The filtering operations filters the given stream and returns a new stream, which contains only those elements that are required for the next operation.

`filter()` method

The `Stream` interface has a `filter()` method to filter a stream. This is an intermediate operation. Below is the method definition of `filter()` method.

> **Stream filter(Predicate<? super T> predicate)**

Parameter -> A predicate to apply to each element to determine if it should be included.

Return Type -> It returns a stream consisting of the elements of this stream that match the given predicate.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class StreamDemo {

    public static void main(String[] args) {

        //Created a list of integers
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(12);
        list.add(23);
        list.add(45);
        list.add(6);

        list.stream()                           // Created a stream from the list
                .filter(num -> num > 10)        //filter operation to get only numbers greater than 10
                .forEach(System.out::println);  // Printing each number in the list after filtering.

        //Again printing the elements of List to show that the original list is not modified.
        System.out.println("Original list is not modified");
        list.stream()
                .forEach(System.out::println);
    }
}
```

Run Save Reset

In the above example, we created a list of integers. We followed the below steps:

1. Create a stream from our list.
2. Apply a `filter()` operation on this stream. We want to print only those numbers which are greater than 10, so we add a filter.

Please note that the filter operation does not modify the original List.

`filter()` with custom object

Let's look at another example of `filter()` with a custom object.

In the below example, we are using multiple conditions in the filter method.

```
import java.util.ArrayList;
import java.util.List;

public class StreamDemo {

    public static void main(String[] args) {
        //Created a list of Person object.
        List<Person> list = new ArrayList<>();
        list.add(new Person("Dave", 23));
        list.add(new Person("Joe", 18));
        list.add(new Person("Ryan", 54));
        list.add(new Person("Iyan", 5));
        list.add(new Person("Ray", 63));

        // We are filtering out those persons whose age is more than 18 and less than 60
        list.stream()
                .filter(person -> person.getAge() > 18 && person.getAge() < 60)
                .forEach(System.out::println);

    }
}

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
```

Run Save Reset

In the above example, we used multiple conditions inside our filter.

`filter()` chaining

In the above example, we wrote all the conditions in a single filter.

We can also chain the filter method to make the code more readable.

```
import java.util.ArrayList;
import java.util.List;

public class StreamDemo {

    public static void main(String[] args) {
        List<Person> list = new ArrayList<>();
        list.add(new Person("Dave", 23));
        list.add(new Person("Joe", 18));
        list.add(new Person("Ryan", 54));
        list.add(new Person("Iyan", 5));
        list.add(new Person("Ray", 63));

        list.stream()
                .filter(person -> person.getName() != null ) // Filtering the object where name is not nu
                .filter(person -> person.getAge() > 18 ) // Filtering the objects where age is greater th
                .filter(person -> person.getAge() < 60) // Filtering the objects where age is less than 6
                .forEach(System.out::println);

    }
}

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
```

Run Save Reset