> ¿Cuales son los contras y los pros de usar For o Stream?

Algunos **pros** de usar `for`:

- Muy probablemente ya lo conoces, entiendes y dominas
- Es una estructura de control que en muchas ocasiones el compilador puede optimizar
- Cuando tienes una clase que implementa la interfaz `Iterable<E>` puedes escribir tu `for` en la forma `for-each` (`for(E e: iterableInstance){…}`) que es bastante mas comoda que un `for` regular (`for( inicializacion; condicion ; avance){…}`)
- Manejar arreglos y colecciones es relativamente facil con un `for`

Algunos posibles **contras** (no necesarimente son malos o problematicos en todos los casos):

- Es relativamente facil cometer errores que alteren el orden del `for`, por ejemplo, :

```
for(int i = 0; i < limite; i++){
// Algo de codigo
i = i % 2
// Mas codigo
}
```

Otro ejemplo de un posible error:

```
for(int index = 0; index <
list.size(); i++) {
   //Algunas operaciones
   list.add(…); // Este es un error,
creando un loop infinito
}
```

- La sintaxis podria no ser tan clara en un principo cuando se usan indices no tan obvios:

```
for(int i = …; … ; …) {
    for(int j = …; … : …) {
        //Error no tan obvio, deberia
usarse la j pero se confunde con la i
        E myElement = myArray[i][i];
    }
}
```

Todos estos son casos hipoteticos, pero pueden llegar a pasar por descuidos pequeños.

La clase `Stream` esta pensada para que las iteraciones o el procesamiento de los elementos sea un poco mas “dinamicos”.

En un `for` pones todas las operaciones dentro del `for` o escribes multiples `for`. En `Stream` cada operacion modifica el `Stream` y genera un nuevo `Stream`. Idealmente un `Stream` tiene nuevos elementos constantemente, por ello no puedes tener un metodo `size()` o un atributo/metodo `length`, pues no sabes cuantos elementos apareceran en el `Stream`. En teoria no puedes determinar cuando un `Stream` dejara de publicar elementos (idealmente). Sin embargo tienes la posibilidad de operara cada nuevo elemento que aparezca en el `Stream`.

Algunos **pros** de tener un `Stream`:

- Es mucho mas facil hacer operaciones en paralelo
- Es mas legible porque las operaciones son un poco mas explicitas (aunque depende del estilo de cada quien)
- Tienes operaciones ya predefinidas
- Hay muchas operaciones que son optimizadas en tiempo de compilacion

- Puedes convertir facilmente un `Stream<A>` en un `Stream<B>` usando los metodos ya existentes en `Stream`
- Puedes convertir facilmente muchas clases a `Stream` (por ejemplo, `Collection#stream()`
- Al ser un tipo de dato puedes recibir o retornar `Stream` parcialmente operado:

```
public Stream<User>
getUserNamesStream(){
    //Obtener los nombres de usuario
    return userNamesStream;
}

public Stream<String>
getUserNamesByActiveStatus(Stream<Strir
 users){
        return
users.filter(User::isActive)

.map(User::getUserName);
}
```

Algunos **contras**:

- Cuando necesitas un dato final para mostrar o retornar algun dato final, tienes que convertir tu `Stream`
- No tienes forma directa de frenar o saltarte pasos de una iteracion de un `Stream` a diferencia de un `for` donde puedes usar `break` y `continue`
- Debes aprender la API de `Stream`
- Buscar errores puede ser un poco mas complicado

Siendo realistas, habra ocasiones que sera mas simple usar un `for` para iterar una coleccion (`List`, `Set`, `Map`, `Vector`, etc.) y habra ocasiones que operar un `Stream` o generar un `Stream` demostrara un mejor *performance* o agregara mucha legbilidad.

Asi que la respuesta corta es: **Depende de cada caso**