz java.util.function.Supplier, se debe utilizar cuando se necesiten generar objetos sin requerir argumentos, por tanto, es útil para cuando no necesitamos suministrar ningún valor y obtener un resultado al mismo tiempo. Esta interfaz consta de una sola función get().

```
// Esta funcion retorna un valor aleatorio
Supplier<Double> randomValue = () -> Math.random();

// Se obtiene el valor
System.out.println(randomValue.get());
```

En consola aparece:

```
0.6884692294055309
```

ACADEMIA CONSULTEC

Interface Consumer

La interfaz java.util.function.Consumer es el opuesto de Supplier, acepta como argumento el tipo T sin devolver ningún valor de retorno, es decir, representa una función que toma un argumento y produce un resultado, sin devolver ningún valor. Esta interfaz consta de dos funciones accept() y andThen().

```
// Consumer multiplica por 2 a cada entero de la lista
Consumer<List<Integer> > modify = list -> {
    for (int i = 0; i < list.size(); i++)
            list.set(i, 2 * list.get(i));
};

// Consumer muestra la lista de enteros
Consumer< List<Integer> > dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));

List<Integer> list = new ArrayList<Integer>();
list.add(2);
list.add(1);
list.add(3);

// usando addThen()
modify.andThen(dispList).accept(list);
```

En consola aparece:

```
4 2 6
```

ACADEMIA CONSULTEC

Interface Functions

La interfaz java.util.function.Funtions reciben un argumento y produce un resultado. La expresión lambda asignada a un objeto de tipo Function se utiliza para definir su apply() que finalmente aplica la función dada sobre el argumento. La interfaz Function consta de 4 métodos: apply(), andThen(), compose() e identity().

```
Function<Integer, String> function = (num) -> {
    if ( num%3==0 || num%5==0 ){ return num+" divisible entre 3 o 5"; }
    else { return num+" no es divisible entre 3 o 5"; }
};
function = function.andThen(cadena -> cadena + "..!!");
System.out.println(function.apply(25));
System.out.println(function.apply(8));

function = function.compose(num -> num + 2);
System.out.println(function.apply(5));
System.out.println(function.apply(7));
```

En consola aparece:

```
25 divisible entre 3 o 5..!!
8 no es divisible entre 3 o 5..!!
7 no es divisible entre 3 o 5..!!
9 divisible entre 3 o 5..!!
```

ACADEMIA CONSULTEC

Interface Predicate

La interfaz java.util.function.Predicate son una especialización de los *Functions* ya que reciben un parámetro y devuelven un valor booleano, se usan de forma intensiva en operaciones de filtrado y ayuda en las pruebas unitarias por separado mejorando la manejabilidad del código. La interfaz *Predicate* consta de los métodos:

- isEqual(Object targetRef): Devuelve un predicado que comprueba si dos argumentos son iguales según Objects.equals(Object, Object).
- and(Predicate other): Devuelve un predicado compuesto que representa un AND lógico de cortocircuito (short-circuiting) de este predicado y otro.
- negate(): Devuelve un predicado que representa la negación lógica de este predicado.
- or(Predicate other): Devuelve un predicado compuesto que representa un OR lógico de cortocircuito (short-circuiting) entre este predicado y otro.
- test(T t): Evalúa este predicado sobre el argumento dado boolean test(T t).

ACADEMIA CONSULTEC

Interface Predicate

```
Predicate<Integer> mayorQue10 = (i) -> i > 10;

Predicate<Integer> menosQue20 = (i) -> i < 20;
boolean result = mayorQue10.and(menosQue20).test(15);
System.out.println("Es mayor que 10 y menor que 20? " + result);

boolean result2 = mayorQue10.and(menosQue20).negate().test(15);
System.out.println("Es mayor que 10 y menor que 20? " + result2);
```

En consola aparece:

```
Es mayor que 10 y menor que 20? true

Es mayor que 10 y menor que 20? False
```

ACADEMIA CONSULTEC

API Stream

La API Stream se utiliza para procesar colecciones de objetos. Un Stream es una secuencia de objetos que soporta varios métodos que pueden ser canalizados para producir el resultado deseado. Las características de Java Stream son:

- Un Stream no es una estructura de datos sino que toma la entrada de Colecciones, Arrays o canales de E/S.
- Los flujos no cambian la estructura de datos original, sólo proporcionan el resultado según los métodos canalizados.
- Cada operación intermedia es ejecutada perezosamente y devuelve un flujo como resultado, por lo tanto varias operaciones intermedias pueden ser canalizadas.
- Las operaciones terminales marcan el final del flujo y devuelven el resultado.

API Stream – Operaciones Intermedias

- map: El método map se utiliza para devolver un flujo formado por los resultados de aplicar la función dada a los elementos de este flujo.

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

- filter: El método filter se utiliza para seleccionar elementos según el Predicate pasado como argumento.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
```

- sorted: El método sorted se utiliza para ordenar el flujo.

```
List names = Arrays.asList("Reflection","Collection","Stream");
List result = names.stream().sorted().collect(Collectors.toList());
```

ACADEMIA CONSULTEC

API Stream – Operaciones de Terminal

- collect: El método collect se utiliza para devolver el resultado de las operaciones intermedias realizadas sobre el flujo.

```
List number = Arrays.asList(2,3,4,5,3);
Set square = number.stream().map(x->x*x).collect(Collectors.toSet());
```

- forEach: El método forEach se utiliza para iterar por cada elemento del flujo.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

- reduce : El método reduce se utiliza para reducir los elementos de un flujo a un único valor. El método reduce toma un BinaryOperator como parámetro.

```
List number = Arrays.asList(2,3,4,5);
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);
```

ACADEMIA CONSULTEC

API Stream – Operaciones de Terminal

```
// crea una lista de enteros
List<Integer> number = Arrays.asList(2,3,4,5);
// Metodo map
List<Integer> square = number.stream().map(x -> x*x).
        collect(Collectors.toList());
System.out.println(square);
// creo una lista de String
List<String> names =
        Arrays.asList("Reflection","Collection","Stream");
// metodo filter
List<String> result = names.stream().filter(s->s.startsWith("S")).
        collect(Collectors.toList());
System.out.println(result);
// Metodo sorted
List<String> show =
        names.stream().sorted().collect(Collectors.toList());
System.out.println(show);
```

```
// creo una lista de enteros
List<Integer> numbers = Arrays.asList(2,3,4,5,2);
// retorno valor como Set
Set<Integer> squareSet =
        numbers.stream().map(x->x*x).collect(Collectors.toSet());
System.out.println(squareSet);
// Metodo forEach
number.stream().map(x->x*x).forEach(y->System.out.println(y));
// metodo reduce
int even =
        number.stream().filter(x->x%2==0).reduce(0,(a,i)-> a+i);

System.out.println("Resultado: " + even);
```

ACADEMIA CONSULTEC

Reference Method

Una referencia a métodos o **métodos referenciados** proporciona una forma de referirse a un método sin ejecutarlo, Se relaciona con **expresiones lambda** porque también requiere un contexto de tipo de objetivo que consiste en una interfaz funcional compatible. El operador '::' se usa como referencia de método. Veamos algunos tipos de referencias de método:

- Referencia a un método de una instancia: object :: instanceMethod
- Referencia a un método estático: Class :: staticMethod
- Referencia a un método de instancia de un objeto arbitrario de un tipo particular: Class :: instanceMethod
- Referencia a un constructor: Class :: new

Typo de Referencia a Metodo	Method Reference	Lambda Expression
Static method	String::valueOf	(int i) -> String.valeOf(i)
Instance method of a **particular** object	s::substring	(int beg, int end) -> s.substring(beg, end)
Instance method of **arbitrary** object	String::equals	(String s1, String s2) -> s1.equals(s2)
Constructor	String::new	() -> new String ()

Reference Method

```
String str[] = {"pink", "orange", "black", "red"};
Arrays.sort(str, String::compareToIgnoreCase);

for(String str1 : str) {
    System.out.println(str1);
}
```

En consola aparece:

```
black
orange
pink
red
```

ACADEMIA CONSULTEC

Receso

15 min

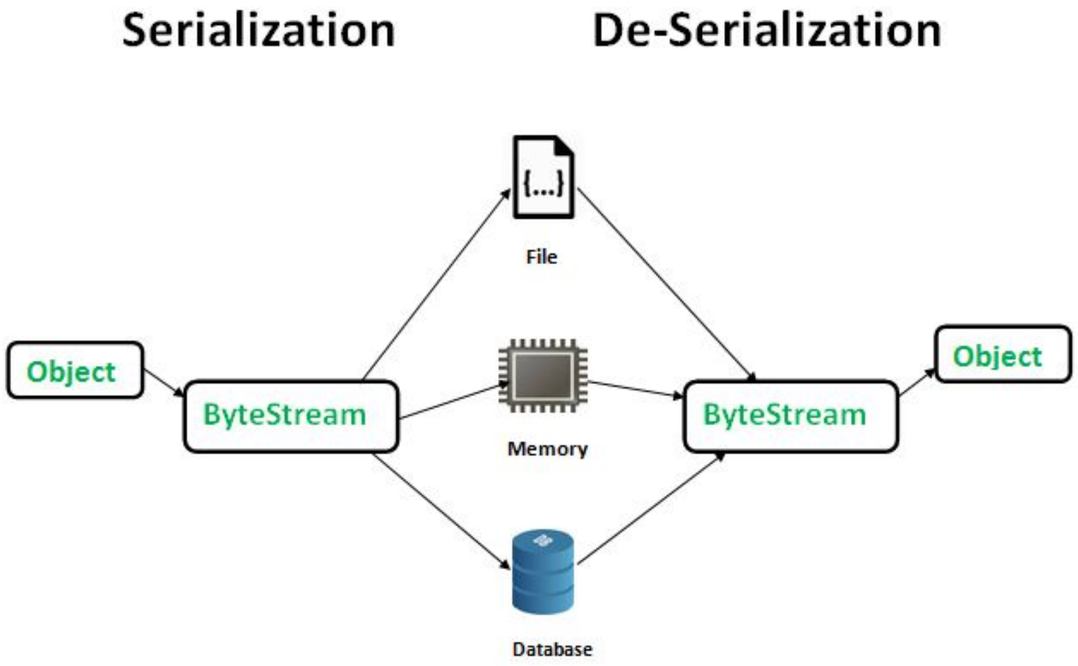
Serialización y **Anotaciones**

ACADEMIA CONSULTEC

Serialización

La **serialización** de un objeto consiste en generar una secuencia de bytes lista para su almacenamiento o transmisión. Después, mediante la **deserialización**, el estado original del objeto se puede reconstruir.

Para que un objeto sea serializable, ha de implementar la interfaz java.io.Serializable (que lo único que hace es marcar el objeto como serializable, sin que tengamos que implementar ningún método).



Serialización

```
ValueOf a = new ValueOf(20, "GeeksForGeeks");

// Serializing 'a'
FileOutputStream fos
        = new FileOutputStream("xyz.txt");
ObjectOutputStream oos
        = new ObjectOutputStream(fos);
oos.writeObject(a);

// De-serializing 'a'
FileInputStream fis
        = new FileInputStream("xyz.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
ValueOf b = (ValueOf)ois.readObject(); // down-casting object

System.out.println(b.i + " " + b.s);

// closing streams
oos.close();
ois.close();
```

```
class ValueOf implements Serializable {
    int i;
    String s;

    // ValueOf class constructor
    public ValueOf(int i, String s)
    {
        this.i = i;
        this.s = s;
    }
}
```

En consola aparece:

```
20 a la vista
```

ACADEMIA CONSULTEC

Notaciones

Las anotaciones Java son un mecanismo para añadir información de metadatos a nuestro código fuente (Programa), por tanto, las anotaciones se utilizan para proporcionar información complementaria sobre un programa. Y se especifica por lo siguiente:

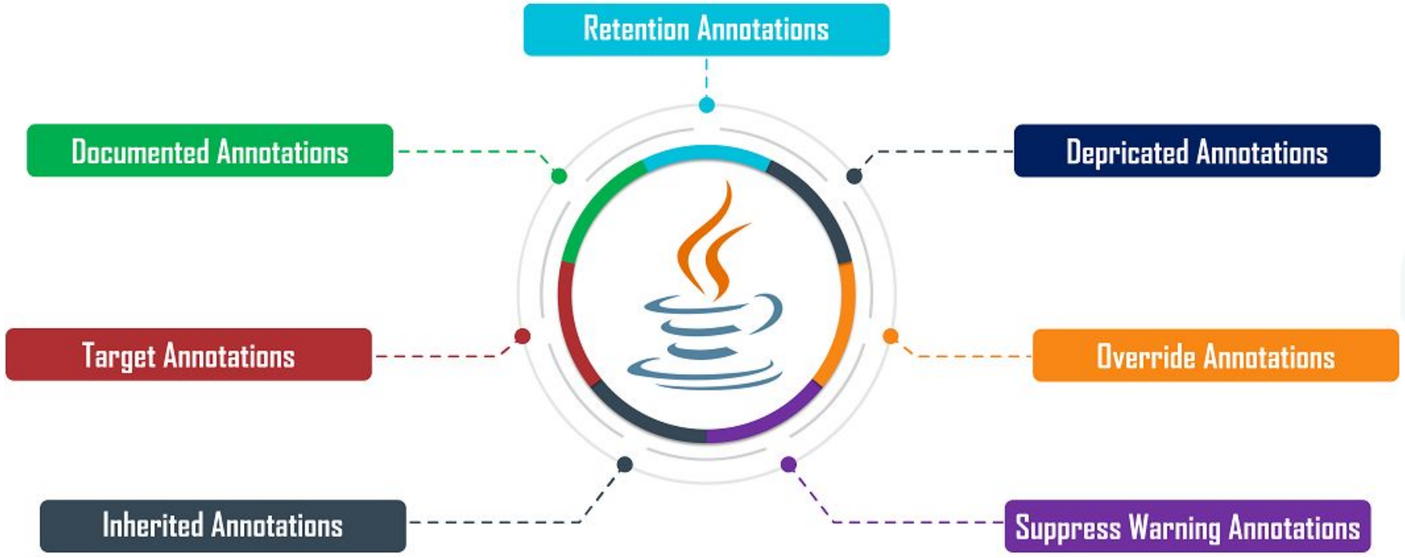
- Las anotaciones empiezan por '@'.
- Las anotaciones no cambian la acción de un programa compilado.
- Las anotaciones ayudan asociar metadatos (información) a los elementos del programa, es decir, variables de
- instancia, constructores, métodos, clases, etc.
- Las anotaciones no son comentarios puros ya que pueden cambiar la forma en que un programa es tratado por el compilador.
- Las anotaciones se utilizan básicamente para proporcionar información adicional, por lo que podrían ser una alternativa a las interfaces de marcadores XML y Java.

ACADEMIA CONSULTEC

Categorías de anotaciones

A grandes rasgos, existen 5 categorías de anotaciones:

- Anotaciones de marcador (Marker Annotations).
- Anotaciones de valor único (Single value Annotations).
- Anotaciones completas (Full Annotations).
- Anotaciones de tipo (Type Annotations).
- Anotaciones de repetición (Repeating Annotations).



Categorías de anotaciones

Anotaciones de Marcador

Se declaran a efectos de una Marca que describe su presencia. No incluyen ningún miembro en ellas lo que hace que permanezcan vacías. @Override es un ejemplo de Anotaciones de Marcador.

```
package Types;
@interface MarkerTypeAnnotation{}

@TestAnnotation()
```

Anotaciones simples

Están diseñadas para incluir un único miembro en ellas. El método abreviado se utiliza para especificar el valor del miembro declarado dentro de la anotación única.

```
package Types;
@interface SingleTypeAnnotation{
    int member() default 0;
}
@TestAnnotation(“testing”);
```

ACADEMIA CONSULTEC

Categorías de anotaciones

Anotaciones completas

Se declaran a efectos de una Marca que describe su presencia. No incluyen ningún miembro en ellas lo que hace que permanezcan vacías. @Override es un ejemplo de Anotaciones de Marcador.

Anotaciones de tipo

Estas anotaciones pueden aplicarse a cualquier lugar donde se utilice un tipo. Por ejemplo, podemos anotar el tipo de retorno de un método. Estos se declaran con la anotación @Target.

```
package Types;
@interface FullAnnotationType{
    int member1() default 1;
    String member2() default "";
    String member3() default "abc";
}
@TestAnnotation(owner=“Rahul”, value=“Class Geeks”)
```

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

// Using target annotation to annotate a type
@Target(ElementType.TYPE_USE)

// Declaring a simple type annotation
@interface TypeAnnoDemo{}
```

Custom Annotation

Una anotación se define por medio de la palabra reservada ***@interface***. Hay que tener ciertas consideraciones en cuenta a la hora de crearlas.

- Cada método dentro de la anotación es un elemento.
- Los métodos no deben tener parámetros o cláusulas throws.
- Los tipos de retorno están restringidos a tipos primitivos, String, Class, enum, anotaciones, y arrrays de los tipos anteriores.
- Los métodos pueden tener valores por defecto

ACADEMIA CONSULTEC

Custom Annotation

@Target – Especifica el tipo de elemento al que se va a asociar la anotación.

- ElementType.TYPE – se puede aplicar a cualquier elemento de la clase.
- ElementType.FIELD – se puede aplicar a un miebro de la clase.
- ElementType.METHOD – se puede aplicar a un método
- ElementType.PARAMETER – se puede aplicar a parámetros de un método.
- ElementType.CONSTRUCTOR – se puede aplicar a constructores
- ElementType.LOCAL_VARIABLE – se puede aplicar a variables locales
- ElementType.ANNOTATION_TYPE – indica que el tipo declarado en sí es un tipo de anotación.

@Retention – Especifica el nivel de retención de la anotación (cuándo se puede acceder a ella).

- RetentionPolicy.SOURCE — Retenida sólo a nivel de código; ignorada por el compilador.
- RetentionPolicy.CLASS — Retenida en tiempo de compilación, pero ignorada en tiempo de ejcución.
- RetentionPolicy.RUNTIME — Retenida en tiempo de ejecución, sólo se puede acceder a ella en este tiempo.

@Documented – Hará que la anotación se mencione en el javadoc.

@Inherited – Indica que la anotación será heredada automáticamente.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MultipleInvocation {

    int timesToInvoke() default 1;
}
```

ACADEMIA CONSULTEC

Gracias

¡Nos vemos pronto!