





Java 8

Hola!



Soy José Julián Ariza

Fundador de **Impacto Tecnológico** desde 2006.

Líder estratégico de Equipos de Desarrollo Ágil en modalidad Online y Presencial

Arquitecto de Software por naturaleza y DevOps por hobby

@JJArizaV – josejulian@impactotecnologico.net

@impactotecno



1. Colecciones, streams y filters



Colecciones

- Una Colección (“Collection”) es un contenedor de múltiples elementos agrupados en una sola unidad.
- El framework Collections de Java es una arquitectura que nos permite representar y manipular colecciones en Java de una forma estándar.
- El framework consta de al menos tres partes:
 - ✓ Interfaces
 - ✓ Clases
 - ✓ Algoritmos

Colecciones - Interfaces

- Las interfaces proveen los tipos de datos abstractos para representar una colección.
- La interfaz superior en la jerarquía de estas interfaces es la llamada “java.util.Collection”. Esta contiene los métodos que se deberán implementar en caso de emplearlas: size(), iterator(), add(), remove() y clear().
- Otras interfaces importantes son:
 - ✓ java.util.List
 - ✓ java.util.Set
 - ✓ java.util.Queue
 - ✓ java.util.Map

Colecciones - Interfaces

- List - Ejemplo

```
List aList = new ArrayList();
        aList.add("test");
        aList.add(123);
        aList.add(null);
        aList.add(55.22);
        System.out.println("Size of the list " + aList.size());
```

Colecciones - Interfaces

- Set - Ejemplo

```
Set hSet = new HashSet();
        hSet.add("test");
        hSet.add(123);
        hSet.add(null);
        hSet.add(55.22);
        System.out.println("Size of the list " + hSet.size());
```

Colecciones - Interfaces

- Queue - Ejemplo

```
Queue pQueue = new PriorityQueue();
    pQueue.add(123);
    pQueue.add(46);
    pQueue.add(78);
    pQueue.add(99);
    System.out.println("Size of the Queue " + pQueue.size());
    System.out.println("Queue Elements" + pQueue);
    System.out.println("First Element in the
Queue"+pQueue.poll());
    System.out.println("Size of the Queue " + pQueue.size());
    System.out.println("First Element in the
Queue"+pQueue.poll());
    System.out.println("Size of the Queue " + pQueue.size());
```

Colecciones - Interfaces

En el caso anterior, estas colas de prioridad se usan frecuentemente para almacenar datos según una prioridad y así mantener el orden de los mismos durante una operación.

Además, al tratarse de una cola, provee métodos para operar sobre su primer y último elemento.

Colecciones - Interfaces

El primer elemento de la queue por prioridades es el último elemento basado en el orden natural de inserción o en el orden según indique el comparador, de haber múltiples objetos con el mismo orden, se ordenan de forma aleatoria.

El tamaño de la queue no tiene límites, sin embargo, podemos especificar su capacidad al momento de su creación, a medida que vayamos agregando elementos a la cola, su capacidad incrementará automáticamente.

Colecciones - Interfaces

Otra cosa a resaltar es que estas colas no son seguras para su uso en hilos, así que Java nos provee de la clase PriorityBlockingQueue que implementa la interfaz BlockingQueue para usar colas en un ambiente multi-hilos.

Colecciones - Interfaces

- Map - Ejemplo

```
Map hMap = new HashMap();
        hMap.put(123,"name1");
        hMap.put("key",12.34);
        hMap.put(null, "123");
        System.out.println("Map Elements ::" + hMap);
        System.out.println("Value is    ::" + hMap.get(null));
        System.out.println("Keys are   ::"+ hMap.keySet());
        System.out.println("Values are ::" + hMap.values());
```

Colecciones - Clases

- Nos permite crear diferentes tipos de clases dentro de un programa java.
- Estas clases nos ayudan a resolver la mayor parte de nuestras necesidades, pero si aún así necesitamos algo diferente, siempre podemos extender las clases para ajustarlas a nuestra necesidad.
- Las clases disponibles:
 - ✓ ArrayList
 - ✓ LinkedList
 - ✓ HashMap
 - ✓ TreeMap
 - ✓ HashSet
 - ✓ TreeSet

Colecciones - Clases

ArrayList - Ejemplo

```
// Declaración de un ArrayList de "String". Puede ser de cualquier otro Elemento u Objeto (float, Boolean, Object, ...)  
ArrayList<String> nombreArrayList = new ArrayList<String>();  
// Añade el elemento al ArrayList  
nombreArrayList.add("Elemento");  
// Añade el elemento al ArrayList en la posición 'n'  
nombreArrayList.add(n, "Elemento 2");  
// Devuelve el numero de elementos del ArrayList  
nombreArrayList.size();  
// Devuelve el elemento que esta en la posición '2' del ArrayList  
nombreArrayList.get(2);  
// Comprueba se existe del elemento ('Elemento') que se le pasa como parametro  
nombreArrayList.contains("Elemento");  
// Devuelve la posición de la primera ocurrencia ('Elemento') en el ArrayList  
nombreArrayList.indexOf("Elemento");  
// Devuelve la posición de la última ocurrencia ('Elemento') en el ArrayList  
nombreArrayList.lastIndexOf("Elemento");
```

Colecciones - Clases

ArrayList - Ejemplo

```
// Borra el elemento de la posición '5' del ArrayList  
nombreArrayList.remove(5);  
  
// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.  
nombreArrayList.remove("Elemento");  
  
//Borra todos los elementos de ArrayList  
nombreArrayList.clear();  
  
// Devuelve True si el ArrayList esta vacio. Sino Devuelve False  
nombreArrayList.isEmpty();  
  
// Copiar un ArrayList  
ArrayList arrayListCopia = (ArrayList) nombreArrayList.clone();  
  
// Pasa el ArrayList a un Array  
Object[] array = nombreArrayList.toArray();
```

Colecciones - Clases

LinkedList - Ejemplo

Son listas cuyos elementos no se encuentran almacenados de forma secuencial, sino que son representados internamente como nodos.

```
/* Declaración */  
LinkedList<String> linkedlist = new LinkedList<String>();  
  
/* Añadiendo elementos */  
linkedlist.add("Item1");  
linkedlist.add("Item5");  
linkedlist.add("Item3");  
linkedlist.add("Item6");  
linkedlist.add("Item2");  
  
/*Mostrando elementos*/  
System.out.println("Linked List Content: " +linkedlist);
```

Colecciones - LinkedList

- ◆ LinkedList es más rápido en agregar y quitar, pero más lento en obtener.
- ◆ LinkedList es preferido si:
 - ◆ No hay gran cantidad de acceso aleatorio de elementos.
 - ◆ Hay un gran número de operaciones de agregar / quitar

ArrayList vs LinkedList

ArrayList

Ventajas	Desventajas
Añadir elementos de forma directa	Costos adicionales al añadir o remover elementos
Acceso a elementos de forma directa	La cantidad de memoria considera la capacidad definida para el ArrayList, aunque no contenga elementos

ArrayList vs LinkedList

LinkedList

Ventajas	Desventajas
Añadir y remover elementos con un iterador	Uso de memoria adicional por las referencias a los elementos anterior y siguiente
Añadir y remover elementos al final de la lista	El acceso a los elementos depende del tamaño de la lista

ArrayList vs LinkedList

¿Cuál usar?

En la práctica la mayoría de las ocasiones se recomienda usar ArrayList debido a que el tiempo de las operaciones y uso de memoria es menor que en LinkedList.

Eso no quiere decir que LinkedList nunca se utilice, existen algunos casos muy específicos donde es la mejor opción, por ejemplo la pila de llamadas del lenguaje C está implementada usando una estructura de datos con estas características.

Colecciones - Clases

HashMap - Ejemplo

Nos permite almacenar datos en pares de clave - valor.

```
Map<String, Integer> map = new HashMap<>();
```

```
System.out.print(map);
map.put("vishal", 10);
map.put("sachin", 30);
map.put("vaibhav", 20);
```

```
System.out.println("Size of map is:- " + map.size());
```

Colecciones - Clases

TreeMap- Ejemplo

Nos facilita el sorting interno de un listado de datos mapeados

```
Map<Integer, String> treemap = new TreeMap<>();  
    treemap.put(3, "TreeMap");  
    treemap.put(2, "vs");  
    treemap.put(1, "HashMap");  
  
    System.out.println(treemap);
```

Colecciones - Clases

HashSet - Ejemplo

```
HashSet<String> hset =  
    new HashSet<String>();  
    hset.add("Apple");  
    hset.add("Mango");  
    hset.add("Grapes");  
    hset.add("Orange");  
    hset.add("Fig");  
  
    hset.add("Apple");  
    hset.add("Mango");  
  
    hset.add(null);  
    hset.add(null);  
  
    System.out.println(hset);  
}
```

Colecciones - Clases

TreeSet - Ejemplo

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("First");
treeSet.add("Second");
treeSet.add("Third");
Iterator<String> itr = treeSet.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

HashSet vs TreeSet

- ◆ HashSet se recomienda para operaciones como buscar, insertar y borrar.
- ◆ HashSet es más rápido que TreeSet. HashSet se implementa utilizando una tabla hash.
- ◆ TreeSet mantiene los datos ordenados. Además, admite operaciones como high() (Devuelve el elemento más alto), floor (), ceiling (), etc. Estas operaciones también son algo más lentas en TreeSet pero no se admiten en HashSet.
- ◆ TreeSet se implementa utilizando un árbol de búsqueda binaria de equilibrio automático (árbol rojo-negro).
- ◆ TreeSet está respaldado por TreeMap

HashSet vs TreeSet

- ◆ HashSet permite objetos nulos. TreeSet no permite null y lanza NullPointerException, porque TreeSet usa el método compareTo() para comparar claves y compareTo() lanzará java.lang.NullPointerException al comparar algo nulo

Colecciones - Algoritmos

- Los algoritmos son métodos para proveer alguna funcionalidad común.
- La clase Collections contiene estos métodos.
- La mayoría de estos algoritmos trabaja sobre Listas (List) pero algunos de ellos son aplicables a todas las clases de colecciones.
- Los algoritmos disponibles son:
 - ✓ Sorting
 - ✓ Shuffling
 - ✓ Searching
 - ✓ Min and Max values

Colecciones - Algoritmos



Sorting - Ejemplo

```
int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};
```

```
Arrays.sort(arr);
```

```
System.out.printf("Modified arr[] : %s",
    Arrays.toString(arr));
```

Colecciones - Algoritmos



Shuffling - Ejemplo

```
public static void main(String[] args)
{
    ArrayList<String> mylist = new ArrayList<String>();
    mylist.add("code");
    mylist.add("quiz");
    mylist.add("geeksforgeeks");
    mylist.add("quiz");
    mylist.add("practice");
    mylist.add("qa");

    System.out.println("Original List : \n" + mylist);

    Collections.shuffle(mylist);

    System.out.println("\nShuffled List : \n" + mylist);
}
```

Colecciones - Algoritmos

Searching- Ejemplo (partamos de una clase de empleados desde la que se creará una lista de objetos) 

```
public class Employee implements Comparable<Employee> {
    int salary;
    String name;

    public Employee(String name) {
        this.name = name;
    }

    public Employee(int salary) {
        this.salary = salary;
    }

    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

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

    public int compareTo(Employee another) {
        return this.name.compareTo(another.name);
    }
}
```

Colecciones - Algoritmos

```
List<Employee> listEmployees = new ArrayList<>();  
  
listEmployees.add(new Employee("Tom", 40000));  
listEmployees.add(new Employee("Adam", 60000));  
listEmployees.add(new Employee("Jim", 70000));  
listEmployees.add(new Employee("Dane", 35000));  
listEmployees.add(new Employee("Jack", 56000));  
listEmployees.add(new Employee("Carol", 67000));  
  
Employee jim = new Employee("Jim");  
  
Collections.sort(listEmployees);  
  
int index = Collections.binarySearch(listEmployees, jim);  
  
if (index >= 0) {  
    jim = listEmployees.get(index);  
    System.out.println("Found employee: " + jim);  
}
```



Colecciones - Algoritmos



Min and max values - Ejemplo

```
List<Integer> list = new ArrayList<Integer>();  
  
list.add(12);  
list.add(53);  
list.add(30);  
list.add(8);  
  
System.out.println("List: " + list);  
  
int minList = Collections.min(list);  
System.out.println("min: " + minList);  
int maxList = Collections.max(list);  
System.out.println("max: " + maxList);
```

Stream

- Dentro del esquema de mejorar la programación funcional, Java 8 ha introducido el concepto de Stream como forma de ejecución de código en lote sobre colecciones de datos.
- La funcionalidad se encuentra en el nuevo paquete `java.util.stream`, este permite operaciones del tipo `filter/map/reduce` (operaciones de agregación sobre volúmenes de datos, en este caso colecciones) en Java 8.
- El API Stream nos permite declarar operaciones que se ejecutan tanto de forma secuencial como paralela sobre los grupos de datos.

Stream - Características

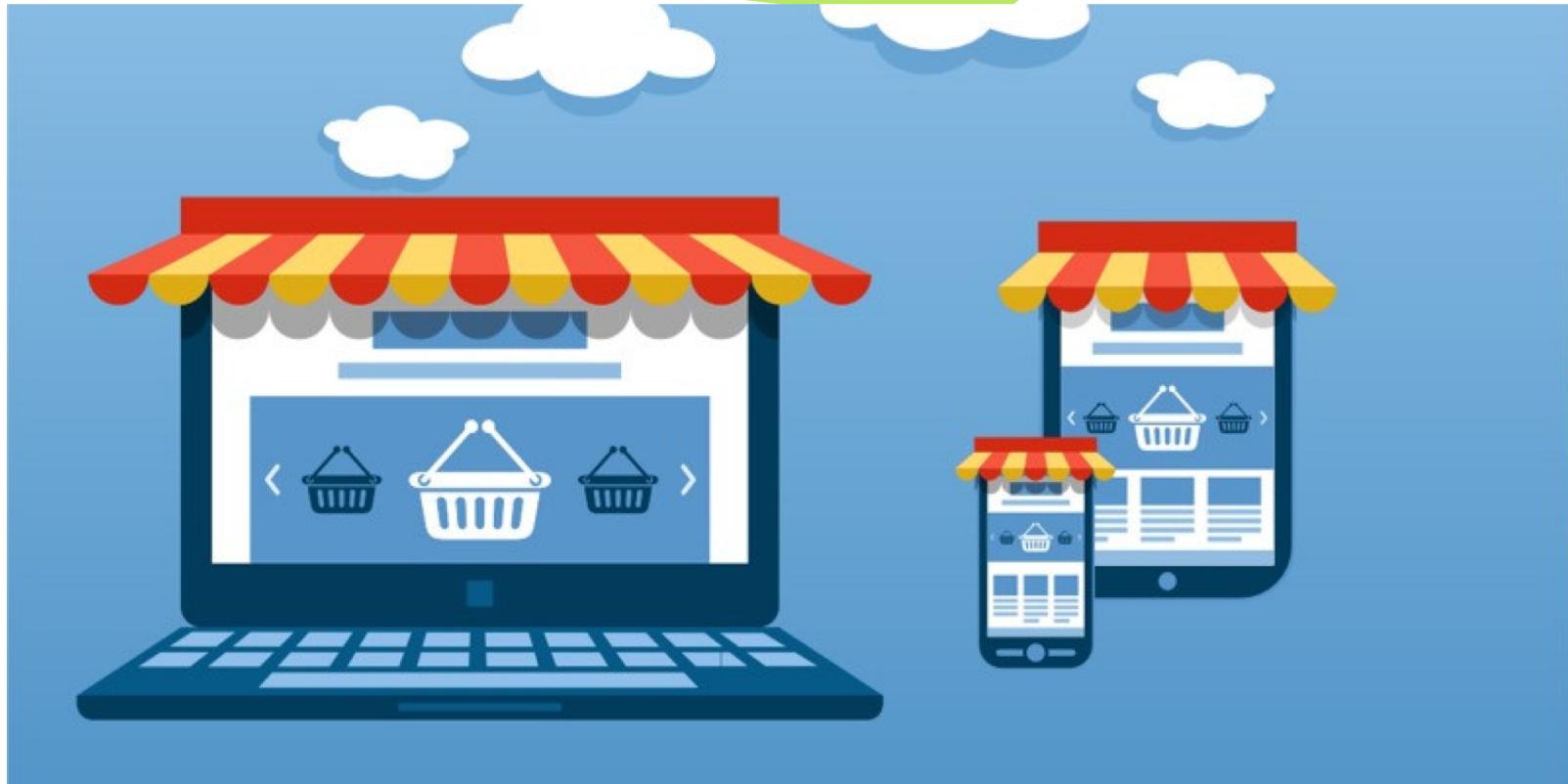
- **Secuencia de elementos:** Un stream provee un conjunto de elementos de un tipo específico de una forma secuencial. Un stream obtiene/calcula elementos a demanda. **Nunca almacena los elementos.**
- **Origen:** Un stream obtiene los valores de entrada desde Collections, Arrays y operaciones de I/O.
- **Operaciones de Agregación:** Soporta operaciones como filter, map, limit, reduce, find, match, etc.
- **Canalización:** La mayoría de las operaciones con streams retornan ellas mismos streams por lo que pueden ser colocados en un proceso continuo. Estas operaciones son llamadas “operaciones intermedias” y su función es tomar una entrada de datos, procesarla y devolver una salida a la función que la solicita.

Stream - Características

Además, las operaciones de los streams tienen dos características fundamentales que las distinguen de las operaciones con colecciones:

- ◆ **Estructura de proceso:** Muchas operaciones de stream devuelven otro stream. Así, es posible encadenar operaciones para formar un proceso más abarcador. Esto, a su vez, permite lograr ciertas optimizaciones, por ejemplo mediante las nociones de "pereza" (laziness) y "corte de circuitos" (short-circuiting). Veamos un caso...

Laziness



Laziness



My Account
Mr. User

Products 49

Images 12

Templates 3

Catalogs 5

All catalogs

Layouts

Import layouts

Showroom 2

Add Products > Design Templates > **Build Catalog** > Publish

My Company Line Sheet

floral strappy-back hi-lo dress cream 121883954375 26	solid textured dress popstar pink 121916837772 19.99	geo print hi-lo knit dress cream 141855952625 19.99	faux gem leopard dress black 141837416276 11.89
faux gem tank dress shirt 121873452646 29.99	striped hi-lo knit dress cream 141855942208 19.99	hi-lo floral dress toile blue 141900050380 26	crinkle dip-dye maxi dress turquoise glint 121906398116 34

A TEXT

PRODUCT

BACKGROUND

IMAGE

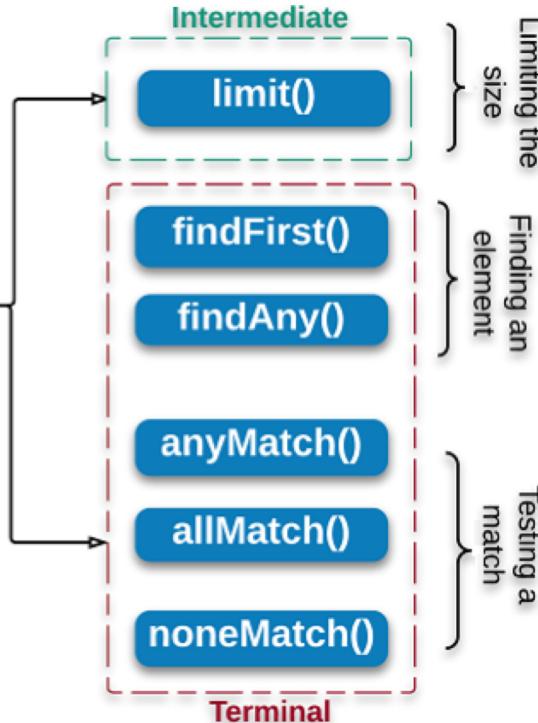
ICON

FILE

Short circuiting operations

- ◆ En la lógica booleana de cortocircuito, por ejemplo `firstBoolean && secondBoolean`, si `firstBoolean` es falso entonces la parte restante de la expresión es ignorada (la operación es cortocircuitada) porque la evaluación restante será redundante. De forma similar en `firstBoolean || secondBoolean`, si `firstBoolean` es verdadero la parte restante está en cortocircuito.
- ◆ Las operaciones de cortocircuito de Java 8 Stream no se limitan a los tipos booleanos. Hay operaciones de cortocircuito preestablecidas.

Short-circuiting Operations



Streams vs Colecciones

Tanto la noción de colecciones que ya existía en Java como la nueva noción de streams se refieren a interfaces con secuencias de elementos. Entonces, ¿cuál es la diferencia? En resumen, **las colecciones hacen referencia a datos mientras que los streams hacen referencia a cómputos.**

Streams vs Colecciones

Pensemos por ejemplo en una película almacenada en un DVD. Se trata de una colección (de bytes o de fotogramas) porque contiene toda la estructura de datos.

Ahora imaginemos el mismo video, pero esta vez lo reproducimos desde Internet. En este caso hablamos de un stream (de bytes o fotogramas).

El reproductor de vídeo por secuencias (streaming) necesita descargar solo unos pocos fotogramas más allá de los que está viendo el usuario; así, es posible comenzar a mostrar los valores del comienzo del stream antes de que la mayor parte del stream se haya computado.

Streams vs Colecciones

Ejemplo - Colección:

```
List<String> transactionIds = new ArrayList<>();  
for(Transaction t: transactions){  
transactionIds.add(t.getId());
```

Aquí generamos la iteración explícita del listado de transacciones secuencialmente para extraer la identificación de cada transacción y agregarla al acumulador. En cambio, cuando se usa un stream, no existe iteración explícita.

Streams vs Colecciones

Ejemplo - Stream:

```
List<Integer> transactionIds =  
transactions.stream()  
.map(Transaction::getId)  
.collect(toList());
```

En este caso, se crea una consulta en la que se ha parametrizado la operación map para extraer las identificaciones de transacciones, y la operación collect convierte el Stream resultante en un listado List.

1.1

Stream builder



Crear un stream de datos a través del builder:

El Stream Builder retorna un “constructor” para generar un Stream, puede usarse de la siguiente forma:

```
// Se instancia el builder para un Stream de strings Stream builder()
    Stream.Builder<String> builder = Stream.builder();

// Añadiendo elementos al Stream
    Stream<String> stream = builder.add("Geeks").build();

// Mostrando elementos del Stream
    stream.forEach(System.out::println);
```

Stream

En el ejemplo anterior, usamos el `forEach` para iterar sobre un `Stream`, este `forEach`, a diferencia del tradicional, ejecuta el `fetch` de modo arbitrario (sin orden particular sobre los elementos).

Otra diferencia relevante es que el `ForEach` tradicional a través de un `Iterable` “bloquea” la colección mientras se recorre, por lo que no permite modificaciones mientras ocurre la iteración, por el contrario, el `Stream.ForEach`, si lo permite.

Además, el uso de streams nos permite aplicar paralelismo en el uso de `forEach` a través de ellos con `parallelStreams`.

Ejemplo:

```
List<String> list = Arrays.asList("a", "b", "c", "d", "e", "f");

list.forEach(s -> System.out.println(s));

for(String s: list){
    System.out.println(s);
}

list.parallelStream().forEach(s -> System.out.println(s));
```

1.2

Iterando una colección



Stream

Usualmente, para iterar sobre una colección usaríamos un bucle como, por ejemplo, un `ForEach`, esto puede mejorarse a través del esquema funcional y la sintaxis Lambda.

Partamos de que tenemos una clase “Persona” con los siguientes atributos:

Stream

13

```
public class Persona {  
  
    private String nombre;  
    private String apellidos;  
    private int edad;  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    public int getEdad() {  
        return edad;  
    }  
    public void setEdad(int edad) {  
        this.edad = edad;  
    }  
    public String getApellidos() {  
        return apellidos;  
    }  
    public void setApellidos(String apellidos) {  
        this.apellidos = apellidos;  
    }  
    public Persona(String nombre, String apellidos, int edad) {  
        super();  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
    }  
}
```

Stream

Si creamos una lista de personas con el siguiente bloque de código:

```
Persona p1 = new Persona("juan", "sanchez", 20);
Persona p2 = new Persona("ana", "gomez", 12);
Persona p3 = new Persona("pedro", "gutierrez", 40);
List<Persona> lista=Arrays.asList(p1,p2,p3);
```

Stream

Podemos iterar sobre esta lista a través de algo como esto:

```
lista.stream().forEach((p)-> {
    System.out.println(p.getNombre());
    System.out.println(p.getApellidos());
    System.out.println(p.getEdad());
});
```

1.3

Interfaz Stream



Stream

La interfaz Stream es genérica, por lo tanto para recuperar un stream de datos podemos hacer esto:

```
Stream<Product> productStream = products.stream();
//flujo de elementos Product
```

```
productStream.forEach(product -> System.out.println(product));
// imprime la lista de productos
```

```
productStream.forEach(System.out::println);
//esta línea es equivalente a la anterior
```

Además, podemos usar la interfaz para operar sobre una lista a través de un stream, por ejemplo:

```
List<String> myList =  
    Arrays.asList("a1", "a2", "b1", "c2", "c1");
```

```
myList  
    .stream()  
    .filter(s -> s.startsWith("c"))  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(System.out::println);
```

Stream

Esto muestra que no necesitamos hacer uso exclusivo del Stream Builder para usar o implementar Streams en nuestras colecciones, sino que también podemos usar la interfaz Stream donde mejor nos parezca para operar sobre ellas de forma más cómoda y funcional.

Stream

La interfaz de Streams nos da acceso a una serie de métodos bastante útiles para operar sobre estas secuencias de datos, por ejemplo, podemos mencionar:

- ◆ map
- ◆ filter
- ◆ sorted
- ◆ distinct
- ◆ count
- ◆ skip
- ◆ limit

Veamos el uso de algunos de ellos

1.4

Filtrado de colecciones con expresiones lambda



- **Filter:** Filtrar un stream de datos es la primera operación natural que podemos necesitar. Mediante ella podemos implementar una expresión lambda que nos permita filtrar la información de una colección:

```
Stream<Persona> personsOver18 = lista.stream().filter(p -> p.getEdad() > 18);
```

- **Map:** Una vez filtrado los registros podemos querer transformar los objetos que recibimos en algo nuevo: personas con el doble de edad

```
Stream<Persona> students = lista.stream().filter(p -> p.getEdad() > 18)  
.map(person -> new Persona(person.getNombre(), person.getApellidos(), person.getEdad() * 2));
```

- **Collect:** posteriormente podemos desear convertir el resultado de los datos en una nueva colección. El API Stream provee una serie de métodos “finales” que permiten recopilar la información. collect() es uno de estos:

```
List collected = lista.stream().filter(p -> p.getEdad() > 18).map(person -> new  
    Persona(person.getNombre(), person.getApellidos(), person.getEdad() *  
    3)).collect(Collectors.toCollection(ArrayList::new));
```

- Finalmente un caso interesante es que empleando Streams podemos seleccionar qué operaciones deben ejecutarse en paralelo y cuales de forma secuencial a disposición. No es necesario que sean una u otra de principio a fin. Veamos el siguiente ejemplo:

```
List parallel = lista.stream().parallel().filter(p -> p.getEdad() > 18).sequential().map(person -> new  
    Persona(person.getNombre(), person.getApellidos(),  
    person.getEdad())).collect(Collectors.toCollection(ArrayList::new));
```

Filtrado

Si nos fijamos en el ejemplo anterior con más detalles, podemos resaltar el uso de “.parallel()”, con esto, indicamos que la próxima operación del Stream se ejecute de forma paralela o concurrente.

```
List parallel = lista.stream()  
    .parallel().filter(p -> p.getEdad() > 18)
```

Filtrado

Luego de hacer el filtrado de forma paralela, podemos volver a retomar una estrategia secuencial para el mapeo de los datos y la recolección en un nuevo ArrayList

```
sequential().map(person -> new Persona(person.getNombre(), person.getApellidos(),
person.getEdad())).collect(Collectors.toCollection(ArrayList::new));
```

Veamos un ejemplo más sencillo... a veces imprimirá 2... u otro valor

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).parallel();
stream = stream.filter(i -> i % 2 == 0);
OptionalInt opt = stream.findAny();
if (opt.isPresent()) {
    System.out.println(opt.getAsInt());
}
```

Filtrado

filtrado

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream().filter(string ->
!string.isEmpty()).collect(Collectors.toList());
System.out.println(filtered);
```

Filtrado

limit y foreach

```
Random random = new Random();
random.ints().limit(4).forEach(System.out::println);
```

map y distinct

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
List<Integer> squaresList = numbers.stream().map( i ->
i*i).distinct().collect(Collectors.toList());
System.out.println(squaresList);
```

Filtrado

Sort

```
Random random2 = new Random();
random2.ints().limit(4).sorted().forEach(System.out::println);
```

Filtrado

Collectors

```
List<String> strings2 = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered2 = strings2.stream().filter(string ->
!string.isEmpty()).collect(Collectors.toList());

System.out.println("Filtered List: " + filtered2);
String mergedString = strings2.stream().filter(string ->
!string.isEmpty()).collect(Collectors.joining(","));
System.out.println("Merged String: " + mergedString);
```

En este caso, el método joining nos ayuda a concatenar cada elemento de la lista en una sola línea que pasa a imprimirse.

Filtrado

Statistics

```
List<Integer> integers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

IntSummaryStatistics stats = integers.stream().mapToInt((x) ->
x).summaryStatistics();

System.out.println("Highest number in List : " + stats.getMax());
System.out.println("Lowest number in List : " + stats.getMin());
System.out.println("Sum of all numbers : " + stats.getSum());
System.out.println("Average of all numbers : " + stats.getAverage());
```

Como vemos, los streams numéricos soportan el método de `summaryStatistics` que nos permite acceder a distintas operaciones estadísticas y mostrar los resultados de forma sencilla.

Práctica Individual

- ◆ Crear un carrito de compra. Solo se necesita que tenga una colección de precios
- ◆ Crear funciones para:
 - ◆ calcularPrecioTotal: calcula el precio total de todos los productos del carrito
 - ◆ calcularDescuentoTotal: comprueba si algún precio es mayor o igual que el que pasamos por parámetro y, por cada coincidencia, descuenta un 5% y retorna el total que se descontará

Práctica Individual

- ◆ Parte 2:
 - ◆ Convertir el carrito para que tenga una lista de Productos (id, nombre, precio)
 - ◆ Realizar las mismas operaciones del ejercicio anterior
 - ◆ Añadir un método:
 - ◆ compraProducto: imprime un boolean si el carrito tiene un producto con el nombre recibido por parámetro. No es necesaria coincidencia exacta del nombre
 - ◆ hayBaratos: imprime un boolean true si existe algún producto con precio inferior al recibido por parámetros

1.5

Referencias a métodos



Definición

- Es una nueva característica de Java 8.
- Es un subset de las expresiones lambda, porque si en un caso una expresión lambda puede usarse, casi siempre podría emplearse el mecanismo de “referencia a método”.
- Solo puede emplearse para llamar a un método.
- Un ejemplo de la sintaxis:

`Person::getName`

Tipos

- Referencia a un método estático
- Referencia a un método de una instancia de un objeto en particular
- Referencia a un método de una instancia de un objeto arbitrario de un tipo en particular
- Referencia a un constructor

Referencia a métodos

16

Ejemplo práctico

Referencias a Métodos y Lambdas

- ◆ Los lambdas se convierten por decirlo de alguna manera en la unidad mínima de código.
- ◆ Para que Java pueda integrar estos conceptos hay que asumir que hay que realizar concesiones ya que Java solo permite el manejo de clases e interfaces.
- ◆ Así pues las expresiones lambda vienen a ser interfaces que únicamente soportan un método abstracto.
- ◆ Existen muchos de estos interfaces predefinidos. Veamos el interface **Consumer<T>** que tiene un único método que se denomina `accept` el cual recibe un parámetro y no devuelve nada. Es uno de los interfaces funcionales más sencillos.

```
public class Principal {  
    public static void main(String[] args) {  
        Consumer<String> consumidor = (x) -> System.out.println(x);  
        consumidor.accept("holo");  
    }  
}
```

Ahora vamos a crear un método reutilizable que reciba el objeto de interfaz lambda...

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        Consumer<String> consumidor = (x) -> System.out.println(x);  
        procesar(consumidor, "Hola Mundo");  
    }  
  
    public static <T> void procesar(Consumer<T> expresion, T mensaje) {  
  
        expresion.accept(mensaje);  
    }  
}
```

```
public class PrincipalFull {  
    public static void main(String[] args) {  
        Consumer<String> consumidor = (x) -> System.out.println(x);  
        procesar(consumidor, "Hola Mundo");  
        procesar((x) -> System.out.println(x), "hola2");  
        procesar(PrincipalFull::imprimir, "hola3");  
    }  
  
    public static <T> void procesar(Consumer<T> expresion, T mensaje) {  
        expresion.accept(mensaje);  
    }  
  
    public static void imprimir(String mensaje) {  
        System.out.println("-----");  
        System.out.println(mensaje);  
        System.out.println("-----");  
    }  
}
```

1.6

Encadenar métodos



Optional

En Java 8 el patrón de programación funcional se encapsula, entre otras clases, en la clase Optional, la cual incluye muchos de los métodos necesarios para trabajar con este patrón.

```
public final class Optional<T>{}
```

En la firma de la clase podemos ver que es una clase genérica que nos permite que el objeto que contenga (o no) sea de cualquier clase.

Encadenar métodos

Otra de las características de los Streams es que pueden encadenarse métodos existentes para procesar una colección de datos, para ello tenemos métodos intermedios (encargados de dar formato y manipular los datos) y métodos finales (encargados de darnos un resultado de datos salientes)

Veamos en qué categoría entraría cada método.

Encadenar métodos

Operación Intermedia	Descripción
filter	Flujo que contiene solamente los elementos que satisfacen una condición.
distinct	Flujo que contiene solamente el elemento único.
limit	Flujo con la cantidad de elementos especificados al inicio del flujo original.
map	Flujo en el cual cada elemento del flujo original se mapea a un nuevo valor, incluso con la posibilidad de que este sea de otro tipo de dato, por ejemplo se pudiera mapear los valores de cada elemento a su raíz cuadrada dando como resultados algunos valores fraccionarios.
sorted	Flujo donde sus elementos están organizados.

Encadenar métodos

Operación Final	Descripción
forEach	Procesa cada elemento del flujo.

Encadenar métodos

Operaciones de Reducción	
average	Calcula el promedio de los elementos en un flujo numérico.
count	Retorna el número de elementos en el flujo.
max	Localiza el elemento cuyo valor sea el mayor de todo el flujo numérico.
min	Localiza el elemento cuyo valor sea el menor de todo el flujo numérico.
reduce	Reduce los elementos de una colección a un solo valor usando una función de asociación acumulativa. Un ejemplo sería una expresión Lambda que añade dos elementos.

Encadenar métodos

Operaciones de Reducción Mutable	
collect	Crea una nueva colección de elementos que contienen los resultados de la operación anterior.
toArray	Crea un arreglo que contiene los resultados de la operación anterior.
Operaciones de Búsqueda	
findFirst	Encuentra el primer elemento del flujo basado en operación intermedia anterior, terminando inmediatamente el procesamiento del flujo de datos cuando es encontrada la primera coincidencia.
findAny	Encuentra cualquier elemento del flujo basado en operación intermedia anterior, terminando inmediatamente el procesamiento del flujo de datos cuando es encontrada la primera coincidencia.

Encadenar métodos - Práctica

17

1. Vamos a crear una clase Alumno con: id, dni, nombre, apellidos, nombreDelCurso, calificacion, edad
2. Añadimos un constructor que reciba todos los parámetros
3. Generamos los getters y setters
4. Sobreescribimos el `toString` para que muestre todas las propiedades
5. Creamos 1 método para cargar alumnos en un Array
6. Desarrollamos Funcionalidad para:
 - a. obtenerTodos
 - b. filtrarNombre: los que empiecen por L o G
 - c. totalAlumnos
 - d. alumnosAprobados: notas > 6
 - e. primeros3Alumnos
 - f. menorEdad: el alumno de menor edad
 - g. elPrimero: el primer alumno de la lista
 - h. filtradoCaracter: que el nombre tenga el carácter recibido

EXAMEN!!!

Ingresa a este enlace y responde según creas correcto:

<https://tinyurl.com/j8examen1>

2.

Lambda incorporando interfaces funcionales



2.1

Lambda incorporando interfaces funcionales



Interfaces Funcionales

Una interface funcional es aquella que tiene un único método abstracto. Así de sencillo. No importa que además tenga métodos estáticos o implementados por defecto.

Java 8 añadió la anotación `@FunctionalInterface` para marcar este tipo de interfaces. Es una anotación **meramente informativa**, usada para reforzar la intención de las interfaces, pero que permite a los compiladores generar un error si se aplica sobre una interface que no cumpla con la definición de interface funcional, o sobre una clase, enumerado u otra anotación.

```
@FunctionalInterface
interface MiFunc{
    MiClase func(String s);
}

class MiClase{
    private static String str;

    MiClase(String s){
        str=s;
    }

    String getStr() {
        return str;
    }
}
```

Interfaces Funcionales

Las interfaces funcionales se pueden instanciar con expresiones lambda, referencias a métodos o referencias a constructores.

```
Computer lambda = () -> {};
```

```
Computer metodo = System::gc;
```

```
Computer constructor = String::new;
```

Java 8 añadió una gran cantidad de interfaces funcionales dentro del paquete `java.util.function`, tanto para uso interno como para su uso general. Las interfaces principales son `Consumer<T>`, `Supplier<T>`, `Function<T, R>` y `Predicate<T>`, el resto son especializaciones de estas.

Interfaces Funcionales - Nomenclatura

La nomenclatura utilizada para el nombre de las clases especializadas indica cómo se diferencian de la interfaces base que especializan:

- ◆ El prefijo Bi indica que admite dos parámetros.
- ◆ El prefijo Boolean, Int, Long y Double indica que admite un parámetro o genera un resultado de tipo boolean, int, long o double respectivamente.
- ◆ El prefijo ObjInt, ObjLong y ObjDouble indica que admite dos parámetros, el primero del mismo tipo que la interface, y el segundo de tipo int, long o double respectivamente.
- ◆ El prefijoToInt, ToLong y ToDouble indica que produce un resultado de tipo int, long o double respectivamente.
- ◆ El prefijo IntToLong, InttoDouble, LongToInt, LongtoDouble, DoubleToInt y DoubleToLong indica que admite un parámetro del primer tipo indicado y produce un resultado del segundo.
- ◆ El sufijo Operator indica una especialización de Function.

Interfaces funcionales

Java 8 incluye varias interfaces funcionales nuevas. Las más comunes son `Function<T, R>`, `Predicate<T>` y `Consumer<T>`, que están definidas en el paquete `java.util.function`.

El método `map` de `Stream` toma `Function<T, R>` como parámetro. De igual manera, `filter` utiliza `Predicate<T>` y `forEach` utiliza `Consumer<T>`.

El paquete también tiene otras interfaces funcionales, como `Supplier<T>`, `BiConsumer<T, U>` y `BiFunction<T, U, R>`.

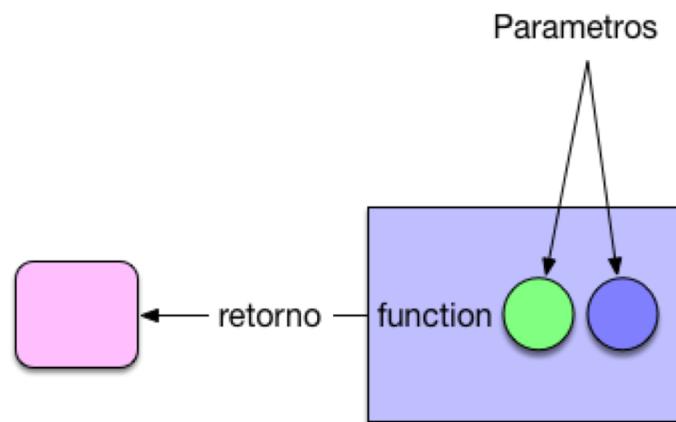
2.2

Predicate, Consumer, Function y Supplier



Function

Reciben uno o más parámetros y retornan un resultado:



Function

Este es el caso de uso de un stream cuando utiliza el método map recibe como parámetro un tipo Function:

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

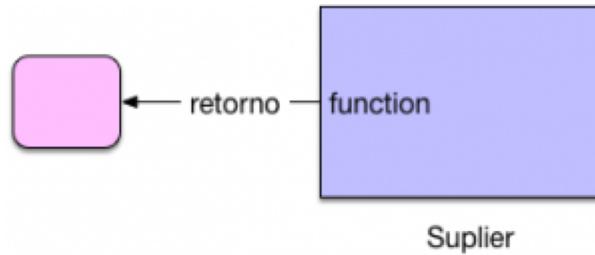
public class main{

    public static void main(String[] args) {

        List<String> lista=Arrays.asList("hola","que","tal");
        lista.stream().map((x)->x.toUpperCase()).forEach((x)->System.out.println(x));
    }
}
```

Supplier

Este caso es algo más complejo ya que se trata de una función que no recibe parámetro alguno y devuelve un resultado.



Supplier

Por ejemplo

```
public class main{  
    public static void main(String[] args) {  
        Supplier<String> i = ()-> "Java 8 Rocks!";  
        System.out.println(i.get());  
  
        Supplier<Double> randomValue = () -> Math.random();  
        System.out.println(randomValue.get());  
  
    }  
}
```

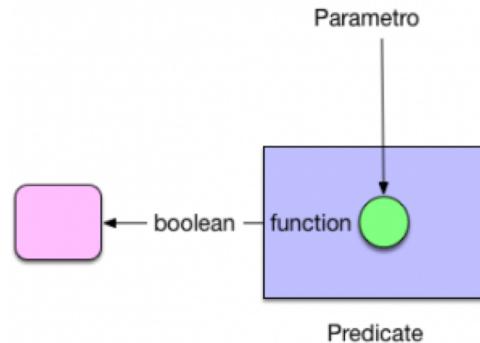
Supplier

Supplier tiene una variedad de interfaces para manejar distintos tipos de datos:

- ◆ IntSupplier
- ◆ LongSupplier
- ◆ BooleanSupplier
- ◆ DoubleSupplier

Predicate

Las interfaces de tipo predicado son una especialización de los Functions ya que, tal como en las matemáticas, reciben un parámetro y retornan un valor booleano.



Predicate

Por ejemplo

Se usan de forma intensiva en operaciones de filtrado:

```
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
import java.util.stream.Stream;  
  
public class Principal4 {  
    public static void main(String[] args) {  
        List<Integer> lista=Arrays.asList(1,3,4,5,6,7,1,2);  
        lista.stream().filter((x)->x>5).forEach(System.out::println);  
    }  
}
```

Predicate

También existen interfaces:

- ◆ IntPredicate
- ◆ LongPredicate
- ◆ BiPredicate
- ◆ DoublePredicate

```
// SIMPLE
IntPredicate i = (x) -> x < 0;
System.out.println(i.test(123));

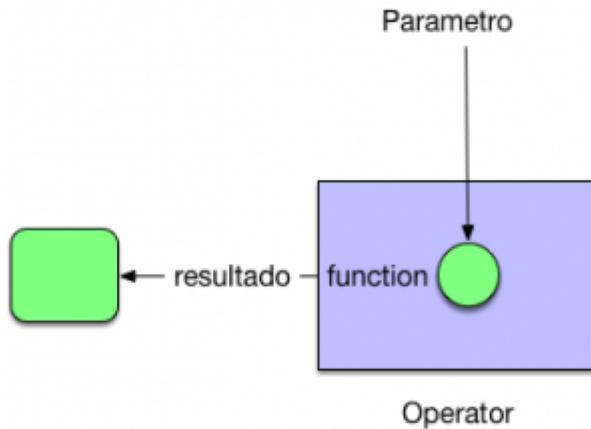
// USANDO AND
IntPredicate j = (x) -> x == 0;
System.out.println(i.and(j).test(123));

// CON 2 VALORES
BiPredicate<Integer, Integer> bi = (x, y) ->
    x > y;
System.out.println(bi.test(2, 3));

// CON NEGACION
DoublePredicate dp = (d) -> d > 0;
System.out.println(dp.test(0.5));
System.out.println(dp.negate().test(0.5));
```

Operators

El uso de Java 8 Operators es otro caso específico de los Function interfaces . Se trata de interfaces funcionales que reciben un tipo de parámetro y devuelven un resultado que es del mismo tipo.



Operators

Por ejemplo

Un ejemplo típico usando streams es el resultado que genera el método reduce ya que se encarga de agrupar un conjunto de elementos del mismo tipo:

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

public class Principal5 {

    public static void main(String[] args) {
        List<String> lista=Arrays.asList("hola","que","tal");
        Optional<String> resultado=lista.stream().reduce(String::concat);
        if(resultado.isPresent()) {

            System.out.println(resultado.get());
        }
    }
}
```

2.3

El uso de versiones primitivas de interfaces base



El uso de versiones primitivas de interfaces base

Ya vimos la inclusión de las interfaces funcionales, interfaces regulares que implementan un método abstracto y que, además, sirven para usarse a través de expresiones lambda y referencias a métodos.

Tenemos el paquete `java.util.function`, que consiste en proveer interfaces funcionales predefinidas, teniendo como base las interfaces `Function`, `Predicate`, `Supplier` y `Consumer`.

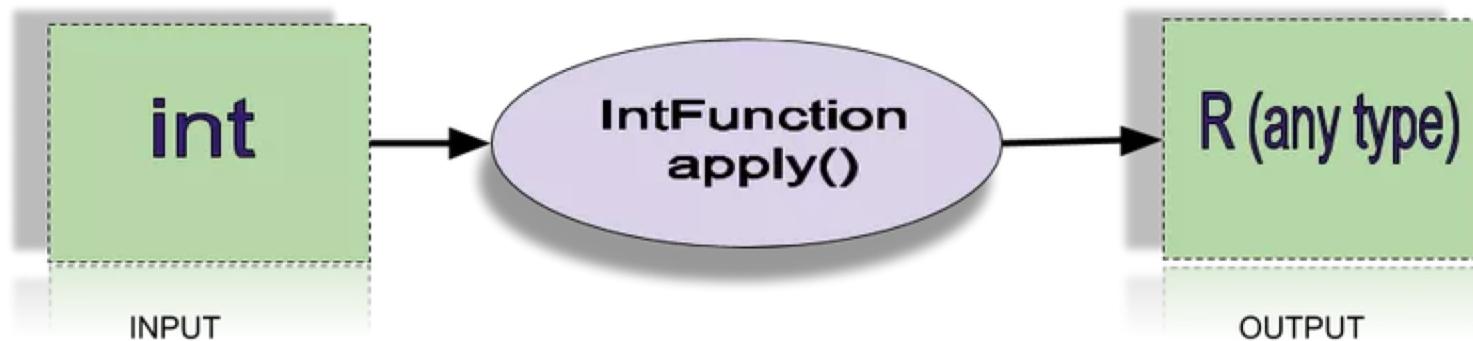
El resto de las interfaces contenidas en el paquete son variaciones primitivas o binarias de estas 4.

Veamos el caso de las primitivas.

El uso de versiones primitivas de interfaces base

IntFunction

Esta interfaz representa una función que obtiene un parámetro de tipo entero y retorna un valor de tipo R.



El uso de versiones primitivas de interfaces base

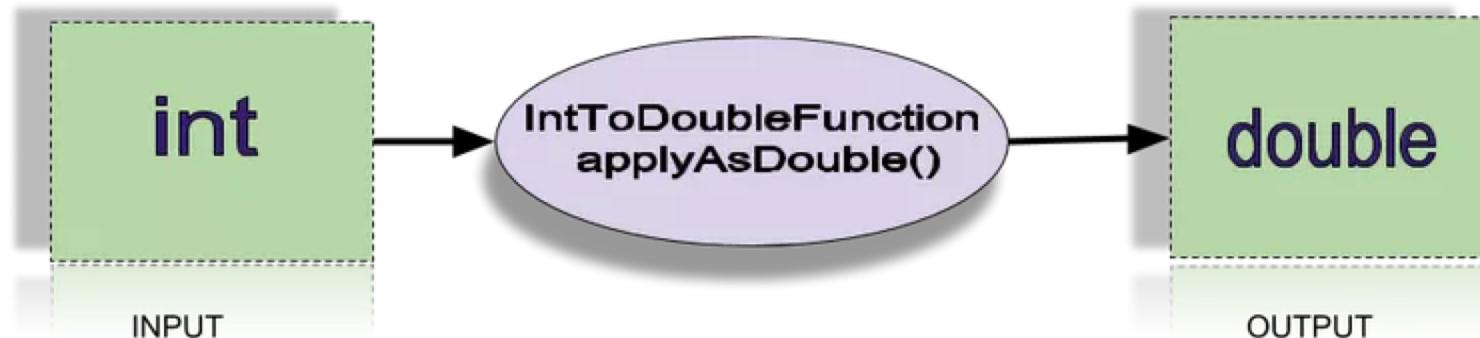
IntFunction - Ejemplo

```
public class Main {  
    public static void main(String[] args) {  
        IntFunction<String> getMonthOfTheYearName = mes -> {  
            Map<Integer, String> meses = new HashMap<>();  
            meses.put(1, "Enero");  
            meses.put(2, "Febrero");  
            meses.put(3, "MarMarzoch");  
            meses.put(4, "Abril");  
            meses.put(5, "Mayo");  
  
            if (meses.get(mes) != null) {  
                return meses.get(mes);  
            } else {  
                return "La entrada debe ser entre 1 y 5";  
            }  
        };  
        int input = 1;  
        String mesEncontrado = getMonthOfTheYearName.apply(input);  
        System.out.println("Mes nro. " + input + " es : " + mesEncontrado);  
    }  
}
```

El uso de versiones primitivas de interfaces base

IntToDoubleFunction

Esta interfaz representa una función que obtiene un parámetro de tipo entero y retorna un valor de tipo double.



El uso de versiones primitivas de interfaces base

IntToDoubleFunction - Ejemplo

```
public static void main(String[] args) {
    IntToDoubleFunction getDouble = intVal->{
        double doubleVal = intVal;
        return doubleVal;
    };

    int input = 10;
    double result = getDouble.applyAsDouble(input);
    System.out.println("Entrada "+input+" como double : "+result);

}
```

El uso de versiones primitivas de interfaces base

IntToLongFunction

Esta interfaz representa una función que obtiene un parámetro de tipo entero y retorna un valor de tipo long.



El uso de versiones primitivas de interfaces base

IntToLongFunction- Ejemplo

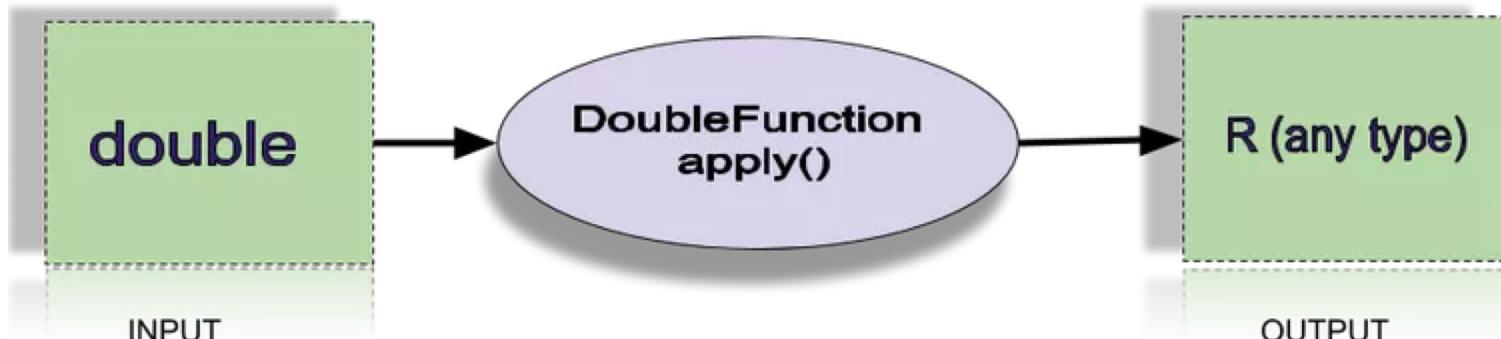
```
public static void main(String[] args) {
    IntToLongFunction getLong = intVal->{
        long longVal = intVal;
        return longVal;
    };

    int input = 10;
    long result = getLong.applyAsLong(input);
    System.out.println("Input "+input+" as long : "+result);
}
```

El uso de versiones primitivas de interfaces base

DoubleFunction

Esta interfaz representa una función que obtiene un parámetro de tipo double y retorna un valor de tipo R



El uso de versiones primitivas de interfaces base

DoubleFunction - Ejemplo

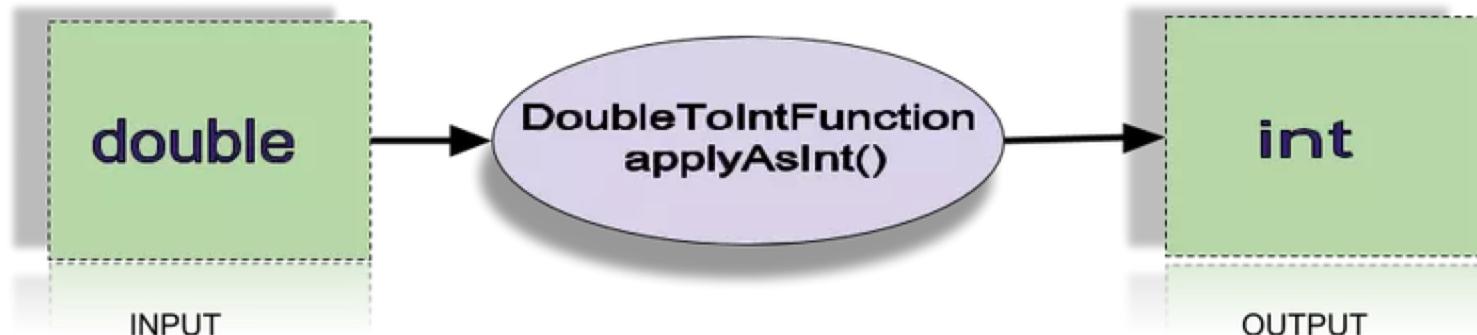
```
public static void main(String[] args) {
    DoubleFunction<String> getGrado = nivel->{
        if(nivel>90 && nivel<=100){
            return "A";
        } else if(nivel>70 && nivel<=90){
            return "B";
        } else if(nivel>50 && nivel<=70){
            return "C";
        } else {
            return "D";
        }
    };

    double input = 91;
    System.out.println("Nivel de entrada: "+input);
    String grado = getGrado.apply(input);
    System.out.println("Grado : "+grado);
```

El uso de versiones primitivas de interfaces base

DoubleToIntFunction

Esta interfaz representa una función que obtiene un parámetro de tipo double y retorna un valor de tipo entero



El uso de versiones primitivas de interfaces base

DoubleToIntFunction - Ejemplo

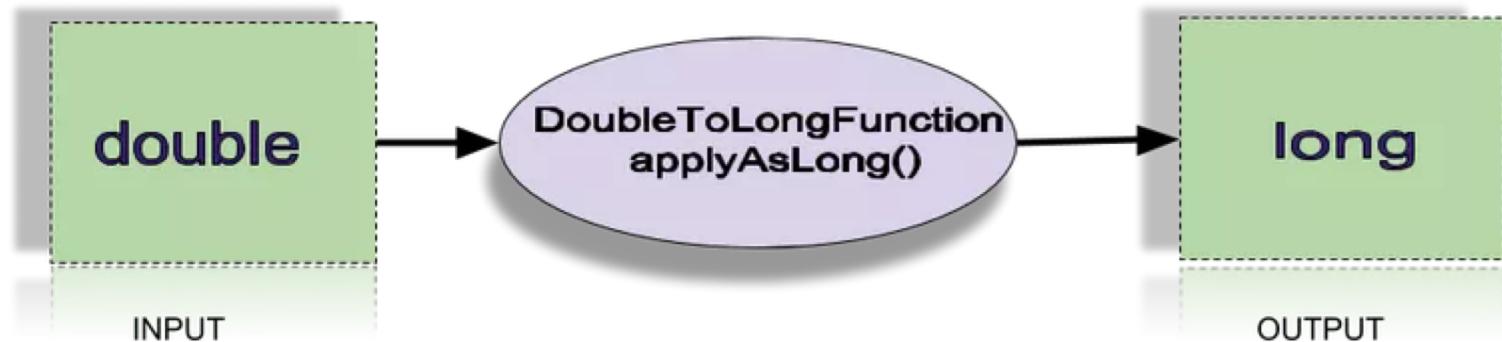
```
public static void main(String[] args) {
    DoubleToIntFunction getInt = doubleVal->{
        int intVal = (int) doubleVal;
        return intVal;
    };

    double input=23.76;
    System.out.println("input : "+input);
    int result = getInt.applyAsInt(input);
    System.out.println("Result : "+result);
}
```

El uso de versiones primitivas de interfaces base

DoubleToLongFunction

Esta interfaz representa una función que obtiene un parámetro de tipo double y retorna un valor de tipo long



El uso de versiones primitivas de interfaces base

DoubleToLongFunction - Ejemplo

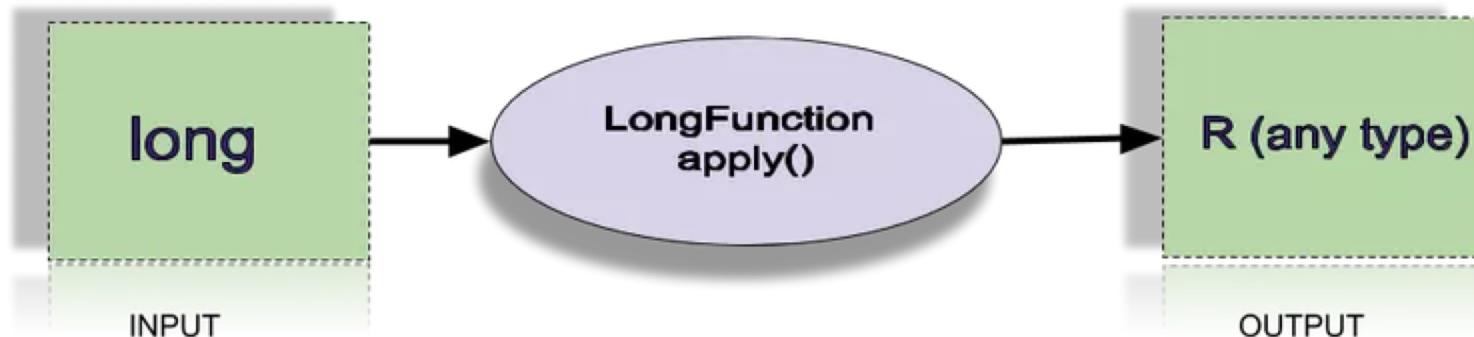
```
public static void main(String[] args) {
    DoubleToLongFunction getLong = doubleVal->{
        long longVal = (long) doubleVal;
        return longVal;
    };

    double input = 23.165;
    System.out.println("input : "+input);
    long result = getLong.applyAsLong(input);
    System.out.println("result : "+result);
}
```

El uso de versiones primitivas de interfaces base

LongFunction

Esta interfaz representa una función que obtiene un parámetro de tipo long y retorna un valor de tipo R



El uso de versiones primitivas de interfaces base

LongFunction - Ejemplo

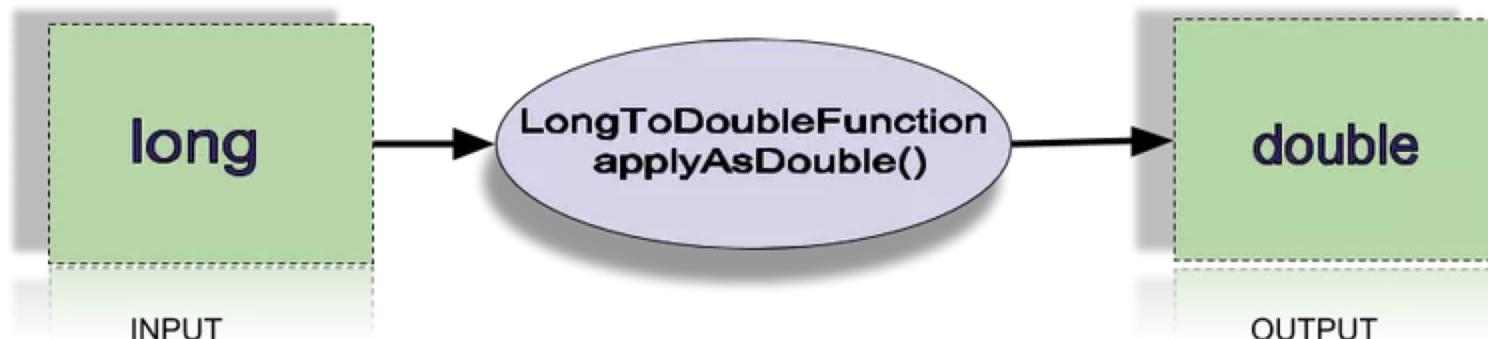
```
public static void main(String[] args) {
    LongFunction<String> getStatus = tramitesExtranjeria->{
        Map<Long,String> tramites = new HashMap<>();
        tramites.put(912312312L, "Proceso Completado");
        tramites.put(844312343L, "In Progress");
        tramites.put(111312353L, "Rechazado");
        tramites.put(777312399L, "Esperando Aprobación");

        if(tramites.containsKey(tramitesExtranjeria)) {
            return tramites.get(tramitesExtranjeria);
        } else {
            return "Nro de Trámite Inválido";
        }
    };
    long input = 912312312L;
    String status = getStatus.apply(input);
    System.out.println("Status de trámite Nro. "+input+": "+status);
}
```

El uso de versiones primitivas de interfaces base

LongToDoubleFunction

Esta interfaz representa una función que obtiene un parámetro de tipo long y retorna un valor de tipo double



El uso de versiones primitivas de interfaces base

LongToDoubleFunction - Ejemplo

```
public static void main(String[] args) {
    LongToDoubleFunction getDouble = longVal->{
        double doubleVal = longVal;
        return doubleVal;
    };

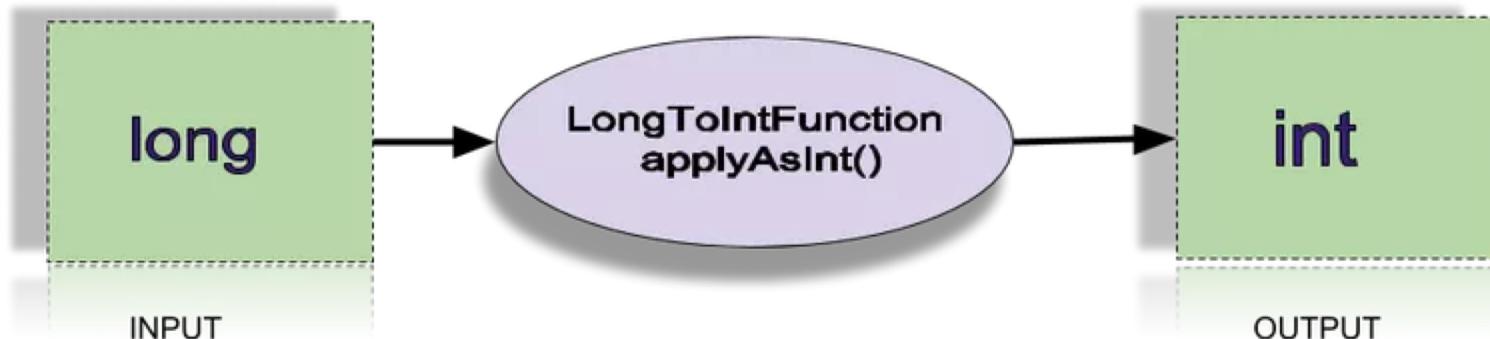
    long input = 675764L;
    System.out.println("input: "+input);

    double result = getDouble.applyAsDouble(input);
    System.out.println("result : "+result);
}
```

El uso de versiones primitivas de interfaces base

LongToIntFunction

Esta interfaz representa una función que obtiene un parámetro de tipo long y retorna un valor de tipo entero



El uso de versiones primitivas de interfaces base

LongToIntFunction - Ejemplo

```
public static void main(String[] args) {
    LongToIntFunction getInt = longVal->{
        int intVal = (int) longVal;
        return intVal;
    };

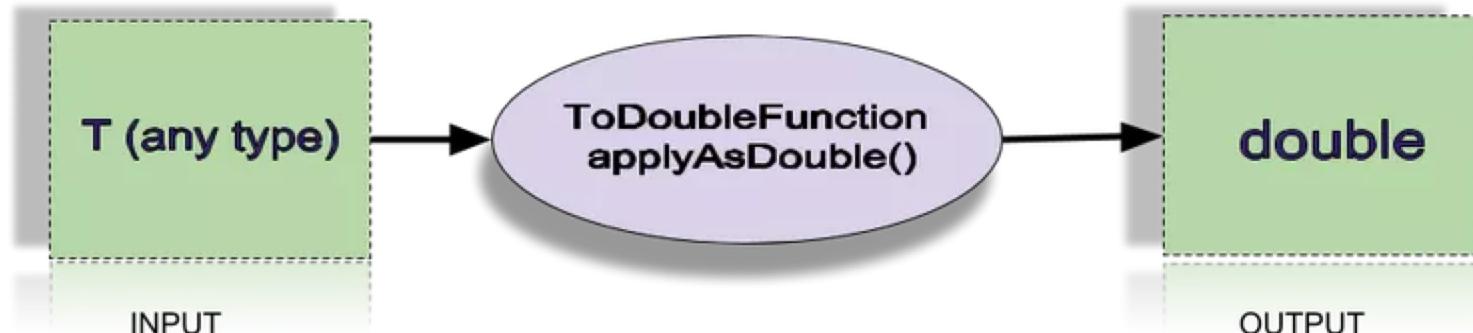
    long input = 675764L;
    System.out.println("input: "+input);

    int result = getInt.applyAsInt(input);
    System.out.println("result : "+result);
}
```

El uso de versiones primitivas de interfaces base

ToDoubleFunction

Esta interfaz representa una función que obtiene un parámetro de tipo T y retorna un valor de tipo double



El uso de versiones primitivas de interfaces base

ToDoubleFunction - Ejemplo

```
public static void main(String[] args) {
    ToDoubleFunction<String> getMarks = student->{
        Map<String,Double> studentMarks = new HashMap<>();
        studentMarks.put("Anthony Lane", 87.0);
        studentMarks.put("Jade Pluk", 34.0);
        studentMarks.put("Olivia Tanner",76.5);
        studentMarks.put("Jesse Jones", 66.5);

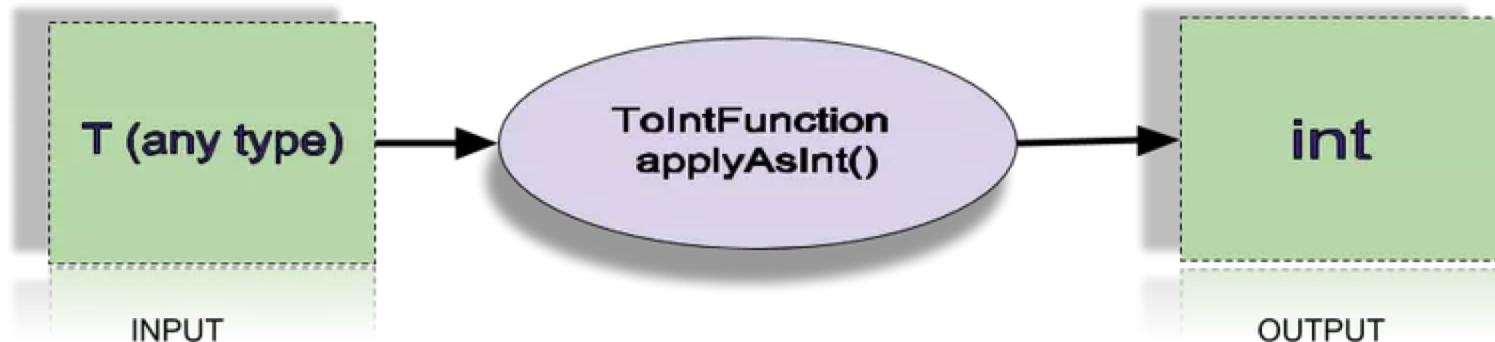
        if(studentMarks.containsKey(student)){
            return studentMarks.get(student);
        } else {
            return 00.00;
        }
    };

    String input = "Anthony Lane";
    double marks = getMarks.applyAsDouble(input);
    System.out.println("Marks of Student "+input+": "+marks);
}
```

El uso de versiones primitivas de interfaces base

ToIntFunction

Esta interfaz representa una función que obtiene un parámetro de tipo T y retorna un valor de tipo entero



El uso de versiones primitivas de interfaces base

ToIntFunction - Ejemplo

```
public static void main(String[] args) {
   ToIntFunction<String> getMarks = student->{
        Map<String,Integer> studentMarks = new HashMap<>();
        studentMarks.put("Anthony Lane", 87);
        studentMarks.put("Jade Pluk", 34);
        studentMarks.put("Olivia Tanner",76);
        studentMarks.put("Jesse Jones", 66);

        if(studentMarks.containsKey(student)){
            return studentMarks.get(student);
        } else {
            return 00;
        }
    };

    String input = "Anthony Lane";
    int marks = getMarks.applyAsInt(input);
    System.out.println("Marks of Student "+input+": "+marks);
}
```

El uso de versiones primitivas de interfaces base

ToLongFunction

Esta interfaz representa una función que obtiene un parámetro de tipo T y retorna un valor de tipo long



El uso de versiones primitivas de interfaces base

ToLongFunction - Ejemplo

```
public static void main(String[] args) {
    ToLongFunction<String> getPhoneNo = person->{
        Map<String,Long> phoneNos = new HashMap<>();
        phoneNos.put("Anthony Lane", 876543871L);
        phoneNos.put("Jade Pluk", 834543982L);
        phoneNos.put("Olivia Tanner",876321654L);
        phoneNos.put("Jesse Jones", 866327745L);

        if(phoneNos.containsKey(person)){
            return phoneNos.get(person);
        } else {
            return 00;
        }
    };

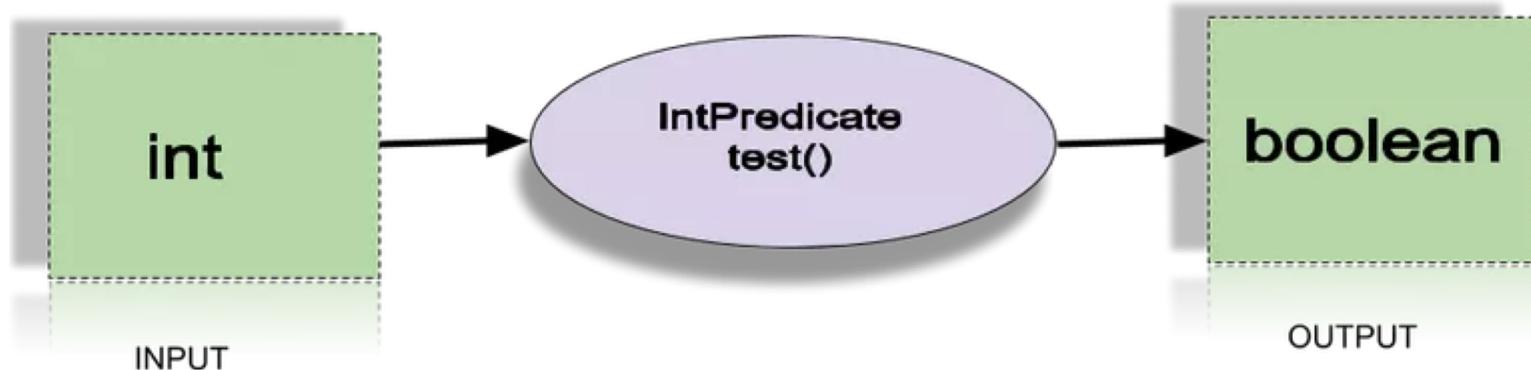
    String input = "Anthony Lane";
    long phoneNo = getPhoneNo.applyAsLong(input);
    System.out.println("Phone number of "+input+": "+phoneNo);
}
```

El uso de versiones primitivas de interfaces base

Estas son las variaciones de la interfaz base Function, veamos ahora las variaciones primitivas de la interfaz base Predicate.

IntPredicate

Es la representación de una función que recibe un parámetro de tipo entero y retorna un valor booleano.



El uso de versiones primitivas de interfaces base

IntPredicate

Sus métodos por defecto son:

- `and()`
- `or()`
- `negate()`

`default IntPredicate and(IntPredicate other)`

`default IntPredicate or(IntPredicate other)`

`default IntPredicate negate()`

El uso de versiones primitivas de interfaces base

IntPredicate - Ejemplo

```
public static void main(String[] args) {
    IntPredicate buscaPares = i->{
        if(i%2==0){
            return true;
        }
        return false;
    };

    System.out.println("10 es par? : "+buscaPares.test(10));
    System.out.println("111 es par? : "+buscaPares.test(111));
    System.out.println("16 es par? : "+buscaPares.test(16));
}
```

El uso de versiones primitivas de interfaces base

IntPredicate - Ejemplo

```
public static void main(String[] args) {
    IntPredicate isWhitespace = Character::isWhitespace;
    IntPredicate isUpper = Character::isUpperCase;
    IntConsumer print = System.out::println;

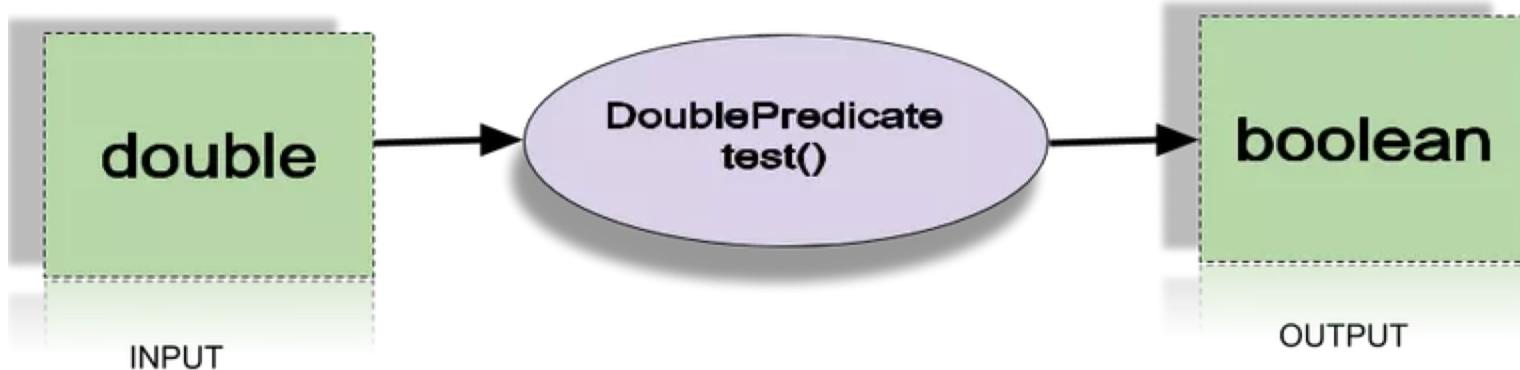
    "aB cD eF".chars().filter(isWhitespace.negate().and(isUpper)).forEach(print);

}
```

El uso de versiones primitivas de interfaces base

DoublePredicate

Es la representación de una función que recibe un parámetro de tipo double y retorna un valor booleano.



El uso de versiones primitivas de interfaces base

DoublePredicate

Sus métodos por defecto son:

- `and()`
- `or()`
- `negate()`

`default DoublePredicate and(DoublePredicate other)`

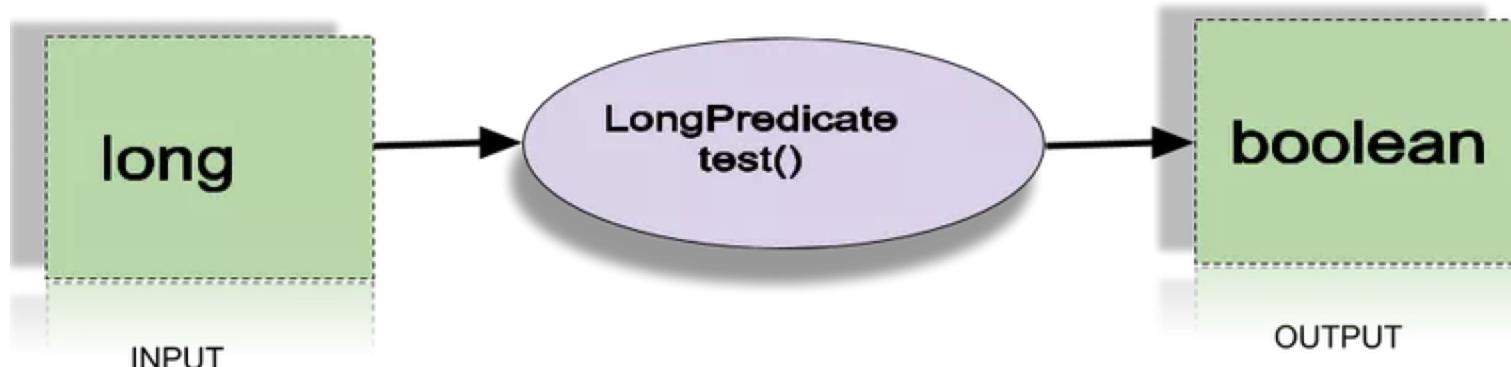
`default DoublePredicate or(DoublePredicate other)`

`default DoublePredicate negate()`

El uso de versiones primitivas de interfaces base

LongPredicate

Es la representación de una función que recibe un parámetro de tipo long y retorna un valor booleano.



El uso de versiones primitivas de interfaces base

LongPredicate

Sus métodos por defecto son:

- `and()`
- `or()`
- `negate()`

`default LongPredicate and(LongPredicate other)`

`default LongPredicate or(LongPredicate other)`

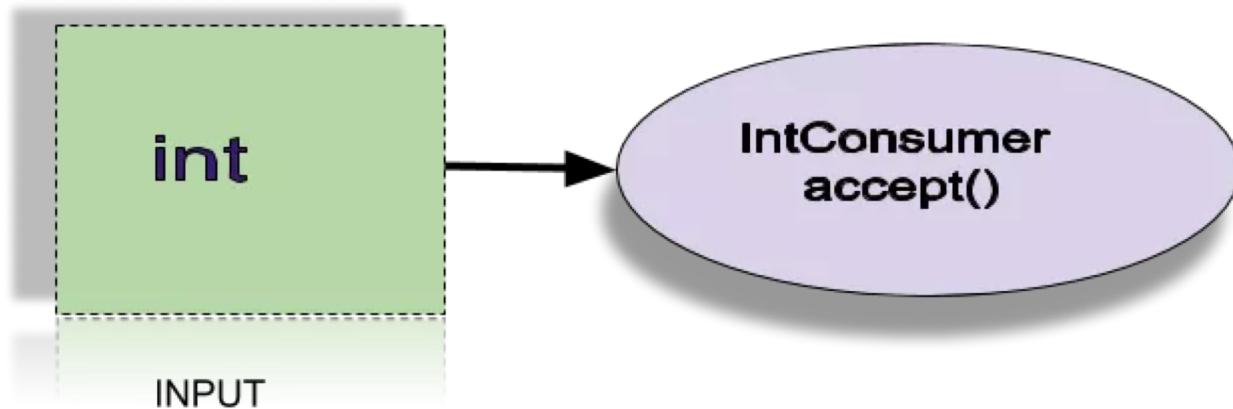
`default LongPredicate negate()`

El uso de versiones primitivas de interfaces base

Estas son las variaciones de la interfaz base Predicate, veamos ahora las variaciones primitivas de la interfaz base Consumer.

IntConsumer

Es la representación de una función que recibe un parámetro de tipo entero y no retorna valor.



El uso de versiones primitivas de interfaces base

IntConsumer

Su método por defecto es:

default IntConsumer andThen(IntConsumer after)

El uso de versiones primitivas de interfaces base

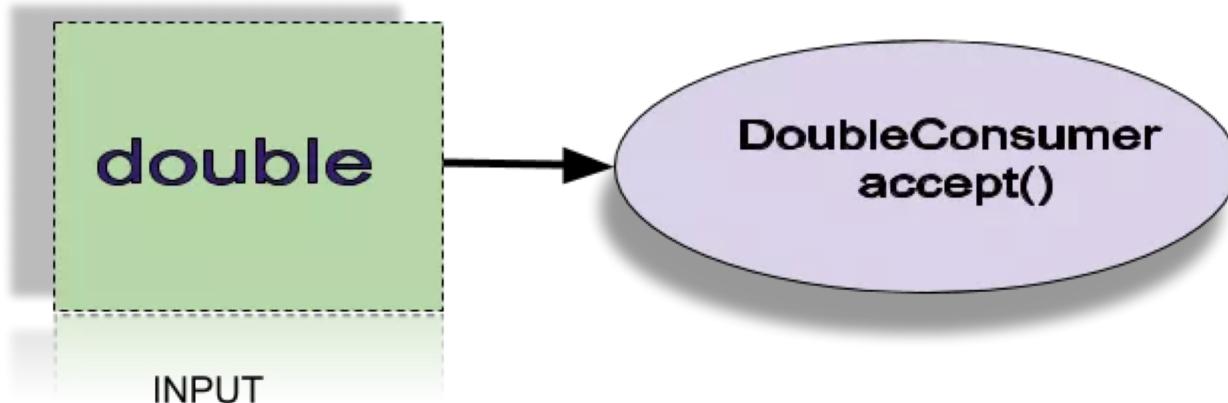
IntConsumer - Ejemplo

```
public class Main {  
  
    public static void main(String[] args) {  
        IntConsumer displayNextInt = i->System.out.println("Next: "+(i+1));  
  
        IntConsumer displaySquare = i->System.out.println("Square: "+(i*i));  
  
        IntConsumer displayBoth = displayNextInt.andThen(displaySquare);  
  
        displayBoth.accept(20);  
  
    }  
}
```

El uso de versiones primitivas de interfaces base

DoubleConsumer

Es la representación de una función que recibe un parámetro de tipo double y no retorna valor.



El uso de versiones primitivas de interfaces base

DoubleConsumer

Su método por defecto es:

default DoubleConsumer andThen(DoubleConsumer after)

El uso de versiones primitivas de interfaces base

DoubleConsumer - Ejemplo

```
public static void main(String[] args) {
    DoubleConsumer incrementByOne = doubleVal->{
        System.out.println("Incrementing "+doubleVal+" by one... ");
        System.out.println("Current Value : "+(doubleVal+1));
    };

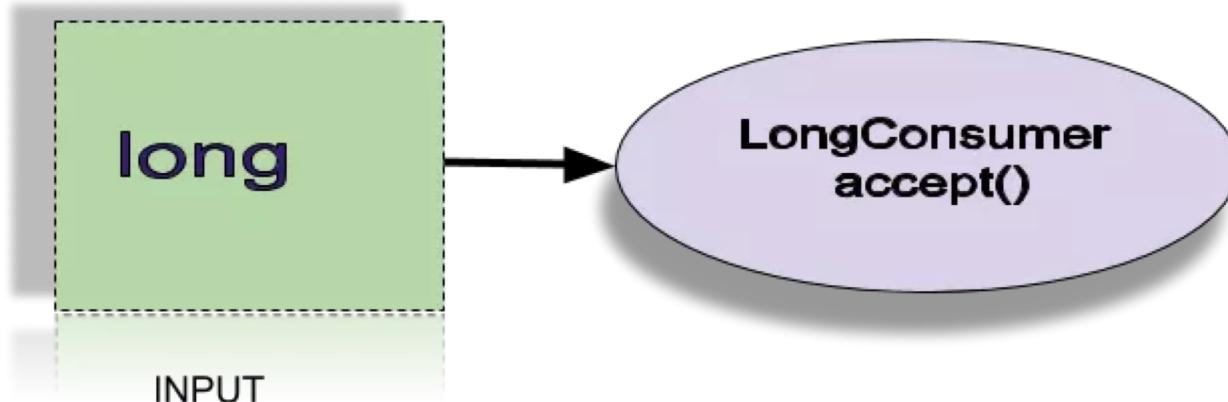
    DoubleConsumer decrementByOne = doubleVal->{
        System.out.println("Decrementing "+doubleVal+" by one... ");
        System.out.println("Current Value : "+(doubleVal-1));
    };

    DoubleConsumer doAction = incrementByOne.andThen(decrementByOne);
    doAction.accept(1000);
}
```

El uso de versiones primitivas de interfaces base

LongConsumer

Es la representación de una función que recibe un parámetro de tipo long y no retorna valor.



El uso de versiones primitivas de interfaces base

LongConsumer

Su método por defecto es:

default LongConsumer andThen(LongConsumer after)

El uso de versiones primitivas de interfaces base

LongConsumer - Ejemplo

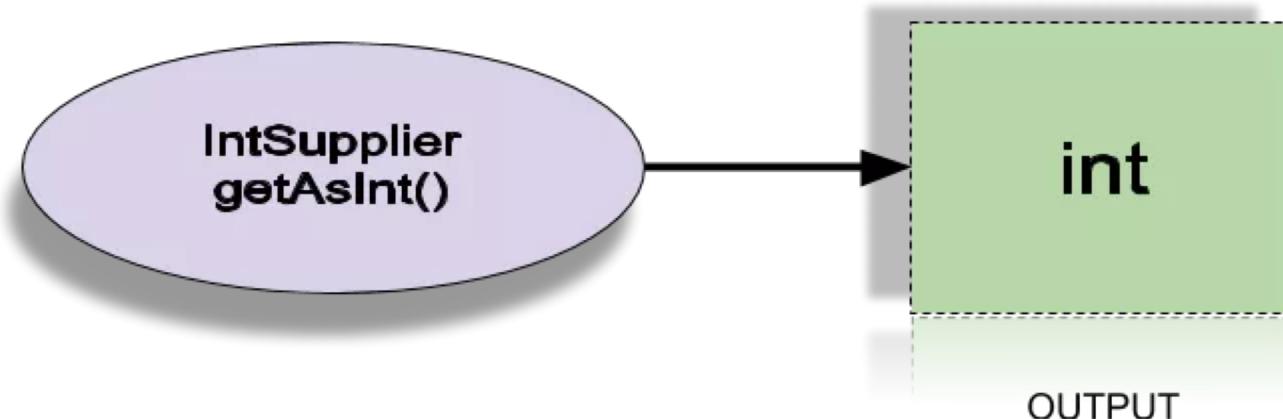
```
public class Exercise {  
    public static void main(String[] args) {  
        LongConsumer displayNextVal = l->{  
            System.out.println("Display the next value to input : "+l);  
            System.out.println(l+1);  
        };  
  
        LongConsumer displayPrevVal = l->{  
            System.out.println("Display the previous value to input : "+l);  
            System.out.println(l-1);  
        };  
  
        LongConsumer displayPrevAndNextVal = displayNextVal.andThen(displayPrevVal);  
  
        displayPrevAndNextVal.accept(1000);  
    }  
}
```

El uso de versiones primitivas de interfaces base

Estas son las variaciones de la interfaz base Consumer, veamos ahora las variaciones primitivas de la interfaz base Supplier.

IntSupplier

Es la representación de una función que no recibe parámetros y retorna un valor entero.



El uso de versiones primitivas de interfaces base

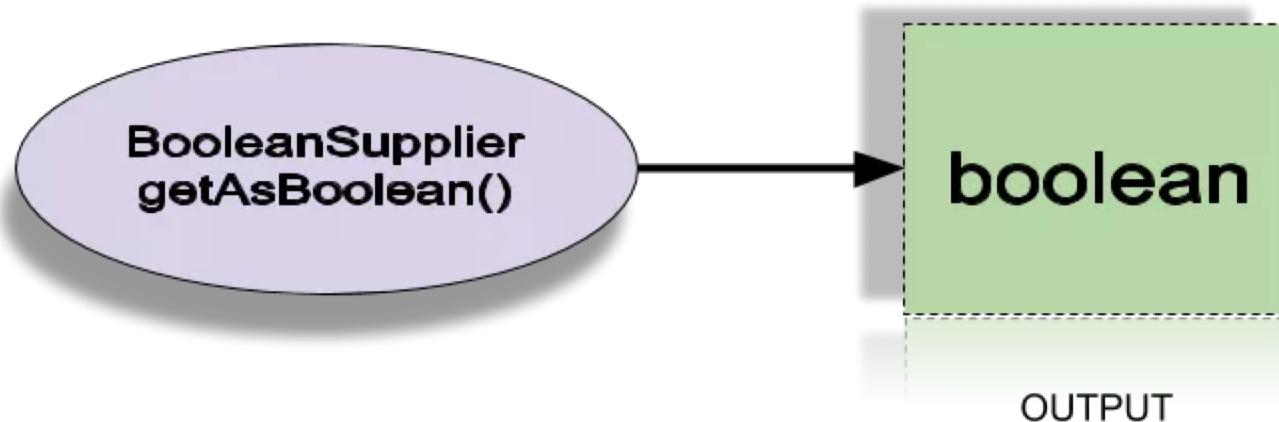
IntSupplier - Ejemplo

```
public class Exercise {  
    public static void main(String[] args) {  
        IntSupplier getRandomInt = ()->{  
            return ThreadLocalRandom.current().nextInt(0000,9999);  
        };  
  
        int randomVal = getRandomInt.getAsInt();  
        System.out.println("Random Integer Generated : "+randomVal);  
  
        //Generate some more random numbers  
        System.out.println("Random Integer Generated : "+getRandomInt.getAsInt());  
        System.out.println("Random Integer Generated : "+getRandomInt.getAsInt());  
        System.out.println("Random Integer Generated : "+getRandomInt.getAsInt());  
  
    }  
}
```

El uso de versiones primitivas de interfaces base

BooleanSupplier

Es la representación de una función que no recibe parámetros y retorna un valor booleano.



El uso de versiones primitivas de interfaces base

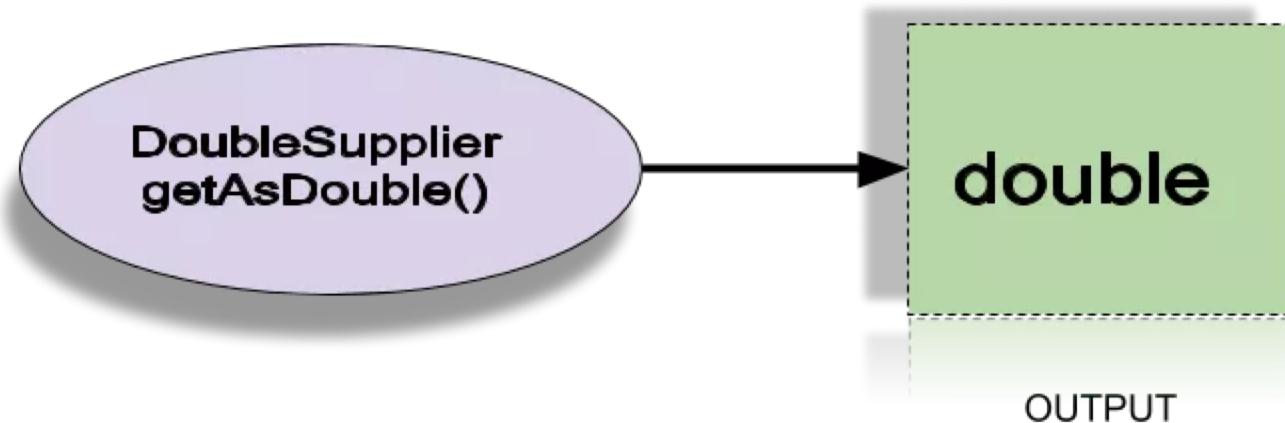
BooleanSupplier - Ejemplo

```
public class Exercise {  
    public static void main(String[] args) {  
        BooleanSupplier checkIFCurrentDateisEven = ()->{  
            Calendar cal = Calendar.getInstance();  
            Integer currentDate = cal.get(Calendar.DAY_OF_MONTH);  
            if(currentDate%2==0){  
                return true;  
            }  
            return false;  
        };  
  
        boolean isCurrentDateEven = checkIFCurrentDateisEven.getAsBoolean();  
  
        System.out.println("Is Current Date Even? :" +isCurrentDateEven);  
    }  
}
```

El uso de versiones primitivas de interfaces base

DoubleSupplier

Es la representación de una función que no recibe parámetros y retorna un valor double.



El uso de versiones primitivas de interfaces base

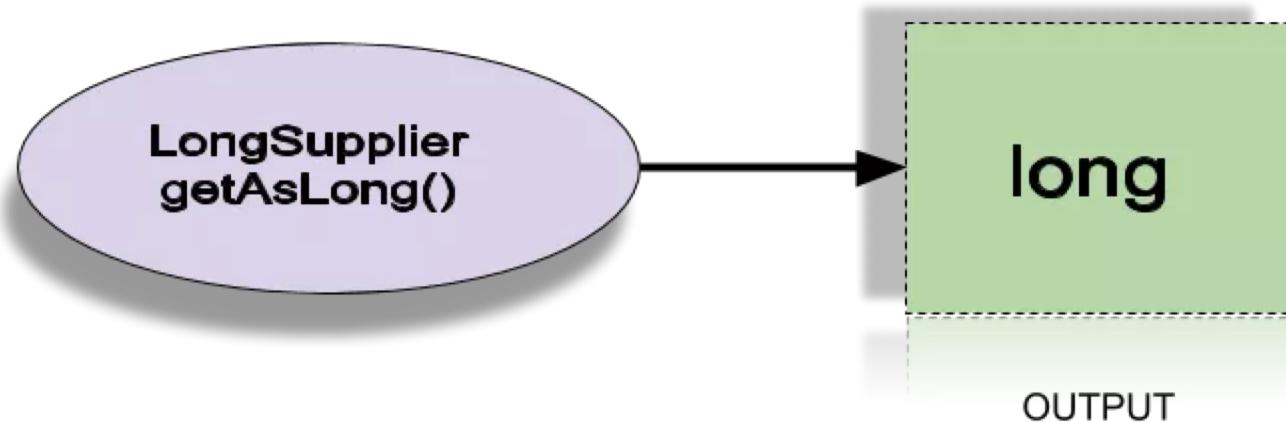
DoubleSupplier - Ejemplo

```
public class Exercise {  
    public static void main(String[] args) {  
        DoubleSupplier getRandomDouble = ()->{  
            double doubleVal = ThreadLocalRandom.current().nextDouble(0000, 9999);  
            return Math.round(doubleVal);  
        };  
  
        double randomVal = getRandomDouble.getAsDouble();  
        System.out.println("Random Double Generated : "+randomVal);  
  
        //Generate some more random numbers  
        System.out.println("Random Double Generated : "+getRandomDouble.getAsDouble());  
        System.out.println("Random Double Generated : "+getRandomDouble.getAsDouble());  
        System.out.println("Random Double Generated : "+getRandomDouble.getAsDouble());  
  
    }  
}
```

El uso de versiones primitivas de interfaces base

LongSupplier

Es la representación de una función que no recibe parámetros y retorna un valor Long.



El uso de versiones primitivas de interfaces base

DoubleSupplier - Ejemplo

```
public class Exercise {  
    public static void main(String[] args) {  
        LongSupplier getRandomLong = ()->{  
            long longVal = ThreadLocalRandom.current().nextLong(10000000, 99999999);  
            return longVal;  
        };  
  
        long randomVal = getRandomLong.getAsLong();  
        System.out.println("Random Long Generated : "+randomVal);  
  
        //Generate some more random numbers  
        System.out.println("Random Long Generated : "+getRandomLong.getAsLong());  
        System.out.println("Random Long Generated : "+getRandomLong.getAsLong());  
        System.out.println("Random Long Generated : "+getRandomLong.getAsLong());  
  
    }  
}
```

2.4

El uso de versiones binarias de interfaces base



El uso de versiones binarias de interfaces base

Habiendo visto las variaciones primitivas de las interfaces base, podemos proceder a ver las variaciones binarias, en concreto:

- BiFunction
- BiConsumer
- BiPredicate

El uso de versiones binarias de interfaces base

BiFuction

Una interfaz BiFuncional debe contener un método abstracto, en este caso, el método que implementa es **apply()** y recibe dos parámetros de tipo T y U para retornar un valor de tipo R

R **apply(T t, U u)**

El uso de versiones binarias de interfaces base

BiFunction

Además, esta interfaz implementa un método por defecto, **andThen()**, cuya funcionalidad radica en llamar al método `apply()` de forma secuencial o controlada.

```
default <V> BiFunction<T,U,V> andThen(Function<? super R,> extends V> after)
```

El uso de versiones binarias de interfaces base

BiFunction - Ejemplo del apply()

```
public static void main(String[] args) {
    BiFunction<List<String>,String,Boolean> checkValueInList = (list,value)->{
        if(list!=null && list.contains(value)){
            return true;
        } else {
            return false;
        }
    };

    List<String> values = new ArrayList<String>();
    values.add("item1");
    values.add("item2");
    values.add("item3");
    values.add("item4");
    values.add("item5");

    boolean valid = checkValueInList.apply(values, "item3");
    System.out.println("La lista contiene item3? :" +valid);

    System.out.println("La lista contiene item1? :" +checkValueInList.apply(values, "item1"));
    System.out.println("La lista contiene item6? :" +checkValueInList.apply(values, "item6"));
    System.out.println("La lista contiene item9? :" +checkValueInList.apply(values, "item9"));
}
```

El uso de versiones binarias de interfaces base

BiConsumer

Representa una función que recibe dos argumentos pero no retorna ningún valor, por lo que el manejo de los valores recibidos recae en la lógica de negocio interna de la función.

Al igual que la BiFunction, esta interfaz consta de dos métodos

Accept()

andThen()

El uso de versiones binarias de interfaces base

BiConsumer- Ejemplo

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

        BiConsumer<List<String>, String> biConsumer = (list, value) -> {
            // print every item on the list except value
            for (String item : list) {
                if (!item.equals(value)) {
                    System.out.println(item);
                }
            }
        };

        List<String> values = new ArrayList<String>();
        values.add("item1");
        values.add("item2");
        values.add("item3");
        values.add("item4");
        values.add("item5");
        biConsumer.accept(values, "item3");

    }
}
```

El uso de versiones binarias de interfaces base

BiPredicate

Representa una función que recibe dos argumentos y retorna un valor booleano, por lo que la lógica de negocio deberá apuntar a validaciones que retornen este tipo de valor.

Métodos por defecto:

default BiPredicate<T,U> and(BiPredicate<? super T,> other)

default BiPredicate<T,U> negate()

default BiPredicate<T,U> or(BiPredicate<? super T,> other)

El uso de versiones binarias de interfaces base

BiPredicate - Ejemplo

```
public class Exercise {
    public static void main(String[] args) {
        BiPredicate<List<String>,String> checkIfInList = (list,value)->{
            if(list!=null && !list.isEmpty() && value!=null){
                if(list.contains(value)){
                    return true;
                }
            }
            return false;
        };

        List<String> fruits = new ArrayList<String>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");

        Boolean isInList = checkIfInList.test(fruits, "Water Melon");
        System.out.println("Is Water Melon present in the List ? : "+isInList);

        System.out.println("Is Grapes present in the List ? : "+checkIfInList.test(fruits, "Grapes"));
        System.out.println("Is Apple present in the List ? : "+checkIfInList.test(fruits, "Apple"));
    }
}
```

EXAMEN!!

Ingresa en el siguiente enlace y responde según consideres correcto:

<https://tinyurl.com/j8examen2>

3

Operaciones Lambda



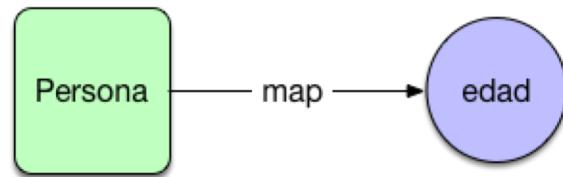
3.1 Map



Operaciones Lambda

Map

El uso de Java Stream map es una de las operaciones más comunes cuando trabajamos con un flujo de Streams . El método map nos permite realizar una transformación rápida de los datos y muy directa sobre el flujo original.



Operaciones Lambda

Map

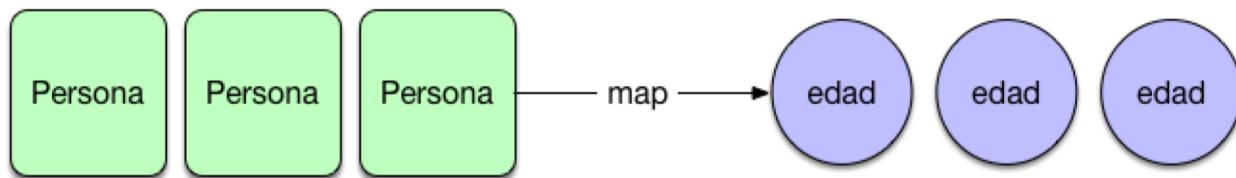
Veamos algunos ejemplos que nos ayuden a clarificar cómo se utiliza Java Stream map. Para ello nos vamos a apoyar en una lista de Personas:

```
Persona p1= new Persona("pedro",20,"perez");
Persona p2= new Persona("juan",25,"perez");
Persona p3= new Persona("ana",30,"perez");
List<Persona> lista= new ArrayList<Persona>();
lista.add(p1);
lista.add(p2);
lista.add(p3);
```

Operaciones Lambda

Map

La primera operación que vamos a realizar es convertir nuestra lista de personas a una lista de números enteros.



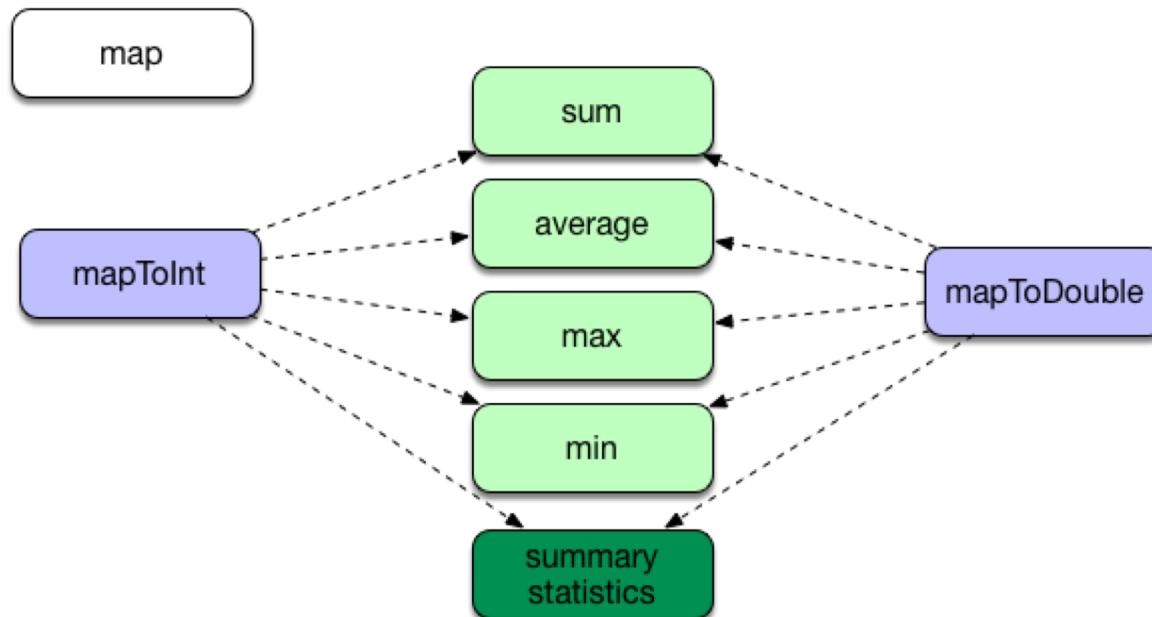
Operaciones Lambda

Map

El método map viene con dos métodos adicionales orientados a trabajar con datos numéricos. Estos métodos son mapToInt y mapToDouble. Si cambiamos nuestro método de map a mapToInt o mapToDouble se nos abrirá la posibilidad de acceder a métodos adicionales muy orientados a estadísticas.

Operaciones Lambda

Map



Operaciones Lambda

Map

```
int total=lista.stream().mapToInt(Persona::getEdad).sum();
System.out.println(total);
lista.stream().mapToDouble(Persona::getEdad).average().ifPresent(System.out::println);
lista.stream().mapToInt(Persona::getEdad).max().ifPresent(System.out::println);
lista.stream().mapToInt(Persona::getEdad).min();
```

Operaciones Lambda

Map

Hay situaciones en las que podemos querer acceder de forma directa a todas las estadísticas. Los streams numéricos soportan el método de `summaryStatistics` que nos permite acceder directamente a todos los valores.

`DoubleSummaryStatistics`

```
estadisticas=list.stream().mapToDouble(Persona::getEdad).summaryStatistics();
```

```
System.out.println(estadisticas.getAverage());
```

3.2

Tipos de operaciones con Streams



Tipos de operaciones con Streams

Las operaciones principales de streams se dividen en intermedias (intermediate operations) y terminales (terminal operations) según sus características.

Las operaciones intermedias se ejecutan siempre de manera lazy ,es decir, que en caso de ejecutarse una operación de búsqueda que por ejemplo quisiera encontrar el primer número primo de una colección, esta no debería examinar todos los valores. Al ejecutarse, su resultado es un nuevo stream con el resultado de la operación aplicada.

Tipos de operaciones con Streams

Las operaciones terminales, a diferencia de las intermedias, debe pasar por todos los elementos del stream para generar un resultado. Cuando se ejecuta una operación de este tipo, el stream se da por concluido/cerrado y no se pueden realizar más funciones de cualquier tipo (terminales o no). Se considera este tipo de operaciones eager, por su característica de ejecutarse antes de retornar el resultado.

Tipos de operaciones con Streams

Ejemplo: partamos de un listado de vehículos:

```
List<Vehiculo> vehiculos = new ArrayList<>();
    vehiculos.add(new Vehiculo("1111AAA", "BMW", 10000));
    vehiculos.add(new Vehiculo("2222BBB", "AUDI", 20000));
    vehiculos.add(new Vehiculo("3333CCC", "OPEL", 30000));
    vehiculos.add(new Vehiculo("4444DDD", "BMW", 100000));
    vehiculos.add(new Vehiculo("5555EEE", "AUDI", 200000));
    vehiculos.add(new Vehiculo("6666FFF", "OPEL", 300000));
    vehiculos.add(new Vehiculo("7777GGG", "CITROEN", 0));
```

Tipos de operaciones con Streams

Operaciones intermedias:

- ◆ Map: Altera los elementos en base a una función y los devuelve alterados. También permite realizar proyecciones de atributos.

Tipos de operaciones con Streams

Operaciones intermedias:

```
// Alteración de kilómetros
List<Integer> result = vehiculos.stream()
    .map(v -> v.getKilometros()*2)
    .collect(Collectors.toList());
// Output
[20000, 40000, 60000, 200000, 400000, 600000, 0]
// Proyección del campo kilómetros
List<Integer> result2 = vehiculos.stream()
    .map(Vehiculo::getKilometros)
    .collect(Collectors.toList());
// Output
[10000, 20000, 30000, 100000, 200000, 300000, 0]
```

Tipos de operaciones con Streams

Operaciones intermedias:

- ◆ **Filter:** Filtra los elementos a partir del la condición de la función pasada como parámetro.

```
List<Vehiculo> result = vehiculos.stream()
    .filter(v -> v.getModelo().equals("AUDI"))
    .collect(Collectors.toList());
    result.forEach(item->System.out.println(item.getPlaca()+
    "+item.getModelo()", "+item.getKilometros()));
```

```
// Output
[[ 2222BBB, AUDI, 20000 ], [ 5555EEE, AUDI, 200000 ]]
```

Tipos de operaciones con Streams

Operaciones intermedias:

- ◆ **Sorted:** Ordena los elementos siguiendo la condición de la función como parámetro.

```
List<Vehiculo> result = vehiculos.stream()
        .sorted((v1, v2) -> Integer.compare(v1.getKilometros(),
v2.getKilometros()))
        .collect(Collectors.toList());
    result.forEach(item->System.out.println(item.getPlaca()+
"+item.getModelo()"+", "+item.getKilometros()));
```

// Output (ordenado por kilómetros)

```
[  
[ 7777GGG, CITROEN, 0 ], [ 1111AAA, BMW, 10000 ],  
[ 2222BBB, AUDI, 20000 ], [ 3333CCC, OPEL, 30000 ],  
[ 4444DDD, BMW, 100000 ], [ 5555EEE, AUDI, 200000 ],  
[ 6666FFF, OPEL, 300000 ]
```

Tipos de operaciones con Streams

Operaciones intermedias:

- ◆ **Distinct:** Retorna un stream con elementos no repetidos basándose en la comparación entre objetos con la función equals (Object.equals(object)).

Tipos de operaciones con Streams

Operaciones intermedias:

```
// Distintos según modelo
List<String> result = vehiculos.stream()
    .map(Vehiculo::getModelo)
    .distinct()
    .collect(Collectors.toList());
result.forEach(item->System.out.println(item));
```

```
// Output
[BMW, AUDI, OPEL, CITROEN]
```

Tipos de operaciones con Streams

Operaciones intermedias:

- ◆ **Peek:** Esta operación no realiza cambios sobre el stream y lo devuelve tal y como entra. El propósito principal consiste en hacer debugging al ejecutar cualquier otra operación, ya que permite imprimir valores de los elementos del stream.

Tipos de operaciones con Streams

Operaciones intermedias:

```
List<String> result = vehiculos.stream()
    .filter(v -> v.getKilometros() > 20000)
    .peek(v -> System.out.println(v.getPlaca() + ", " + v.getModelo() + ", " + v.getKilometros()))
    .map(Vehiculo::getPlaca)
    .peek(v -> System.out.println(v))
    .collect(Collectors.toList());

// Output
[ 3333CCC, OPEL, 30000 ]
3333CCC
[ 4444DDD, BMW, 100000 ]
4444DDD
[ 5555EEE, AUDI, 200000 ]
5555EEE
[ 6666FFF, OPEL, 300000 ]
6666FFF
```

Tipos de operaciones con Streams

Operaciones intermedias:

- ◆ **Limit:** Limita el número de elementos que tiene como salida el nuevo stream.

```
List<Vehiculo> result = vehiculos.stream()
    .limit(2)
    .collect(Collectors.toList());
result.forEach(item->System.out.println(item.getPlaca(),
    "+item.getModelo()", "+item.getKilometros()));
```

```
// Output
[[ 1111AAA, BMW, 10000 ], [ 2222BBB, AUDI, 20000 ]]
```

Tipos de operaciones con Streams

Operaciones terminales:

- ◆ **forEach:** Realiza la acción de la función por parámetro para cada elemento

```
vehiculos.parallelStream()
    .limit(5)
    .forEach(item->System.out.println(item.getPlaca()+
        "+item.getModelo()"+
        "+item.getKilometros()));
```

```
// Output
[ 2222BBB, AUDI, 20000 ]
[ 3333CCC, OPEL, 30000 ]
[ 4444DDD, BMW, 100000 ]
[ 1111AAA, BMW, 10000 ]
[ 5555EEE, AUDI, 200000 ]
```

Tipos de operaciones con Streams

Operaciones terminales:

- ◆ **Collect:** Ejecuta lo que denominan mutable reduction, que consiste en acumular los resultados en una Collection a medida que los va procesando en la pipeline del stream.

```
List<Vehiculo> result = vehiculos.parallelStream()
    .limit(5)
    .collect(Collectors.toList());
result.forEach(item->System.out.println(item.getPlaca()+" "+item.getModelo()+" "+item.getKilometros()));

// Output
[
  [ 1111AAA, BMW, 10000 ],
  [ 2222BBB, AUDI, 20000 ],
  [ 3333CCC, OPEL, 30000 ],
  [ 4444DDD, BMW, 100000 ],
  [ 5555EEE, AUDI, 200000 ]
]
```

Tipos de operaciones con Streams

Operaciones terminales:

- ◆ **Reduce:** Permite hacer una reducción de los resultados del stream, lo cual consiste en acumular el resultado en un resultado resumido de la entrada, por ejemplo, encontrar la suma de un stream de enteros. También hay otras operaciones que utilizan la operación reduce en background y son terminales, como sum().

```
// Suma de todos los kilómetros de todos los vehículos
Integer result = vehículos.stream()
    .map(v -> v.getKilometros())
    .reduce(0, Integer::sum);
```

Examen!!

Ingresa en el siguiente enlace y responde según creas correcto:

https://docs.google.com/forms/d/13-BPjiARMAuBhV3rmkIXRmMk7vVnfKxGi8aS5_ckar0/edit

3.3

Clase Optional



Optional

Muchos de nosotros hemos sufrido alguna Null Pointer Exception, a veces en partes del código donde parece imposible que sucedan.

Para corregir estos errores, algunos lenguajes han decidido eliminar por completo los temidos **null**, pero para aquellos lenguajes que en su momento lo incluyeron, no es tan fácil. De ahí la existencia de alternativas como el patrón Option, el cual nos permite mostrar de manera explícita (mediante el sistema de tipos) la posibilidad de que un método pueda no devolver el valor deseado. Esto nos obliga a controlar la posible ausencia de valor de manera explícita, permitiéndonos elegir un valor alternativo en caso de dicha ausencia o simplemente realizar otra acción.

Optional

En Java las **Exception** pueden usarse para avisar de un error inesperado en un método. Estas, pueden ser de dos tipos, checked y unchecked, las checked exceptions obligan a quien llame a la exception a capturarla, sin embargo las unchecked pueden ser lanzadas sin avisarlo previamente en la signatura del método. Esto puede provocar errores silenciosos, y no deberían ser usadas excepto para errores insalvables.

Optional

El problema de las checked exceptions es que son bastante pesadas de utilizar, además de que un método puede lanzar varias dependiendo del problema, y acaban convirtiéndose más en un problema que en una solución.

Optional sin embargo parece una solución más ligera, sobre todo usado junto a otros patrones importados de la programación funcional

Optional

En Java 8 este patrón se encapsula en la clase Optional, la cual incluye muchos de los métodos necesarios para trabajar con este patrón. Vamos a hacer un breve repaso de la clase

```
public final class Optional<T>{}
```

Lo más importante es la signatura de la clase, en la cual podemos ver que es una clase genérica que nos permite que el objeto que contenga (o no) sea de cualquier clase.

Optional

El siguiente punto que vamos a ver, es cómo crear la clase, Optional tiene un constructor privado, y nos proporciona tres métodos factoría estáticos para instanciar la clase.

```
public static<T> Optional<T> empty()  
public static <T> Optional<T> ofNullable(T value)  
public static <T> Optional<T> of(T value)
```

Siendo el método **.of** el que nos permite recubrir cualquier objeto en un optional.

Optional

```
String nombre = "Daniel";  
Optional<String> oNombre = Optional.of(nombre);
```

Los otros dos métodos nos permiten recubrir un valor nulo o devolver un objeto Optional vacío en caso de que queramos avisar de la ausencia de valor.

La opción de recubrir un nulo viene dada principalmente para permitirnos trabajar con APIs que hacen uso de nulos para avisar de estas ausencias de valor.

Optional

```
public boolean isPresent()  
public T get()
```

El método `isPresent` es el equivalente a variable `== null` y cómo el propio nombre indica nos dice si el objeto `Optional` contiene un valor o está vacío.

Este método se usa principalmente si trabajamos de manera imperativa con `Optional`.

Optional

Y el método `get` es el encargado de devolvernos el valor, devolviendo una excepción si no estuviera presente.

Optional

Ejemplo:

```
private static void isPresentEjemplo() {  
    Optional<String> str = Optional.of("Pedro");  
    if(str.isPresent()) {  
        System.out.println(str.get());  
    }  
}
```

Optional

```
public Optional<T> filter(Function f)
public<U> Optional<U> map(Function f)
public<U> Optional<U> flatMap(Function f)
```

estos tres métodos hacen que trabajar con Optional sea verdaderamente interesante, nos da la posibilidad de encadenar distintas operaciones que devuelvan Optional sin tener que estar comprobando si el valor está presente después de cada operación.

Optional

Ejemplo:

```
private static void filterEjemplo() {  
    Optional<Car> cocheVacio = Optional.empty();  
    cocheVacio.filter(x->"250".equals(x.getPrecio())).ifPresent(x-  
>System.out.println(x.getPrecio() + " coche vacio"));  
  
    //Si el valor no pasa el filtro  
    Optional<Car> cocheMuyCaro = Optional.of(new Car("3000"));  
    cocheMuyCaro.filter(x->"250".equals(x.getPrecio())).ifPresent(x-  
>System.out.println(x.getPrecio() + " coche muy caro"));  
  
    //Si pasa el filtro y tiene valores  
    Optional<Car> cocheAceptable = Optional.of(new Car("250"));  
    cocheAceptable.filter(x->"250".equals(x.getPrecio())).ifPresent(x-  
>System.out.println(x.getPrecio() + " coche aceptable"));  
  
}
```

Optional

Ejemplo:

```
private static void ejemploMap() {  
    Optional<String> str = Optional.of("lorem ipsum lo que sigue en la cadena de texto en  
latín");  
    Optional<Integer> sizeOptional = str.map(String::length);  
  
    System.out.println("el tamaño del string " + sizeOptional.orElse(0));  
  
    //else  
    Optional<String> strNull = Optional.ofNullable(null);  
    Optional<Integer> sizeNull = strNull.map(x -> x.length());  
  
    System.out.println("El tamaño del string: "+sizeNull.orElse(0));  
}
```

Optional

Ejemplo flatmap:

```
List<String> myList = Stream.of("a", "b")  
    .map(String::toUpperCase).collect(Collectors.toList());  
    System.out.println(myList);
```

map() funciona bastante bien en casos simples, pero ¿qué pasa si tenemos algo más complejo como una lista de listas como entrada?

```
List<List<String>> list = Arrays.asList(  
    Arrays.asList("a"),  
    Arrays.asList("b"));  
  
System.out.println(list  
    .stream()  
    .flatMap(Collection::stream)  
    .collect(Collectors.toList()));
```

Optional

```
public T orElse(T other)
```

```
public T orElseGet(Supplier<? extends T> other)
```

```
public <X extends Throwable> T orElseThrow(Function f)
```

Finalmente estos métodos nos permiten finalizar una serie de operaciones, teniendo tres maneras:

La primera **orElse** nos devuelve el valor o si no devolverá el valor que le demos.

Optional

Ejemplo:

```
private static voidorElseEjemplo() {  
    Car coche1 = new Car("200");  
    Car cochePorDefecto = new Car("500");  
  
    Optional<Car> optionalCoche = Optional.of(coche1);  
    String precio = optionalCoche.orElse(cochePorDefecto).getPrecio();  
    System.out.println("El precio del coche es: "+precio);  
  
    //else  
    Optional<Car> coche2 = Optional.empty();  
    precio = coche2.orElse(cochePorDefecto).getPrecio();  
    System.out.println("El precio del coche es: "+precio);  
}
```

Optional

Ejemplo:

```
private static void ejemploThrow() {  
    try {  
        Car nulo = null;  
        Optional<Car> optionalCarNulo = Optional.ofNullable(nulo);  
        optionalCarNulo.orElseThrow(IllegalStateException::new);  
    }catch(IllegalStateException ex) {  
        System.out.println("El coche tiene valor nulo");  
    }  
}
```

Optional

orElseGet nos devolverá el valor si está presente y si no, invocará la función que le pasemos por parámetro y devolverá el resultado de dicha función.

Y finalmente **orElseThrow** nos devolverá el valor si está presente y sino invocará la función que le pasemos, la cual tiene que devolver una excepción y lanzará dicha excepción. Esto nos ayudará a trabajar en conjunto con APIs que todavía usen excepciones.

Optional

Usando optional de forma imperativa

Pongamos el caso siguiente, tenemos un método que obtiene un disco a partir de un nombre, puede darse el caso de que no se encuentre ningún disco con ese nombre, sin Optional tendríamos dos opciones:

- Devolver null en caso de que no encontrásemos el disco.
- Lanzar una excepción indicando que no se ha encontrado el disco.

Optional

Usando optional de forma imperativa

Con Optional se nos abre la tercera opción:

```
public Optional<Album> getAlbum(String artistName)
```

Optional

Usando optional de forma imperativa

A la hora de usar este método, de la manera imperativa haríamos lo siguiente.

```
Album album;  
Optional<Album> albumOptional = getAlbum("Random Memory Access");  
if(albumOptional.isPresent()){  
    album = albumOptional.get();  
}else{  
    // Avisar al usuario de que no se ha encontrado el album  
}
```

Optional

Usando optional de forma imperativa

Esto ya es un avance respecto a los null, ya que estamos indicando explícitamente al usuario de la API de que es posible que no se encuentre el álbum y de que es necesario que actúe en caso de error.

Optional

- Se encuentra en el **java.util.package**.
- Se utiliza para representar si un valor está presente o no.
- La mayor ventaja de este método es que no se requiere realizar más chequeos de nulos.
- No se requiere evaluación de **NullPointerException**.
- Nos permite el desarrollo de API's y Aplicaciones claras y sencillas.

Optional

- Para emplear Optional debemos tomar en cuenta lo siguiente:
 - Sus instancias son inmutables (aunque pueden tener referencias a objetos mutables).
 - No tiene un constructor público, por lo que debemos instanciarlas por medio de un factory
 - Implementa los métodos equals, hashCode y toString

EXAMEN!!

Ingresa en el siguiente enlace y responde según creas correcto:

<https://tinyurl.com/j8examen4>

3.4

Corte de un Stream



Corte de un Stream

Si bien los Streams facilitan la lectura y escritura del código para operar sobre colecciones y listas dándonos una aproximación más funcional, los bucles, por ejemplo, siguen teniendo sus ventajas, un caso palpable de esto es cuando queremos detener la ejecución de un forEach.

Si estamos ejecutando un forEach regular, podemos llamar a un break en lo que se cumpla cierta condición y éste se detendrá, pero en los Streams no podemos hacer esto.

Corte de un Stream

En cambio, tenemos a disposición otros métodos que funcionan para iterar sobre la colección pero que también nos permiten detener esta iteración al cumplirse una condición X.

Ejemplo de ello son los métodos `anyMatch()` y `findFirst()`

Corte de un Stream

Por ejemplo:

```
Optional<SomeObject> result =  
    someObjects.stream().filter(obj -> some_condition_met).findFirst();
```

Este método retorna el primer objeto de la colección que coincide con la condición dada.

Corte de un Stream

Por ejemplo:

```
boolean result = someObjects.stream().anyMatch(obj ->  
some_condition_met);
```

Este método retorna un booleano, por ejemplo, si encuentra algún registro en la colección que cumpla con la condición dada retornará true y detendrá la iteración.

3.5

Agrupar datos a través de collectors



Agrupar con Collectors

Cuento más usamos los streams más usamos los diferentes Java Stream Collectors para transformar estos. Vamos a ver cuales son las diferentes opciones que Java soporta para transformar un stream a otro tipo de estructura.

Veamos algunos ejemplos partiendo de una clase base.

Agrupar con Collectors

```
public class Libros {  
  
    private String titulo;  
    private int paginas;  
  
    public Libros(String titulo, int paginas) {  
        super();  
        this.titulo = titulo;  
        this.paginas = paginas;  
    }  
  
    getters.....  
  
    setters....  
  
}
```

Agrupar con Collectors

```
public class Principal {  
  
    static Libros l1= new Libros("El señor de los anillos",1000);  
    static Libros l2= new Libros("La fundacion",500);  
    static Libros l3= new Libros("El caliz de fuego",600);  
  
    public static void main(String[] args) {  
  
    }  
}
```

Agrupar con Collectors

En muchas ocasiones no queremos simplemente imprimir un resultado sino que lo que queremos es convertir el Stream a un tipo de dato para su posterior gestión .



Agrupar con Collectors

Vamos a ver algunas opciones, una de las más sencillas es convertirlo a un array con el método `toArray()`:

```
Stream<Libros> libros = Stream.of(l1,l2,l3);
Libros[] arrayLibro= libros.toArray(Libros[]::new);

for(int i=0;i<arrayLibro.length;i++) {
    System.out.println(arrayLibro[i].getPaginas());
}
```

Agrupar con Collectors

De igual forma que podemos convertir el Stream a un array, podemos usar los Java Stream Collectors y convertir nuestro stream a una Lista o Conjunto , utilizando la clase Collectors y su método `toList()` o `toSet()`.

```
Stream<Libros> libros = Stream.of(l1,l2,l3);
List<Libros> lista= libros.collect(Collectors.toList());

for (Libros l:lista) {
    System.out.println(l.getTitulo());
}
```

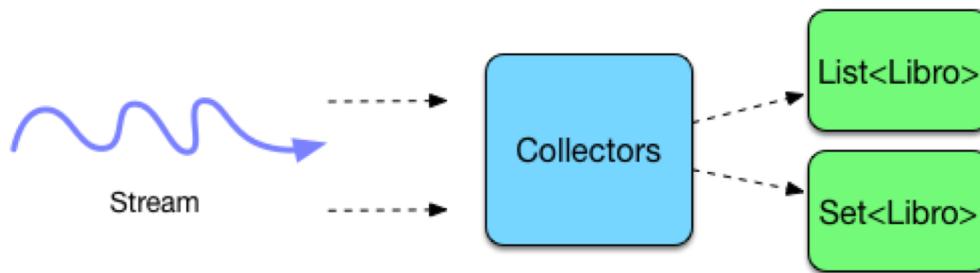
Agrupar con Collectors

```
Stream<Libros> libros = Stream.of(l1,l2,l3);
Set<Libros> lista= libros.collect(Collectors.toSet());

for (Libros l:lista) {
    System.out.println(l.getTitulo());
}
```

Agrupar con Collectors

Acabamos de usar dos de los métodos fundamentales de Collectors para generar Sets y List a partir de un Stream.



Agrupar con Collectors

Los Java Stream collectors nos permiten realizar operaciones más complejas con poco código . Por ejemplo podemos obligar a que todos los títulos de los Libros se impriman en una única línea usando el método joining de la clase Collectors.

```
Stream<Libros> libros = Stream.of(l1,l2,l3);
String resultado = libros.map((l) -> l.getTitulo()).collect(Collectors.joining(","));
System.out.println(resultado);
```

Agrupar con Collectors

Podemos complicarlo más y sumar todas las páginas de los libros utilizando el método reducing de Collectors.

```
Optional<Integer> resultado3 = st.map((l) ->  
    l.getPaginas().collect(Collectors.reducing(Integer::sum));
```

```
System.out.println(resultado3.get());
```

EXAMEN!!

Ingresa en el enlace y responde según creas correcto:

<http://tinyurl.com/j8examen5>

4

Concurrencia



Concurrencia

- Es la habilidad de ejecutar múltiples programas o múltiples parte de un programa en paralelo.
- Los ordenadores de hoy en día tienen varios CPU's y cada CPU tiene varios núcleos, por ello la habilidad de emplear este sistema de múltiples-núcleos es la clave para el éxito de las aplicaciones que requieren volumen intensivo de operaciones y datos.

Concurrencia - Proceso vs. Hilo

- Un PROCESO:
 - ✓ Se ejecuta independientemente y de forma aislada de otros procesos.
 - ✓ No puede acceder directamente a data compartida por otros procesos.
 - ✓ Los recursos para la ejecución del proceso (CPU, memoria, etc.) son asignados por el sistema operativo.
- Un HILO (conocido como proceso ligero):
 - ✓ Tiene su propio entorno, pero puede acceder data compartida por otros hilos en el mismo proceso.
 - ✓ Cada hilo tiene su propio cache de memoria.
 - ✓ Si un hilo lee data compartida por otro, almacena dicha data en su propio cache de memoria.

Concurrencia - ¿Y en Java?

- Un programa Java siempre se ejecuta en su propio proceso y por defecto en un solo hilo.
- Java soporta los hilos como parte del lenguaje mediante la clase **Thread**.
- Una aplicación Java puede crear nuevo hilos mediante la clase Thread.
- También podemos emplear las clases del paquete **java.util.concurrent** para poder gestionar la concurrencia.

4.1

Java.util.concurrent



Java.util.concurrent

El paquete util.concurrent nos permite crear aplicaciones concurrentes, para ello, el paquete nos provee de varias utilidades, entre las que destacan:

- ◆ Executor
- ◆ ExecutorService
- ◆ ScheduledExecutorService
- ◆ Future

Java.util.concurrent

- ◆ El Concurrency API introduce el concepto de un ExecutorService como un sustituto de mayor nivel para trabajar con hilos directamente.
- ◆ Los ejecutores son capaces de ejecutar tareas asíncronas y normalmente gestionan un grupo de subprocessos, por lo que no tenemos que crear nuevos subprocessos manualmente.
- ◆ Todos los hilos del pool interno se reutilizarán para tareas costosas, por lo que podemos ejecutar tantas tareas simultáneas como queramos a lo largo del ciclo de vida de nuestra aplicación con un único servicio executor.

Java.util.concurrent

Executor

Es una interfaz que representa un objeto que ejecuta una tarea especificada.

Dependiendo de su uso, puede ejecutarse la tarea en cuestión en un hilo en ejecución o en un nuevo hilo, sin embargo, su uso no se limita a tareas asíncronas.

Java.util.concurrent

Executor

Primero, creamos un invocador:

```
public class Invoker implements Executor {  
    @Override  
    public void execute(Runnable r) {  
        r.run();  
    }  
}
```

Java.util.concurrent

Executor

Y ejecutamos la tarea a través del mismo:

```
public void execute() {  
  
    Executor executor = new Invoker();  
  
    executor.execute( () -> {  
  
        // tarea a realizar  
  
    });  
  
}
```

ExecutorService

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});
```

Al ejecutar el código notaremos una diferencia importante: ¡el proceso java nunca se detiene! Los ejecutores tienen que ser detenidos explícitamente - de lo contrario, siguen escuchando las nuevas tareas.

ExecutorService

ExecutorService proporciona dos métodos para ese propósito: **shutdown()** que espera a que las tareas que se están ejecutando actualmente terminen y **shutdownNow()** interrumpe todas las tareas que se están ejecutando y apaga al ejecutor inmediatamente.

Veamos una mejora del ejemplo anterior **añadiendo** lo siguiente:

```
try {
    System.out.println("Intento de detener el executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

Java.util.concurrent

ExecutorService

Es una solución completa para servicios asíncronos, se usa a través de la clase Runnable:

```
public class Tarea implements Runnable {
    private String nombre;
    public Tarea(String nombre) {
        this.nombre = nombre;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(nombre);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Java.util.concurrent

```
public class Principal {  
    public static void main(String[] args) {  
        Thread t1= new Thread(new Tarea("tarea1"));  
        t1.start();  
        Thread t2= new Thread(new Tarea("tarea2"));  
        t2.start();  
        Thread t3= new Thread(new Tarea("tarea3"));  
        t3.start();  
    }  
}
```

Java.util.concurrent

```
public class PrincipalExecutor {  
    public static void main(String[] args) {  
        Stream<String> flujo = Stream.of("tarea1", "tarea2", "tarea3");  
        ExecutorService servicio = Executors.newCachedThreadPool();  
        flujo.map(t->new Tarea(t)).forEach(servicio::execute);  
    }  
}
```

El resultado será identico pero tendrá varias ventajas. En primer lugar la reducción de código y la flexibilidad que tendríamos si utilizáramos un stream infinito.

En segundo lugar al no ser nosotros los que inicializamos los Threads dejamos al API de Java que se encargue de hacerlo de forma correcta.

Java.util.concurrent

ScheduledExecutorService

Es una interfaz similar al ExecutorService pero con la capacidad de ejecutar tareas de forma periódica.

Junto con esta interfaz pueden usarse las interfaces Runnable o Callable sin problemas.

newSingleThreadScheduledExecutor

```
ScheduledExecutorService executor = Executors.  
    → newSingleThreadScheduledExecutor();  
  
ScheduledFuture future = executor.schedule(  
    new Callable<String>() {  
  
        @Override  
        public String call() throws Exception {  
            return "Ya pasaron 10 segundos.";  
        }  
    }, 10, TimeUnit.SECONDS);  
  
System.out.println(future.get()); ←  
  
executor.shutdown(); ←
```

ScheduledExecutorService

- ◆ **newSingleThreadScheduledExecutor**: Crea un ejecutor de un solo hilo que puede programar comandos para que se ejecuten después de un delay dado o para que se ejecuten periódicamente.
- ◆ **ScheduledFuture**: Representación futura de la ejecución del thread.
- ◆ **schedule**: Recibe tres parámetros, el primero un objeto Callable el cual contiene el proceso a ejecutar por el Thread. El segundo parámetro es un entero que representara el tiempo a esperar para la ejecución. Por último, la unidad de tiempo a medir utilizando la enumeración TimeUnit
- ◆ **future.get()**: obtiene el resultado de la ejecución del thread
- ◆ **shutdown**: Inicia un apagado ordenado en el que se ejecutan las tareas enviadas previamente, y no se aceptarán nuevas tareas.

Java.util.concurrent

```
ScheduledExecutorService execService = Executors.newScheduledThreadPool(5);
execService.scheduleAtFixedRate(()->{
    System.out.println("hi there at: " + new java.util.Date());
}, 0, 1000L, TimeUnit.MILLISECONDS);
```

Maneja dos tiempos, el de espera para la primera ejecución y el tiempo de ejecuciones sucesivas luego de ocurrida la primera

scheduleAtFixedRate

- ◆ Crea y ejecuta una acción periódica que se activa **después del retardo inicial** dado, y posteriormente con el período dado; es decir, las ejecuciones comenzarán después de initialDelay luego initialDelay+period, luego initialDelay + 2 * period, y así sucesivamente.
- ◆ Si alguna ejecución de la tarea encuentra una excepción, **se suprimen las ejecuciones posteriores**. De lo contrario, la tarea sólo terminará a través de la cancelación
- ◆ Si la ejecución de esta tarea lleva más tiempo que su período, las ejecuciones posteriores pueden comenzar tarde, pero **no se ejecutarán simultáneamente**.

CompletableFuture

El **CompletableFuture** es un gran paso para java y la forma en como abstraemos la programacion asíncrona.

Supongamos que tenemos una app que consulta cierta información de 2 servicios diferentes, un servicio responde muy rápido con resultado impreciso, mientras que el otro se tarda un poco mas pero el resultado siempre es predecible, sea cual sea el caso nos interesa tener el valor. Asi que utilizamos CompletableFuture para hacer un **comportamiento “race”** entre ambas consultas

CompletableFuture

```
CompletableFuture<String> rapido = consultaRapido();
CompletableFuture<String> predecible = consultaPredecible();
rapido.acceptEither(predecible, s -> {
    System.out.println("Resultado: " + s);
});
```

Esto nos devolverá el CompletableFuture que termine primero.

CompletableFuture

El API de la clase CompletableFuture, nos proporciona muchos métodos, casi todos ellos en tres “versiones”. Por ejemplo, para el método ‘thenAccept’ tenemos:

```
CompletableFuture<Void> thenAccept(Consumer<? super T> action);
```

```
CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action);
```

```
CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action, Executor executor);
```

- La versión sin “Async” ejecutará la función lambda en el mismo thread que el que la llama.
- La versión con “Async” ejecutará la ejecutara en un thread nuevo, usando para ello el Executor por defecto.
- La versión con “Async” y con parámetro ExecutorService, la ejecutará en un thread nuevo usando el Executor pasado como parámetro.

CompletableFuture

```
CompletableFuture cf = CompletableFuture.completedFuture("message");
System.out.println(cf.isDone());
System.out.println(cf.getNow(null));
```

El getNow(null) devuelve el resultado si está completado (lo que obviamente es el caso), pero por lo demás devuelve null (el argumento).

CompletableFuture

El siguiente ejemplo toma el CompletableFuture completo del ejemplo anterior, que lleva la cadena de resultados "message" y aplica una función que lo convierte en mayúsculas:

```
CompletableFuture cf = CompletableFuture.completedFuture("message").thenApply(s -> {
    return s.toUpperCase();
});
if ("MESSAGE".equals(cf.getNow(null))) {
    System.out.println("OK");
}
```

CompletableFuture

Las palabras clave de comportamiento en thenApply:

then, significa que la acción de esta etapa ocurre cuando la etapa actual se completa normalmente (sin excepción). En este caso, la etapa actual ya está completada con el valor "mensaje".

apply, significa que la etapa devuelta aplicará una Función sobre el resultado de la etapa anterior.

La ejecución de la función se bloqueará, lo que significa que getNow() sólo se alcanzará cuando se realice la operación en mayúsculas.

CompletableFuture

Si la siguiente etapa acepta el resultado de la etapa actual pero no necesita devolver un valor en el cálculo (es decir, su tipo de retorno es nulo), entonces en lugar de aplicar una Función, puede aceptar un Consumidor, de ahí el método `thenAccept`:

```
StringBuilder result = new StringBuilder();
CompletableFuture.completedFuture("thenAccept message").thenAccept(s -> result.append(s));
    if (result.length() > 0) {
        System.out.println("Result was empty");
    }
```

El Consumidor será ejecutado sincrónicamente, así que no necesitamos unirnos en el CompletableFuture devuelto.

4.2

Los métodos Runnable y Callable



Runnable y Callable

Entendiendo la importancia de la concurrencia y el uso de multihilos en Java para mejorar el rendimiento de nuestras aplicaciones y la experiencia del usuario final, podemos ver las interfaces que nos ofrece el lenguaje para el manejo de esta operativa.

En este caso hablamos de Runnable, la interfaz core para representar tareas multi-hilos y Callable

Runnable y Callable

Ambas interfaces sirven para representar tareas que pueden ser ejecutadas bajo múltiples hilos, la diferencia radica en que la interfaz Runnable puede ser ejecutada usando la clase Thread o la clase ExecutorService, mientras que Callable solo se ejecuta bajo la clase ExecutorService.

Los callables son interfaces funcionales al igual que los runnables, pero en lugar de ser nulos, devuelven un valor.

Runnable y Callable

Runnable es una interfaz funcional que contiene un único método **run()** que no acepta parámetros ni retorna ningún valor, por esto, suele usarse para operaciones donde no necesitemos resultados retornados como, por ejemplo, el log de una aplicación en ejecución.

Por ejemplo:

Runnable y Callable

```
public class EventLoggingTask implements Runnable{
    private static final Logger LoggerFactory = null;
        private Logger logger = LoggerFactory.getAnonymousLogger();

    @Override
    public void run() {
        logger.info("Mensaje de log");
    }

}
```

Runnable y Callable

```
public void executeTask() {  
    executorService = Executors.newSingleThreadExecutor();  
    Future future = executorService.submit(new EventLoggingTask());  
    executorService.shutdown();  
}
```

En el ejemplo anterior, se envían mensajes de una cola a un archivo log, la tarea no retorna nada y es ejecutada a través del ExecutorService.

Runnable y Callable

En el caso de Callable, nos encontramos con una interfaz genérica que contiene un método call() y retorna un valor genérico V.

Runnable y Callable

```
import java.util.concurrent.Callable;

public class MiCallable implements Callable<Integer> {

    @Override
    public Integer call() throws Exception {

        int total = 0;
        for(int i=0;i<5;i++) {

            total+=i;
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName());

        return total;
    }
}
```

Runnable y Callable

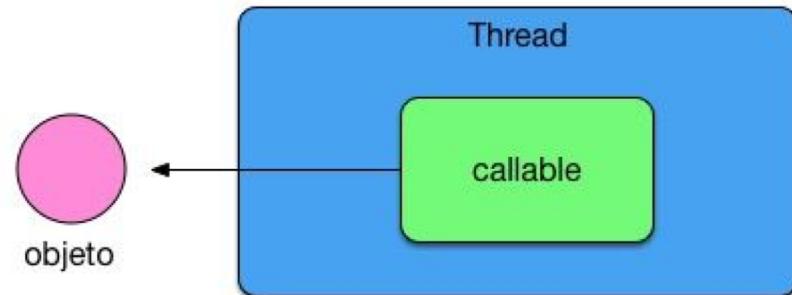
Acabamos de crear una clase que implemente Java Callable interface. Es momento de usarla desde un método main.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class PrincipalCallable {
    public static void main(String[] args) {
        try {
            ExecutorService servicio= Executors.newFixedThreadPool(1);

            Future<Integer> resultado= servicio.submit(new MiCallable());
            if(resultado.isDone()) {
                System.out.println(resultado.get());
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ExecutionException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

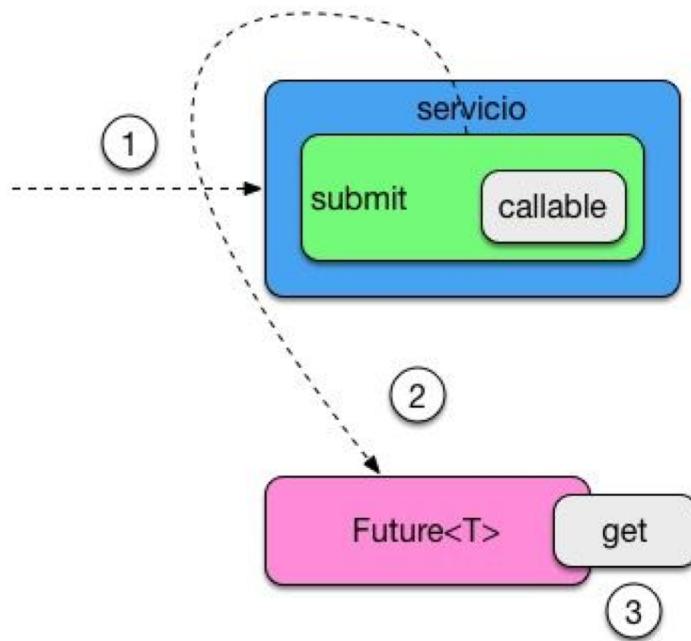
Runnable y Callable

En este caso hemos usado ExecutorService para crear un pool de Thread con un único hilo y enviar la tarea al pool utilizando el método submit.



Runnable y Callable

Cuando invoquemos el servicio recibiremos de forma automática una variable de tipo Future la cual recibirá un futuro valor 10 que podremos imprimir usando el método get().



4.3

Posibles problemas al usar hilos



Possibles problemas al usar hilos

- ◆ Comportamiento errático al no sincronizar tiempos entre tareas dependientes
- ◆ Embasuramiento por falta de limpieza de variables o terminado de hilos ya ejecutados
- ◆ Deadlocks causados por dos hilos que esperan la ejecución del otro
- ◆ Cuellos de botella causados por una serie de hilos en espera de ejecución
- ◆ Fallas de rendimiento por uso excesivo de hilos de ejecución de tareas en la aplicación

4.4

Control de ejecución de hilos



Control de ejecución de hilos

La clave para la sincronización en Java es el concepto de monitor, que controla el acceso a un objeto. Un monitor funciona implementando el concepto de bloqueo (lock). Cuando un objeto está bloqueado por un hilo, ningún otro hilo puede obtener acceso al objeto. Cuando el hilo sale, el objeto está desbloqueado y está disponible para ser utilizado por otro hilo.

Control de ejecución de hilos

Todos los objetos en Java tienen un monitor. Esta característica está integrada en el lenguaje Java en sí. Por lo tanto, todos los objetos se pueden sincronizar. La sincronización está respaldada por la palabra clave `synchronized` y algunos métodos bien definidos que tienen todos los objetos.

Como la sincronización se diseñó en Java desde el principio, es mucho más fácil de usar de lo que se podría esperar. De hecho, para muchos programas, la sincronización de objetos es casi transparente.

Control de ejecución de hilos

Puede sincronizar el acceso a un método modificándolo con la palabra clave `synchronized`. Cuando se llama a ese método, el hilo de llamada entra en el monitor del objeto, que luego bloquea el objeto.

Mientras está bloqueado, ningún otro hilo puede ingresar al método, o ingresar cualquier otro método sincronizado definido por la clase del objeto. Cuando el hilo retorna del método, el monitor desbloquea el objeto, permitiendo que sea utilizado por el siguiente hilo. Por lo tanto, la sincronización se logra con prácticamente ningún esfuerzo de programación

Control de ejecución de hilos

Ejemplo práctico

Control de ejecución de hilos

Hasta el momento podemos sincronizar de manera sencilla la ejecución de tareas dentro de clases creadas por nosotros a través de la creación de métodos synchronized, sin embargo, ¿qué pasa si quiero sincronizar métodos de librerías de terceros?

Para esto basta con llamar al método requerido dentro del bloque synchronized:

Control de ejecución de hilos

```
synchronized(objref) {  
    // declaraciones a sincronizar  
}
```

Aquí, objref es una referencia al objeto que se sincroniza. Una vez que se ha ingresado un bloque sincronizado, ningún otro hilo puede llamar a un método sincronizado en el objeto referido por objref hasta que se haya salido del bloque.

¿Por qué no se permite usar synchronized en métodos de interfaces en java 8?

Aunque nos parezca lógico que se pueda soportar el modificador synchronized en métodos por defecto, resulta que su uso en este ámbito no es muy seguro que digamos, por ello, ha sido “prohibido”.

La sincronización es sobre el bloqueo. El bloqueo trata de coordinar el acceso compartido al estado mutable. Cada objeto debe tener una política de sincronización que determine qué bloqueos guardan qué variables de estado.

Muchos objetos utilizan como política de sincronización el patrón de supervisión de Java en el que el estado de un objeto está protegido por su bloqueo intrínseco. No hay nada especial en este patrón, pero es conveniente, y el uso de la palabra clave sincronizada en los métodos asume implícitamente este patrón.

5

El framework Fork-Join



El Framework Fork-Join - Paralelismo

- El paralelismo es la ejecución simultánea de dos o más tareas. este mecanismo acelera la ejecución de una tarea dividiéndola en tareas computacionales independientes y ejecutandolas sobre hardware capaz de realizar esas tareas simultáneamente, como el caso de un procesador con varios núcleos.
- El paralelismo no es concurrencia. Suele confundirse a menudo. Ambos se basan en la programación multitarea pero no es lo mismo. Dado que la concurrencia puede existir sin hardware multitarea.

El Framework Fork-Join - Paralelismo

- ◆ El paralelismo acelera la ejecución de una tarea dividiéndola en computaciones independientes y ejecutándola sobre hardware capaz de realizar computaciones simultáneas, como por ejemplo un procesador con varios núcleos.
- ◆ Cuando ejecutamos una tarea parallelizada en múltiples ordenadores, en vez de en los múltiples cores de un solo ordenador, decimos que la computación paralela es distribuida.
- ◆ Por ejemplo, cada búsqueda en Google se ejecuta simultáneamente en cientos de ordenadores, cada uno de los cuales busca al mismo tiempo en un subconjunto del índice del web.

El Framework Fork-Join - Paralelismo

- ◆ Mientras que el paralelismo es la ejecución simultánea de dos o más tareas, la concurrencia es la ejecución de dos o más tareas en periodos de tiempo solapados, pero no necesariamente simultáneos.
- ◆ La concurrencia, es una propiedad del programa. Su computación debe dividirse en tareas independientes, cuya ejecución puede o no solaparse.
- ◆ Puede existir sin soporte hardware multitarea, por ejemplo, los sistemas operativos multitarea ejecutados en máquinas con un procesador de un solo núcleo asignan breves intervalos de tiempo a cada tarea, dando la ilusión de ejecución simultánea.

El Framework Fork-Join - ¿Qué es y cómo funciona?

- Fue creado para la ejecución de tareas que pueden dividirse en subtareas más pequeñas, ejecutándose en paralelo y combinando posteriormente los resultados para obtener el resultado de la tarea única.
- Las subtareas deberán ser independientes unas de otras y no contendrán estados.
- La parallelización se realiza de forma recursiva, aplicando el principio de “Divide y Vencerás”.
- Existe un umbral debajo del cual la tarea es indivisible, una vez alcanzado ese umbral, el rendimiento de las mismas disminuirá en caso de seguir dividiéndolas.

El Framework Fork-Join - ¿Cómo se implementa? (I)

- Se encuentra en la librería **java.util.concurrent**
- Toma forma a través de las clases:
 - ✓ ForkJoinTask
 - ✓ ForkJoinPool
 - ✓ RecursiveAction
- La clase ForkJoinPool invoca a una tarea del tipo ForkJoinTask, pero para poder invocar múltiples subtareas en paralelo de forma recursiva, invocará una tarea del tipo RecursiveAction, que extiende de ForkJoinTask.

El Framework Fork-Join - ¿Cómo se implementa? (II)

- NOTA: el método `compute` de la clase `RecursiveAction` es el encargado de ejecutar una tarea paralelizable
- Para paralelizar una tarea secuencial, creamos una clase que extienda de `RecursiveAction` y sobrecargamos el método `compute()` con dicha tarea.
- El método `compute()` creará de forma recursiva nuevas instancias de la misma clase, correspondiendo cada instancia a una subtarea distinta.

El Framework Fork-Join - ¿Cómo se implementa? (II)

Ejemplo práctico

Supongamos que queremos transformar un array largo de números. La transformación será simplemente multiplicar cada elemento en el array por un número específico.

En el siguiente ejemplo usaremos fork-join para hacer la transformación dividiendo, a través del método `compute`, las tareas necesarias para dicha transformación para ejecutarlas de forma paralela.

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
public class ArrayTransform extends RecursiveAction{
    int[] array;
    int number;
    int threshold = 100000;
    int start;
    int end;
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
public ArrayTransform(int[] array, int number, int start, int end) {  
    this.array = array;  
    this.number = number;  
    this.start = start;  
    this.end = end;  
}  
  
public void convierte() {  
    for(int i=start;i<end;i++) {  
        array[i] = array[i] * number;  
    }  
}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
protected void compute() {  
    if(end - start < threshold) {  
        convierte();  
    }else {  
        int middle = (end + start) / 2;  
        ArrayTransform subtask1 = new ArrayTransform(array,  
number, start, middle);  
        ArrayTransform subtask2 = new ArrayTransform(array,  
number, middle, end);  
  
        invokeAll(subtask1, subtask2);  
    }  
}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
public class Main {  
    static final int SIZE = 10000000;  
    static int[] array = randomArray();  
  
    static int[] randomArray() {  
        int[] array = new int[SIZE];  
        Random random = new Random();  
  
        for(int i = 0; i < SIZE; i++) {  
            array[i] = random.nextInt(100);  
        }  
        return array;  
    }  
}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
public static void main(String[] args) {
    int number = 9;

        Stream<int> stream1 = Arrays.stream(array);
stream1.limit(10).forEach(x -> System.out.println(x));

    ArrayTransform mainTask = new ArrayTransform(array, number, 0, SIZE);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(mainTask);

    stream1 = Arrays.stream(array);
stream1.limit(10).forEach(x -> System.out.println(x));

}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

Ejemplo práctico

En este caso, si retornaremos un resultado, partamos del supuesto que queremos verificar cuántas ocurrencias hay en un array.

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
public class ContadorArray extends RecursiveTask<Integer>{
    int[] array;
    int threshold = 100000;
    int start;
    int end;

    public ContadorArray(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }
}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
protected Integer calcula() {  
    Integer count = 0;  
    for(int i = start; i< end; i++) {  
        if(array[i]%2==0) {  
            count++;  
        }  
    }  
    return count;  
}  
}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
protected Integer compute() {  
    if(end-start < threshold) {  
        return calcula();  
    }else {  
        int middle = (end + start) / 2;  
  
        ContadorArray subtask1 = new ContadorArray(array,  
start, middle);  
        ContadorArray subtask2 = new ContadorArray(array,  
middle, end);  
  
        invokeAll(subtask1, subtask2);  
  
        return subtask1.join()+subtask2.join();  
    }  
}
```

El Framework Fork-Join - ¿Cómo se implementa? (II)

```
public static void main(String [] args) {  
    ContadorArray mainTask = new ContadorArray(array, 0, SIZE);  
    ForkJoinPool pool = new ForkJoinPool();  
    Integer contadorDePares = pool.invoke(mainTask);  
  
    System.out.println("El total de números pares es: " +  
contadorDePares);  
}
```

Work stealing

En la versión 5 de Java se incluyen métodos e interfaces para manejar la concurrencia de una manera sencilla a través de hilos y programación multi-hilos, todo esto a través del uso de la interfaz Executor.

Además de Executor, tenemos algunos métodos e interfaces que nos permiten delegar el control de la concurrencia a estos métodos y encargarnos solo de la lógica de negocio que, como desarrolladores, nos compete.

De esta forma, la creación, control y ciclo de vida de un hilo queda delegado a las interfaces de Java creadas para este fin.

Work stealing

En el núcleo de estas interfaces se encuentran las “pools”. Las más comunes son:

- newCachedThreadPool()
- newFixedThreadPool(int nThreads)
- newSingleThreadExecutor()
- newScheduledThreadPool(int corePoolSize)

En Java 8, aparece un nuevo tipo de pool de hilos para complementar los anteriores, se trata del newWorkStealingPool()

Work stealing

Este pool se trata de un algoritmo “work-stealing” donde una tarea puede desplazar otras tareas de menor tamaño o prioridad ejecutándose en un hilo, además, si un hilo ha terminado y no tiene tareas en cola, puede “robar” tareas de otros hilos en ejecución.

Ahora bien, esto de robar tareas de otros hilos ya era posible con ForkJoinPool que implementaba desde antes este mecanismo para reducir el tiempo de espera entre ejecuciones de hilos.

Así que veamos qué trae de nuevo este mecanismo de “work-stealing” de este ExecutorService.

Work stealing

Para verlo, veamos el código del Executors.java:

```
/**  
 * Creates a work-stealing thread pool using all  
 * {@link Runtime#availableProcessors available processors}  
 * as its target parallelism level.  
 * @return the newly created thread pool  
 * @see #newWorkStealingPool(int)  
 * @since 1.8  
 */  
public static ExecutorService newWorkStealingPool() {  
    return new ForkJoinPool(Runtime.getRuntime().availableProcessors(),  
                          ForkJoinPool.defaultForkJoinWorkerThreadFactory,  
                          null, true);  
}
```

Work stealing

Como puede verse, se está retornando un ForkJoinPool pero con parámetros predefinidos:

- **Runtime.getRuntime().availableProcessors()** – Representa el número de procesadores disponibles en la JVM.
- **ForkJoinPool.defaultForkJoinWorkerThreadFactory** – Factory por defecto para retornar nuevos hilos.
- **null - Thread.UncaughtExceptionHandler** pasado como valor nulo
- **true** – Este parámetro hace que trabaje en modo asíncrono y setea el orden de las tareas a FIFO (First In First Out)

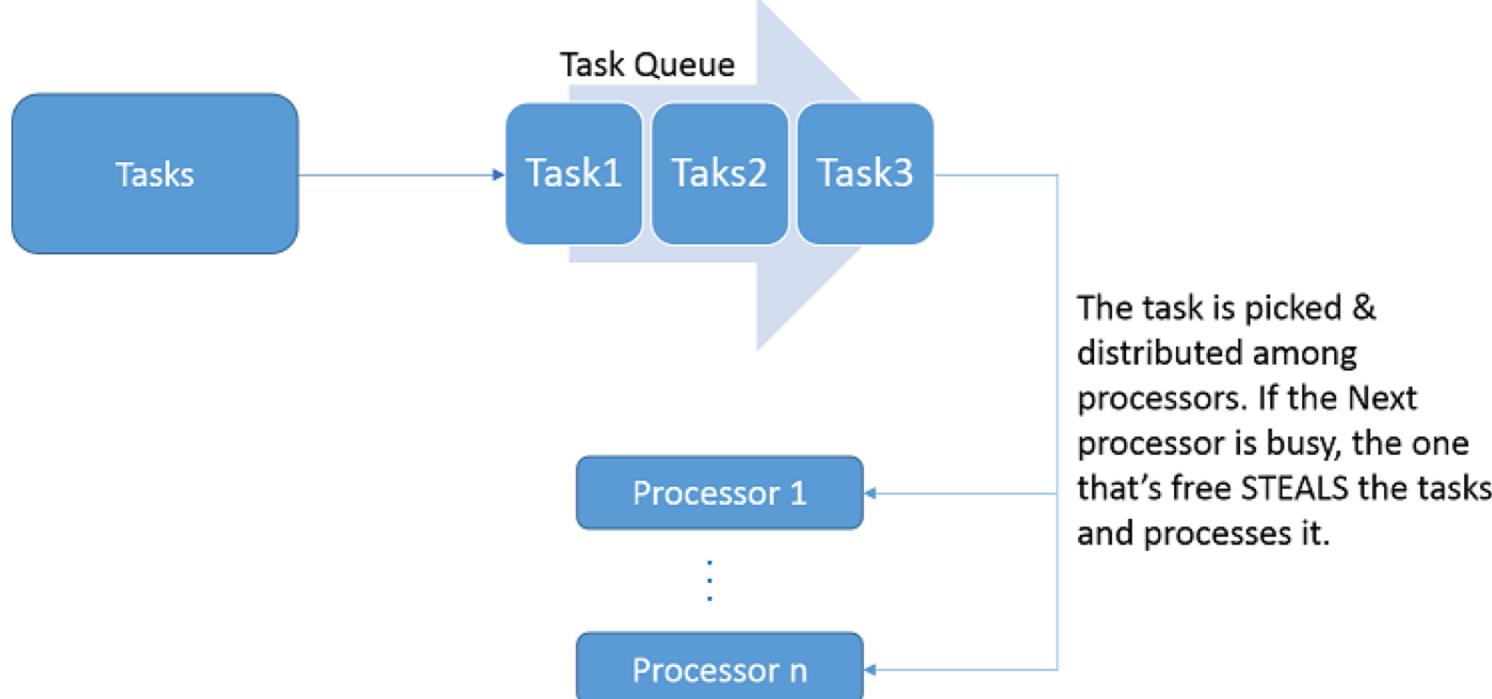
Work stealing

El newWorkStealingPool no es nada nuevo a nivel de lógica o utilidad en Java, pero si nos ayuda a ahorrar código al ser una capa de abstracción más sobre ForkJoinPool.

Work stealing

Un pool de hilos clásico tiene una cola, cada subprocesso que vaya a ejecutarse bloquea la cola, saca el proceso de la cola y la desbloquea.

Si las tareas son cortas y hay muchas de ellas, se crean contenciones en la cola, por lo que usar, por ejemplo, colas desbloqueadas ayuda, pero no resuelve por completo el problema.



Work stealing

En el caso de las interfaces y métodos modernos, se tiene que cada proceso posee su cola, cada que llega una nueva tarea, el hilo asignado la incluye en su cola, al momento de procesar una tarea, el hilo revisa si tiene tareas pendientes, si las tiene, sigue procesando su cola, de lo contrario, revisa otros hilos y “roba” procesos de sus colas para agilizar la ejecución de toda la aplicación.

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
    () -> "task1",
    () -> "task2",
    () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);

executor.shutdown();
}
```

Work stealing

Entonces, ¿usamos ForkJoinPool o newWorkStealingPool? Dependerá de lo que queramos hacer, si estamos trabajando con el framework Fork-Join y sabemos que las tareas pueden bloquearse de forma indefinida, podemos optar por ForkJoinPool, si nuestras tareas son relativamente cortas y dependen del CPU, podemos optar por newWorkStealingPool.

Examen!!

Ingresa en este enlace y responde según creas correcto:

<https://tinyurl.com/j8examen6>

6

Streams paralelos



Streams Paralelos - ¿Qué son?

- Permite la creación de streams paralelos.
- De esta forma un stream puede tomar ventaja de arquitecturas multicore para mejorar el rendimiento de código Java.
- Las operaciones son ejecutadas en paralelo y se pueden crear de dos maneras:
 - ✓ Usando el método parallelStream() sobre una colección
 - ✓ Usando el método parallel() en un stream

Streams Paralelos - Implicaciones

- Dado que cada substream es un hilo simple corriendo y actuando sobre los datos, tiene una carga adicional si lo comparamos a un stream secuencial.
- La comunicación entre los hilos es peligrosa y requiere de tiempo para la coordinación

Streams Paralelos - ¿Cuándo usarlo?

- Deben ser utilizados cuando la salida de la operación no requiere que sea dependiente del orden presente en la colección de origen.
- Puede utilizarse en casos de agregación de datos.
- Itera rápidamente sobre colecciones muy grandes.
- Lo pueden emplear desarrolladores para mejorar el rendimiento de operaciones en streams secuenciales.
- Si el entorno no es multihilo, entonces los streams paralelos crea un hilo y puede afectar a la nueva solicitud entrante.

Streams Paralelos

El concepto de Java Parallel Stream es un concepto sencillo de entender . En muchas ocasiones podemos tener un flujo de trabajo que necesitemos mejorar su rendimiento permitiendo su ejecución en paralelo a través de varios Threads.

Esto es algo que dependiendo del código del programa hacerlo sin utilizar streams es complicado. Vamos a ver un ejemplo.

Streams Paralelos

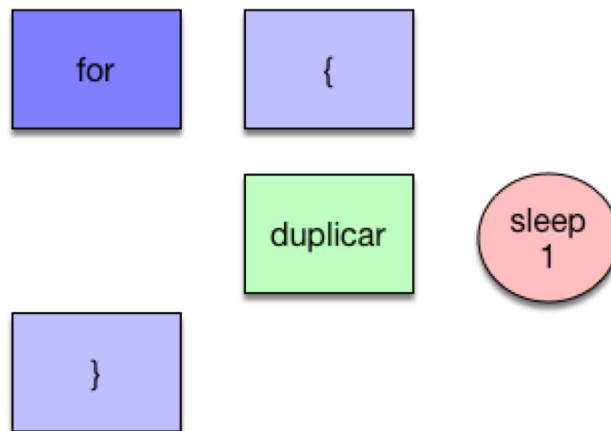
```
public static int duplicar( int numero) {  
  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
  
    return numero*2;  
}
```

Streams Paralelos

```
public static void main(String[] args) {  
  
    int total=0;  
  
    long numero1=System.currentTimeMillis();  
  
    for (int i=0; i < 10 ; i++) {  
  
        total+=duplicar(i);  
    }  
  
    long numero2=System.currentTimeMillis();  
  
    System.out.println(numero2-numero1);  
    System.out.println(total);  
}
```

Streams Paralelos

Acabamos de construir un sencillo bucle for en el cual invocamos al método duplicar. Lo curioso de este método es que pone el thread a dormir durante un segundo.



Streams Paralelos

Por lo tanto si realizamos 10 iteraciones el bucle tardará unos 10 segundos en ejecutarse.

¿Podríamos tardar menos? . En principio si , si dividieramos la tarea en varios Threads . El problema es que no se nos ocurre cómo dividirla de una forma clara o por lo menos de una forma sencilla. Es aquí donde el manejo de streams es una ventaja. ¿Como podemos realizar la misma operación con Streams?

Streams Paralelos

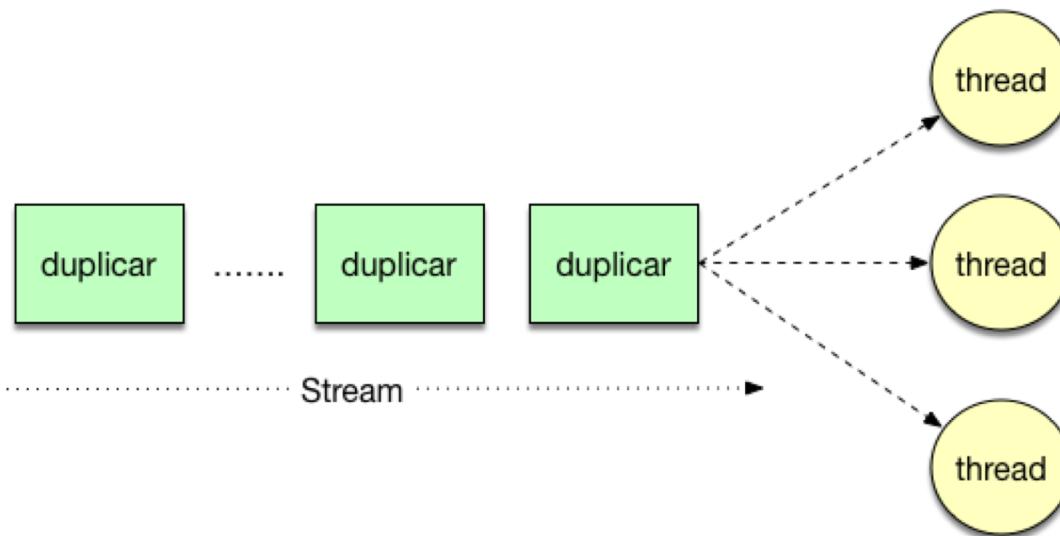
```
public static void main (String[] args) {  
  
    long numero1=System.currentTimeMillis();  
  
    IntStream lista= IntStream.range(1, 10);  
    int total=lista.map(Principal::duplicar).sum();  
  
    long numero2=System.currentTimeMillis();  
    System.out.println(numero2-numero1);  
    System.out.println(total);  
  
}
```

Streams Paralelos

```
public static int duplicar( int numero) {  
  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
  
    return numero*2;  
}
```

Streams Paralelos

En este caso el resultado es muy similar. Sin embargo es aquí donde si podemos utilizar de una forma sencilla la programación en paralelo y modificar el número de Threads que se encargan de procesar este Stream.



Streams Paralelos

Para ello nos es suficiente con modificar la siguiente línea y añadir el método `parallel()` para que java lance un conjunto de hilos para realizar la tarea:

```
int total=lista.parallel().map(Principal::duplicar).sum();
```

Streams Paralelos

Ejemplo práctico

Examen!!

Ingresa al siguiente formulario y responde según creas correcto:

<https://docs.google.com/forms/d/12JMUsX7x7-tbYm2uVu9576SEglw-vdIrpH0IHMeHeUQ/edit>

7

La API date-time



DateTime API

- Usualmente utilizabamos las clases **java.util.Date** y **java.util.Calendar**
- Ahora disponemos de clases que vienen a simplificar el uso de las clases relacionadas a fecha:
 - ✓ LocalDate
 - ✓ LocalTime
 - ✓ LocalDateTime
 - ✓ Instant
 - ✓ Duration
 - ✓ Period

7.1

Crear objetos de fecha



DateTime API

Haciendo uso de las nuevas clases disponibles en Java 8, obtener un objeto con la fecha actual es muy sencillo, basta con invocar el método now de la clase LocalDate, por ejemplo:

```
date = LocalDate.now();
System.out.println(date);
```

Representa una fecha sin tener en cuenta el tiempo.

DateTime API

Además de obtener la fecha actual, también podemos construir objetos de fecha pasando como parámetros el día, mes y año específicos:

```
LocalDate date = LocalDate.of(1990,11,11);
System.out.println(date.getYear());
System.out.println(date.getMonth());
System.out.println(date.getDayOfMonth());
```

se puede hacer uso del enum Month para dar legibilidad al código.

```
date = LocalDate.of(1990, Month.NOVEMBER, 11);
System.out.println(date);
```

DateTimeFormatter

- ◆ Formateador de fechas y horas para parsear, transformar u obtener las mismas en diferentes formatos.

```
LocalDate ld = LocalDate.now();
// Fecha en formato ISO
String fecha = ld.format(DateTimeFormatter.ISO_LOCAL_DATE);
```

```
DateTimeFormatter dateFormatter =
DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
// Fecha en formato SHORT
String otra_fecha = dateFormatter.format(ld);
```

7.2

Crear objetos de tiempo



DateTime API

Al igual que para las fechas, tenemos la clase LocalTime para crear objetos de tiempo, por ejemplo, para obtener la hora actual:

```
time = LocalTime.now();
System.out.println(time);
```

Representa un tiempo determinado

DateTime API

O crear un objeto de tiempo con parámetros específicos:

```
LocalDateTime dateTime = LocalDateTime.of(1990, 11, 11, 5,30,45,35);  
System.out.println(dateTime);
```

```
date = LocalDate.of(1990, Month.NOVEMBER, 11);  
time = LocalTime.of(5, 30,45,35);
```

```
dateTime = LocalDateTime.of(date, time);  
System.out.println(dateTime);
```

7.3

Combinar fecha y hora



DateTime API

Por otro lado, tenemos disponible el uso de `LocalDateTime` para crear un objeto con la fecha y hora actual combinados:

```
dateTime = LocalDateTime.now();
System.out.println(dateTime);
```

DateTime API

Con parámetros específicos, la llamada quedaría así:

//Con números

```
LocalDateTime dateTime = LocalDateTime.of(1990, 11, 11, 5,30,45,35);  
System.out.println(dateTime);
```

//Con objetos de hora y fecha separados

```
date = LocalDate.of(1990, Month.NOVEMBER, 11);  
time = LocalTime.of(5, 30,45,35);
```

```
dateTime = LocalDateTime.of(date, time);
```

```
System.out.println(dateTime);
```

Clock

- ◆ Representa el tiempo del sistema. Tiene información sobre la zona horaria en la que nos encontramos. Puede ser usado para acceder al tiempo en milisegundos (desde el 1 de enero de 1970) o para obtener diferentes instantes de tiempo para crear diferentes representaciones del mismo en diferentes zonas horarios o cronologías.
- ◆ `Clock myClock = Clock.systemDefaultZone();`

DateTime API

Otra clase que podemos usar para lograr esto es la clase Instant:

```
Instant instant = Instant.now();
System.out.println(instant);
```

Representa un instante en la línea temporal del reloj (Clock). Esta siempre en UTC. Puede ser usado como una representación universal de tiempo o para obtener otras representaciones del mismo (pero requeriría de zona horaria).

7.4

Zonas horarias



DateTime API

Para controlar la zona horaria del objeto de fecha, podemos hacer uso de la clase ZonedDateTime, a través del método atZone, podemos especificar la zona que queremos establecer en el objeto, veamos un ejemplo:

```
// Creamos un objeto de fecha hora actuales  
Instant instant = Instant.now();
```

```
// Podemos imprimir el objeto para ver el valor de fecha original  
System.out.println("Instant: " + instant);
```

DateTime API

```
// Creamos un objeto ZonedDateTime con el id de la zona horaria  
ZonedDateTime zone = ZonedDateTime.of("Asia/Aden");  
  
// Hacemos uso del método atZone, pasando como parámetro la zona  
ZonedDateTime result = instant.atZone(zone);  
  
// Imprimimos el resultado  
System.out.println("ZonedDateTime: " + result);
```

DateTime API

Incluir la zona horaria en nuestro objeto de fecha es sumamente útil, sobre todo a la hora de manejar cambios de horario, si queremos mantener siempre la hora y fechas correctas sin importar el país es aconsejable almacenar, por ejemplo, en la base de datos de nuestra aplicación, la hora y fechas junto con su zona horaria.

Si queremos manejar varios países, podemos almacenar la hora en UTC y aplicar la zona horaria al momento de realizar consultas.

7.5

Duraciones y períodos de tiempo



DateTime API

Además de crear y formatear objetos de fechas y horas, podemos calcular períodos (basados en días de duración) y duraciones (basadas en horas) a través de las clases Period y Duration.

Veamos algunos ejemplos:

Duración

- ◆ Duration, hace referencia a la diferencia que existe entre dos objetos de tiempo.
- ◆ En el siguiente ejemplo, la duración se calcula haciendo uso de dos objetos LocalTime:
 - ◆ LocalTime localTime1 = LocalTime.of(12, 25);
 - ◆ LocalTime localTime2 = LocalTime.of(17, 35);
 - ◆ Duration duration = Duration.between(localTime1, localTime2);

Duración

- ◆ Podemos hacer lo mismo con LocalDateTime
- ◆ `LocalDateTime localDateTime1 = LocalDateTime.of(2019, Month.MARCH, 18, 14, 13);`
- ◆ `LocalDateTime localDateTime2 = LocalDateTime.of(2019, Month.MARCH, 20, 12, 25);`
- ◆ `Duration duration = Duration.between(localDateTime1, localDateTime2);`

Duración

- ◆ También, se puede crear una duración basada en el método of(long amount, TemporalUnit unit).
 - ◆ Duration oneDayDuration = Duration.of(1, ChronoUnit.DAYS);
- ◆ El enum ChronoUnit, la cual es una implementación de TemporalUnit y nos brinda una serie de unidades de períodos de tiempo como ERAS, MILLENNIA, CENTURIES, DECADES, YEARS, MONTHS, WEEKS, etc

Duración

- ◆ También, se puede crear Duration basado en los métodos ofDays(long days), ofHours(long hours), ofMilis(long milis), ofMinutes(long minutes), ofNanos(long nanos), ofSeconds(long seconds)
 - ◆ Duration oneDayDuration = Duration.ofDays(1);

Period

- ◆ Hace referencia a la diferencia que existe entre dos fechas.
- ◆ `LocalDate localDate1 = LocalDate.of(2016, Month.JULY, 18);
LocalDate localDate2 = LocalDate.of(2018, Month.JULY, 20);`
- ◆ `Period period = Period.between(localDate1, localDate2);`

7.6

Aplicar formato a fechas



DateTime API

Ya sabemos crear objetos de fechas y horas, sin embargo, no siempre el formato por defecto es el ideal, por ello, Java 8 cuenta con la clase `DateTimeFormatter`.

Esta clase nos permite elegir entre formatos predefinidos e incluso definir nuestro propio formato para aplicarlo al objeto de fecha, tiempo o incluso de timezone.

Veamos algunos ejemplos:

DateTime API

```
//Formateando con ISO LOCAL DATE una fecha  
DateTimeFormatter.ISO_LOCAL_DATE.format(LocalDate.of(2018, 3, 9));
```

```
//Agregando TimeZone y un Offset de 3 horas  
DateTimeFormatter.ISO_OFFSET_DATE.format(LocalDate.of(2018, 3,  
9).atStartOfDay(ZonedDateTime.of("UTC-3")));
```

DateTime API

Agregando formatos con estilos predefinidos:

```
LocalDate anotherSummerDay = LocalDate.of(2016, 8, 23);
System.out.println(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).format(anotherSummerDay));
System.out.println(DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG).format(anotherSummerDay));
System.out.println(DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM).format(anotherSummerDay));
System.out.println(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT).format(anotherSummerDay));
```

DateTime API

Agregando nuestro formato:

```
String europeanDatePattern = "dd.MM.yyyy";
DateTimeFormatter europeanDateFormatter =
DateTimeFormatter.ofPattern(europeanDatePattern);
System.out.println(europeanDateFormatter.format(LocalDate.of(2016, 7, 31)));
```

DateTime API

Agregando nuestro formato:

```
String timeColonPattern = "HH:mm:ss";
DateTimeFormatter timeColonFormatter =
DateTimeFormatter.ofPattern(timeColonPattern);
LocalTime colonTime = LocalTime.of(17, 35, 50);
System.out.println(timeColonFormatter.format(colonTime));
```

DateTime API

Agregando nuestro formato:

```
String newYorkDateTimePattern = "dd.MM.yyyy HH:mm z";
DateTimeFormatter newYorkDateFormatter =
DateTimeFormatter.ofPattern(newYorkDateTimePattern);
LocalDateTime summerDay = LocalDateTime.of(2016, 7, 31, 14, 15);
System.out.println(newYorkDateFormatter.format(ZonedDateTime.of(summerDay, ZoneId.of("UTC-4"))));
```



Gracias!

Alguna pregunta?

Pueden contactarnos mediante
info@impactotecnologico.net y apuntarse a
algunos de nuestros cursos



