**1 stream() Method (First Call)**

mylist.stream()

* **What happens in memory?**
  + The stream() method **does not create a new list**.
  + Instead, it creates a **Stream object** that wraps mylist.
  + This stream is just a **pipeline** that refers to the original list but does **not process anything yet**.
  + This is why **streams are lazy**—they don’t do anything until a terminal operation (like .sum()) is called.

**2 filter(i -> i > 10)**

mylist.stream().filter(i -> i > 10)

* filter() **does not modify the original list**.
* It **returns a new stream** that **wraps the previous stream** (not a new list!).
* This new stream just **remembers the filter condition** (i > 10), but **no filtering happens yet**.

**Internally:**

* The stream **stores the filtering logic**.
* When a terminal operation (like .sum()) is called, it **pulls elements one by one** and applies the filter.

**3 mapToInt(i -> i)**

mylist.stream().filter(i -> i > 10).mapToInt(i -> i)

* mapToInt(i -> i) transforms the **Stream** into an **IntStream**.
* The **IntStream** also **wraps** the previous stream and can **convert** elements to primitive int.

**Internally:**

* It doesn’t process anything immediately.
* It **remembers the mapping logic** (i -> i).

**4 sum() (Triggering Execution as “sum” is a terminal operation)**

mylist.stream().filter(i -> i > 10).mapToInt(i -> i).sum();

* **Now the pipeline executes!**
* **Steps during execution:**
  1. **Pull elements one by one** from mylist.
  2. **Apply filter** (i > 10).
  3. **Apply mapToInt** (convert Integer to int).
  4. **Sum up the filtered values**.

**Summary:**

1. **First stream() wraps the original list** (but doesn’t copy elements).
2. **Each intermediate operation (filter, mapToInt) creates a new stream that wraps the previous one** (but still does nothing until a terminal operation is called).
3. **Only when .sum() is called, elements start flowing through the pipeline** (filter → map → sum).