

Var

Java, la palabra reservada **var** se utiliza para declarar variables locales de forma simplificada mediante la inferencia de tipos. La inferencia de tipos es un proceso que determina automáticamente el tipo de datos de una variable en tiempo de compilación, en función del valor que se le asigna.

Stream

En Java, un **stream** es una biblioteca que permite trabajar con colecciones, como *List* y *Sets*, mediante una única llamada de método. Los **streams** son clases que encapsulan *arrays* o colecciones y permiten que estos soporten operaciones utilizando lambdas. Esto simplifica la operación sobre los elementos de la colección o sobre la colección misma.

Final (inmutabilidad)

Para variables de tipo primitivo, la palabra clave **final** significa que el valor, una vez asignado, no se puede cambiar. Para las variables de referencia, significa que después de asignar un objeto, no puede cambiar la referencia a ese objeto.

final: en este contexto indica que una variable es de tipo constante: no admitirá cambios después de su declaración y asignación de valor. **final** determina que un atributo no puede ser sobrescrito o redefinido. O sea: no funcionará como una variable “tradicional”, sino como una constante.

IntStream

IntStream puede simplificar operaciones cotidianas y el código es mucho más compacto que usar el clásico bucle *for*.

Representa una secuencia de elementos primitivos con valores *int* que admite operaciones agregadas secuenciales y en paralelo.

Los **streams** son envoltorios alrededor de una fuente de datos, lo que permite operar con esa fuente de datos y hacer que el procesamiento masivo sea conveniente y rápido.

Range Stream

Podemos crear un *Stream* de 100 números de una forma más compacta usando el método *range()*.

El código es mucho más compacto que usar el clásico bucle *for*. A veces se suele olvidar que los *IntStreams* pueden simplificar sobre manera operaciones cotidianas.

Filter Stream

Filtro de secuencia (Predicado predicado) devuelve una secuencia que consta de los elementos de esta secuencia que coinciden con el predicado dado. Esta es una operación intermedia. Estas operaciones siempre son diferidas, es decir, ejecutar una operación intermedia como *filter()* en realidad no realiza ningún filtrado, sino que crea una nueva secuencia que, cuando se recorre, contiene los elementos de la secuencia inicial que coinciden con el predicado dado.

Parallel Stream

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.

∴

Es un operador de doble punto que se llama **referencia de método**. Fue introducido en Java 8 y se utiliza para aumentar la reutilización de una expresión lambda o función.

Los operadores en programación son símbolos que permiten realizar operaciones aritméticas, relacionar elementos o hacer preguntas que involucran más de una condición.

java.lang.Thread.currentThread()

El método `java.lang.Thread.currentThread()` devuelve una referencia al objeto de subproceso que se está ejecutando actualmente.

Este método devuelve el hilo que se está ejecutando actualmente.

→ operador

En Java, el operador de flecha (`->`) es el operador lambda Java. El token (`->`) puede tener cero o más argumentos, y (cuerpo) contiene una o más líneas de código que se ejecutan cuando se llama la función.

Este operador nos sirve para crear funciones sin nombre.

Runnable

La implementación de la interface `Runnable` es la forma habitual de crear threads. Las interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requerimientos comunes al conjunto de clases a implementar.

`Runnable` es la interface de Java que dispone de un método `run` y nos permite ejecutar una Tarea en paralelo desde un programa main usando la clase `Thread`.

Diferencias

Un `Thread` tiene un estado al que el `Runnable` probablemente no necesita acceder. Tener acceso a más estados de lo necesario es un diseño deficiente. Los hilos ocupan mucha memoria. Crear un nuevo subproceso para cada pequeña acción requiere tiempo de procesamiento para asignar y desasignar esta memoria.

Por un lado, tenemos la interfaz `Runnable`, que proporciona la base para correr procesos en hilos independientes. Y por otro lado tenemos también la clase `Thread`, que extiende de `Runnable`, que implementa los procesos a ejecutar más fácilmente.

Esto es lo del JShell

```
PS C:\Users\D E L L\Documents\Nahomi\CICLO IV\Programación Avanzada\trabajo_clases>
jshell
```

```
| Welcome to JShell -- Version 11.0.16.1
| For an introduction type: /help intro
```

```
jshell> public class HelloThread extends Thread {
...>     @Override
...>     public void run() {
...>         String helloMess = String.format("Hola, soy %s",
Thread.currentThread().getName());
...>         System.out.println(helloMess);
...>     }
...> }
```

```
| created class HelloThread
```

```
jshell> public class HelloRunneable implements Runnable {
...>     @Override
...>     public void run() {
...>         String threadName = Thread.currentThread().getName();
...>         String helloMess = String.format("Hola, soy %s", threadName);
...>         System.out.println(helloMess);
...>     }
...> }
```

```
| created class HelloRunneable
```

```
jshell> var t1 = new HelloThread();
```

```
t1 ==> Thread[Thread-0,5,main]
```

```
jshell> var t2 = new Thread(new HelloRunneable());
```

```
t2 ==> Thread[Thread-1,5,main]
```

```
jshell> Thread myThread = new Thread(new HelloRunneable(), "my-thread");
```

```
myThread ==> Thread[my-thread,5,main]
```

```
jshell> Thread t3 = new Thread(() -> {
```

```
    ...>
```

Signatures:

Thread(Runnable target)

Thread(ThreadGroup group, Runnable target)

Thread(String name)

Thread(ThreadGroup group, String name)

Thread(Runnable target, String name)

Thread(ThreadGroup group, Runnable target, String name)

Thread(ThreadGroup group, Runnable target, String name, long stackSize)

Thread(ThreadGroup group, Runnable target, String name, long stackSize, boolean inheritThreadLocals)

```
jshell> Thread t3 = new Thread(() -> {
```

```
    ...> System.out.println(String.format("Hola, soy %s",
Thread.currentThread().getName()));
```

Signatures:

Thread(Runnable target)

Thread(ThreadGroup group, Runnable target)

Thread(String name)

Thread(ThreadGroup group, String name)

Thread(Runnable target, String name)

Thread(ThreadGroup group, Runnable target, String name)

Thread(ThreadGroup group, Runnable target, String name, long stackSize)

Thread(ThreadGroup group, Runnable target, String name, long stackSize, boolean inheritThreadLocals)

```
jshell> Thread t3 = new Thread(() -> {  
    ...> System.out.println(String.format("Hola, soy %s",  
Thread.currentThread().getName()));  
    ...> });  
t3 ==> Thread[Thread-2,5,main]
```

Otras clases

De la clase donde no estaba el parallel()

```
import java.util.stream.IntStream;

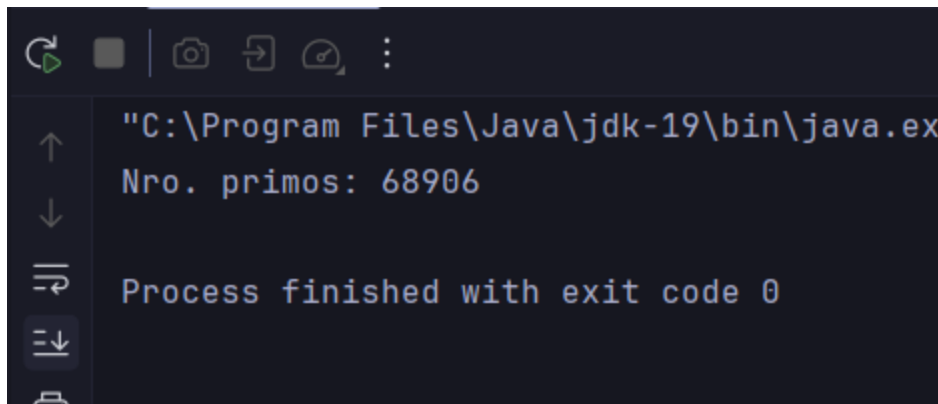
public class Main {
    1 usage
    private static final IntStream values = IntStream.range(100_000,1_000_000);
    public static void main(String[] args) {

        var results = values.filter(Main::isPrime); // es una referencia al método ---> ::
        // :: es un operador que permite reutilizar varias veces el método al que hace referencia

        System.out.printf("Nro. primos: %d\n" , results.count());
    }
}

2 usages
public static boolean isPrime(int nro) {
    if(nro < 2) return false;
    if(nro == 2) return true;
    if(nro % 2 == 0) return false;

    for(var i = 2; i < nro; i++) {
        if(nro % i == 0) return false;
    }
    return true;
}
}
```



```
"C:\Program Files\Java\jdk-19\bin\java.exe
Nro. primos: 68906

Process finished with exit code 0
```

Del que si tenía el parallel()

```
import java.util.stream.IntStream;

public class Test1 {
    1 usage
    private static final IntStream values = IntStream.range(100_000,1_000_000);
    public static void main(String[] args) {
        var results = values.parallel().filter(Main::isPrime);

        System.out.printf("Nro. primos: %d\n" , results.count());
    }

    no usages
    public static boolean isPrime(int nro) {
        if(nro < 2) return false;
        if(nro == 2) return true;
        if(nro % 2 == 0) return false;

        for(var i = 2; i < nro; i++) {
            if(nro % i == 0) return false;
        }
        return true;
    }
}
```

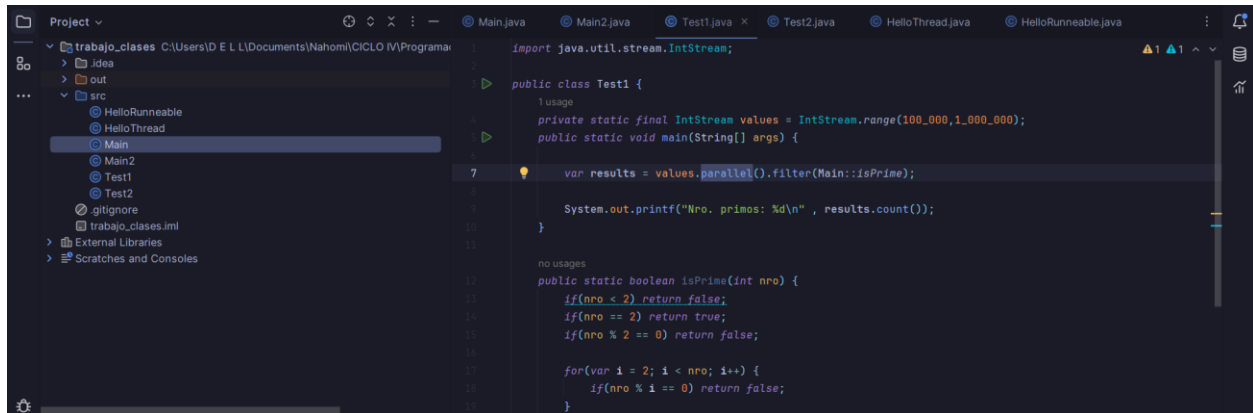
```
"C:\Program Files\Java\jdk-19\bin\java.exe"  
Nro. primos: 68906  
  
Process finished with exit code 0
```

// No sé cómo configurar para que se me muestre el tiempo de ejecución 😞

Clase de los 5 hilos

```
public class Test2 {  
    public static void main(String[] args) {  
        System.out.printf("%s - ejecutándose\n", HelloThread.currentThread().getName());  
        // creación 5 hilos  
        for(int i = 0 ; i < 5; i++) {  
            new Thread() -> {  
                System.out.printf("%s - ejecutándose\n", HelloThread.currentThread().getName());  
            }.start();  
        }  
        System.out.printf("Fin del %s\n", HelloThread.currentThread().getName());  
    }  
}
```

```
"C:\Program Files\Java\jdk-19\bin\java.exe"  
main - ejecutándose  
Thread-0 - ejecutándose  
Fin del main  
Thread-1 - ejecutándose  
Thread-4 - ejecutándose  
Thread-3 - ejecutándose  
Thread-2 - ejecutándose  
  
Process finished with exit code 0
```

The screenshot shows an IDE with a project named 'trabajo_clases'. The 'src' directory contains several files: 'HelloRunnable', 'HelloThread', 'Main', 'Main2', 'Test1', and 'Test2'. The 'Test1.java' file is open, showing the following code:

```
import java.util.stream.IntStream;

public class Test1 {
    1 usage
    private static final IntStream values = IntStream.range(100_000, 1_000_000);
    public static void main(String[] args) {
        7 var results = values.parallel().filter(Main::isPrime);

        System.out.printf("Nro. primos: %d\n", results.count());
    }

    no usages
    public static boolean isPrime(int nro) {
        13 if(nro < 2) return false;
        14 if(nro == 2) return true;
        15 if(nro % 2 == 0) return false;
        16
        17 for(var i = 2; i < nro; i++) {
        18     if(nro % i == 0) return false;
        19 }
    }
}
```

Esas son las clases trabajadas uwu