



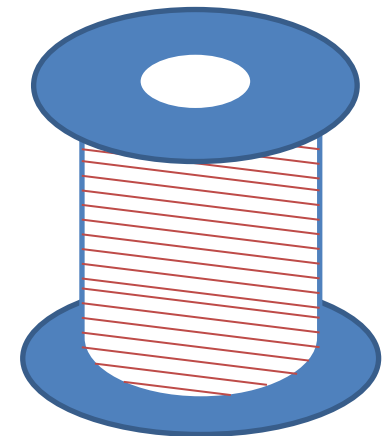
imatia
innovation

We help you to do more



Hilos

- Los hilos en Java nos permiten ejecutar diferentes tareas de forma paralela.
- Todo el sistema de gestión de los tiempos de ejecución o prioridades es llevado a cabo por la JVM
- Los hilos se pueden crear de dos maneras, bien sea extendiendo la clase *Thread* o bien implementando la interfaz *Runnable*



Hilos

- La ejecución de una clase que extiende de la clase *Thread* se realiza mediante la llamada al método *start()*.
- La ejecución de una clase que implementa la interfaz de *Runnable* se realiza mediante la instanciación de un hilo nuevo que recibe como parámetro del constructor una instancia nueva de la clase *Runnable*, y luego la ejecuta mediante el método *start()*.

```
public class CustomThread extends Thread {  
  
    @Override  
    public void run() {  
        //Ejecutar tarea  
    }  
}
```

```
public class ThreadExample {  
  
    public static void main(String[] args) {  
        CustomThread thread = new CustomThread();  
        thread.start();  
    }  
}
```

```
public class CustomRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        //Ejecutar tarea  
    }  
}
```

```
public class RunnableExample {  
  
    public static void main(String[] args) {  
        Thread customRunnable = new Thread(new CustomRunnable());  
        customRunnable.start();  
    }  
}
```

Ciclo de vida de un hilo

- Los hilos tienen diferentes estados, que se pueden consultar mediante el método *getState()* que poseen.
- **NEW:** Un hilo creado que no ha sido iniciado con el método *start()*.
- **RUNNABLE:** Hilo creado e iniciado con *start()*.
- **BLOCKED:** Un hilo en este estado no puede ejecutarse. Entra en este estado cuando intenta acceder a una sección de código de un método sincronizado bloqueado por otro hilo.
- **WAITING:** Esperando a que otro hilo realice una acción. Entra en este estado mediante los métodos *wait()* y *join()*

Ciclo de vida de un hilo

- **TERMINATED:** Estado de un hilo finalizado. Se puede usar el método *isAlive()* para comprobar su estado.
- **TIMED_WAITING:** Hilo en pausa hasta que acabe el tiempo indicado en el método *sleep()*. Permanece en este estado hasta que acabe el tiempo o se llame al método *interrupt()*.

Sincronización de los hilos

- La clase `Object` posee métodos que permiten la sincronización de hilos y comunicar bloqueos de recursos.
- ***wait()***: Libera un recurso para que otro hilo pueda usarlo (método sincronizado) y que da esperando hasta que otro hilo utiliza el método *notify()* o *notifyAll()*.
- ***notify()* y *notifyAll()***: Despiertan a los hilos que están esperando un acceso a un recurso compartido.

Hilos en espera

- Un hilo se pone en espera mediante el método `sleep()` o el `TimeUnit`.
- ***`sleep(long millis)`*** pone el hilo en espera durante la cantidad de milisegundos que se le pase por parámetro.
- ***`TimeUnit.XXXX.sleep(int number)`*** pone el hilo en espera la misma unidad de tiempo que se le pasa por parámetros, siendo XXXX la unidad temporal deseada (`MILLISECONDS`, `SECONDS`, `DAYS`....).

Otros elementos

- ***ThreadLocalRandom.current().nextInt(int origin, int bound)*** permite obtener un número aleatorio desde *origin* (incluído) hasta *bound* (excluído).
- ***AtomicInteger*** nos permite el uso de un entero sincronizado entre hilos que permiten su decremento o incremento de valor.

Timers

- Los timers nos permiten agendar ejecuciones de hilos en el futuro, y si se deben repetir.

```
Timer timer = new Timer();
```

- A través del método *schedule*, podremos programar la tarea para un tiempo futuro y poder controlar el retraso con el que queremos que se repita la tarea.
- Una vez se acabe el Timer, se debe ejecutar el método *cancel()* para finalizarlo.
- *awaitTermination(TimeUnit t)* bloquea el hilo para esperar por los hilos programados el tiempo determinado antes de continuar con la ejecución.

Executors

- Los ejecutores nos permiten almacenar hilos que se ejecutan una vez el hilo anterior se haya ejecutado.
- Los más comunes son el `newSingleThreadExecutor()` y el `newFixedThreadPool(int nThreads)`.

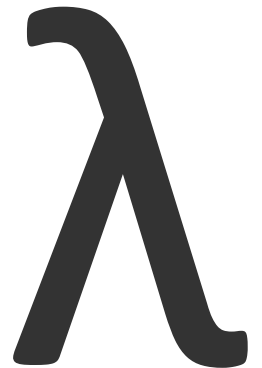
```
ExecutorService executor = Executors.newSingleThreadExecutor();  
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 2);
```

- Ejecutan el thread mediante el método *submit(Runnable t)*.

```
Future<?> resultRunnable = executor.submit(taskRunnable);
```

Expresiones lambda y métodos de referencia

- Las expresiones lambda nos permiten implementar un método sin que sea necesario que exista una clase que la contenga, mediante el uso de interfaces funcionales (*@FunctionalInterface*) que nos permiten implementar un método.
- Existen diferentes tipos de expresiones lambda en Java, como son los *Consumer*, *Supplier*, *Function* y *Predicate*.



Consumer

- Las lambdas de tipo Consumer reciben por parámetro un elemento y no devuelven nada.

```
Consumer<Date> printDate = (date) -> {  
    System.out.println(date);  
};  
printDate.accept(new Date());
```

- Se indica el parámetro que va a recibir, una flecha y la implementación del cuerpo del método. Para ejecutarla, se llama al método *accept()*.
- Se puede simplificar eliminando los paréntesis iniciales y como tiene una instrucción, no usar llaves.

```
Consumer<Date> printDate = date -> System.out.println(date);
```

BiConsumer

- Lo mismo que el Consumer, pero recibe 2 parámetros.
- Como recibe 2 parámetros, no se pueden simplificar eliminando los paréntesis de los parámetros de entrada.

Supplier

- Esta expresión lambda no recibe ningún parámetro de entrada, pero tiene que devolver un valor.

Function y BiFunction

- Estas expresiones lambda reciben uno o dos parámetros respectivamente y devuelven un valor.

Predicate y BiPredicate

- Estas expresiones lambda reciben de uno a dos parámetros respectivamente y devuelven un valor booleano.

Métodos referencia

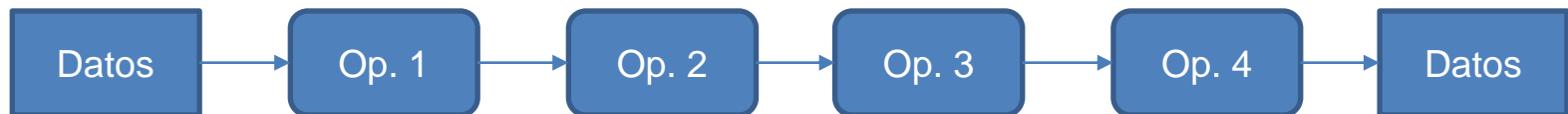
- Una función lambda se puede simplificar si usamos los métodos referencia para abreviar su construcción.

```
Function<String, String> toUpperCaseMethodReference = String::toUpperCase;  
Supplier<PersonJava8> createPerson = PersonJava8::new;  
BiConsumer <PersonJava8, String> setNameMethodReference = PersonJava8::setName;
```

- Permite eliminar los parámetros de entrada, asociando el primer parámetro al tipo de objeto que ejecuta la clase y el segundo parámetro, en caso de existir, como parámetro de entrada de dicho método.

API Stream

- Permite ejecutar acciones en orden para tratar flujos de datos, elemento a elemento.
- Un símil sería el de un fluido que va por una canalización sobre el que se realiza cierta manipulación mientras recorre dicha canalización.



API Stream

- Son flujos de datos que se procesan de manera secuencial o paralela.
- Reducen las tareas de transformación de elementos.
- Transforma los valores a través de operadores
- Puede ser creado desde elementos con la interfaz *Collection*, arrays o rangos.
- Son inmutables, por lo que devuelven un nuevo stream después de cada operación, sin afectar al stream previo.
- Facilita la concurrencia.

API Stream - Generador

- Un Stream comienza con un *Collection*, un array o una función generadora.

```
Stream<String> names = Stream.of( ...values: "María", "Jake", "Gregory");  
                                String[] arrayNames = {"María", "Jake", "Gregory"};  
                                Stream<String> names = Arrays.stream(arrayNames);  
  
List<String> nameList = new ArrayList<>();  
nameList.add("María");  
nameList.add("Jake");  
nameList.add("Gregory");  
Stream<String> names = nameList.stream();  
  
IntStream i = IntStream.range(0, 20);
```

API Stream – Operaciones intermedias

- De 0 a n operaciones intermedias. Son operaciones que no se ejecutan hasta que se ejecuta una operación terminal, y son las que encadenan y transforman los datos del stream, permitiendo su encadenación.
- Estas operaciones son, entre otras, *map()*, *limit()*, *filter()*, *sorted()*, *distinct()*, *reduce()*, *peek()*, *mapToInt()*, *flatMap...*
- *map()* → Utiliza una expresión lambda para transformar el valor.
- *limit()* → Establece un límite de tamaño para el stream.

API Stream – Operaciones intermedias

- *sorted()* → Permite ordenar los elementos de un stream.
- *distinct()* → Solo permite elementos únicos en el stream, sin duplicados.
- *reduce()* → Permite la reducción de los elementos de un stream hasta que solo quede uno
- *peek()* → Permite consultar el valor de un elemento del stream en ese punto de la ejecución.

API Stream – Operaciones intermedias

- *mapToInt()* → Devuelve un stream de enteros después de aplicar una función a cada elemento
- *flatMap()* → Divide los elementos en multiples streams, les ejecuta una operación y los vuelve a juntar en un único stream

API Stream – Operación terminal

- Una única operación terminal. Sirve para ejecutar todas las operaciones intermedias y una operación que devuelve algo que no es un stream.
- Las más usadas son *collect()*, *forEach()*, *count()*, *anyMatch()*.
- *collect()* → Reúne todos los elementos procesados en una *Collection*, como por ejemplo, un *List*.
- *forEach()* → Lleva a cabo una acción en cada elemento del stream
- *count()* → cuenta los elementos finales del stream.

API Stream – Operación terminal

- *anyMatch()* → Ejecuta una operación lambda de tipo Predicate y devuelve un valor booleano dependiendo del resultado de la operación en todo el stream.

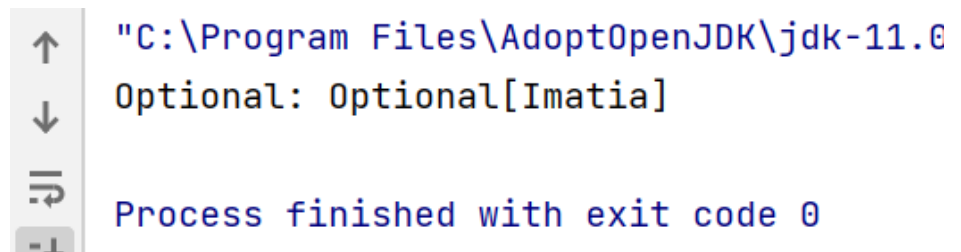
API Stream – Parallel

- Es un operador intermedio que permite operar con varios valores simultáneos en vez de secuencial. Solo se debe usar cuando el orden de procesamiento no sea relevante ni se ejecuten operaciones basadas en un orden, como *findFirst()*

API Stream – Optional

- Optional nos permite representar que un valor esté presente o no, evitando que en tiempo de ejecución puedan ocurrir *NullPointerException*.
- Es una clase que contiene a otro objeto.
- El tipo de dato es *Optional<T>*, siendo T el tipo de dato que vamos a envolver

```
String b_name = "Imatia";  
Optional<String> opt = Optional.of(b_name);  
System.out.println("Optional" + opt);
```

A screenshot of a terminal window with a light gray background. On the left side, there is a vertical toolbar with icons for back, forward, search, and other navigation functions. The terminal text is as follows:

```
"C:\Program Files\AdoptOpenJDK\jdk-11.0  
Optional: Optional[Imatia]  
  
Process finished with exit code 0
```

API Stream – Optional

- Tenemos 3 constructores de Optional
- ***Optional.empty()*** → Devuelve un Optional vacío
- ***Optional.of(value)*** → Devuelve un Optional de una variable no nula.
- ***Optional.ofNullable(value)*** → Devuelve un Optional de una variable que puede ser nula.
- Se emplean los métodos *.ifPresent()* o *.isEmpty()* para comprobar si internamente tiene un valor o no

```
String b_name = "Imatia";
Optional<String> opt = Optional.of(b_name);
System.out.println("Optional: " + opt.isPresent());

String b_name_null = null;
Optional<String> opt_null = Optional.of(b_name);
System.out.println("Optional null: " + opt.isEmpty());
```

API Stream – Optional

- El método ***.ifPresentOrElse()***, disponible a partir de Java 9, tiene como parámetro un *Consumer* que se ejecutará en caso de que el *Optional* contenga un valor y un *Runnable* que se ejecutará en caso de que el *Optional* no tenga valor

```
opt.ifPresentOrElse(System.out::println,  
                    () -> System.out.println("Valor nulo"));
```

API Stream – Optional

- Para obtener el valor de un *Optional*, se usa el método `.get()`, **únicamente** si el valor de `isPresent()` es *true*
- En caso de que queramos obtener el valor del *Optional* y, en caso de que el valor de `.isPresent()` sea *false*, devolver otro valor por defecto, existen los métodos `.orElse()` `.orElseGet()` (el método recomendado y más eficiente es **`.orElseGet()`** debido a que el método `.orElse()` instancia el objeto a devolver independientemente del valor del *Optional*, y no sólo en caso de que `.isPresent()` sea *false*)

API Stream – Optional

- En caso de que *.isPresent()* sea *false* y no queramos mandar un valor en su lugar, podemos utilizar el método *.orElseThrow()* para que, de no estar presente un valor, propague la excepción ***NoSuchElementException*** (mismo funcionamiento que *.get()*). Este método también admite por parámetro una función lambda de tipo *Supplier*

API Stream – Optional

- Los Optional son compatibles con los métodos *.map()*, *.filter()* y *.flatMap()* de un *Stream*
- Ciertos métodos de los *Stream* otorgan también elementos Optional, como *.findFirst()*, *.findAny()*, *.min()*, *.max()* y *.reduce()*