**Java 8 Stream Filter with examples**

The filter() is an intermediate operation that reads the data from a stream and returns a new stream after transforming the data based on the given condition. Lets take a simple example first and then we will see the examples of stream filter with other methods of the stream.

## A Simple Example of Java Stream Filter()

## Java Stream Example

To understand how stream works, lets take an example without using stream and then we will see the same example with streams.

**Finding certain strings without using Stream**

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

int count = 0;

for (String str : names) {

if (str.length() < 6)

count++;

}

System.out.println("There are "+count+" strings with length less than 6");

}

}

Output:

There are 3 strings with length less than 6

**Same example using Stream**

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

//Using Stream and Lambda expression

long count = names.stream().filter(str->str.length()<6).count();

System.out.println("There are "+count+" strings with length less than 6");

}

}

Output:

There are 3 strings with length less than 6

In this example we are creating a stream from the list of names using stream() method and then we are creating another stream of long names using stream filter(). As I mentioned above, the stream filter transforms the data of one stream into another stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

//Creating the stream of all names

Stream<String> allNames = names.stream();

//Creating another stream by filtering long names using filter()

Stream<String> longNames = allNames.filter(str -> str.length() > 6);

//displaying the long names

longNames.forEach(str->System.out.print(str+" "));

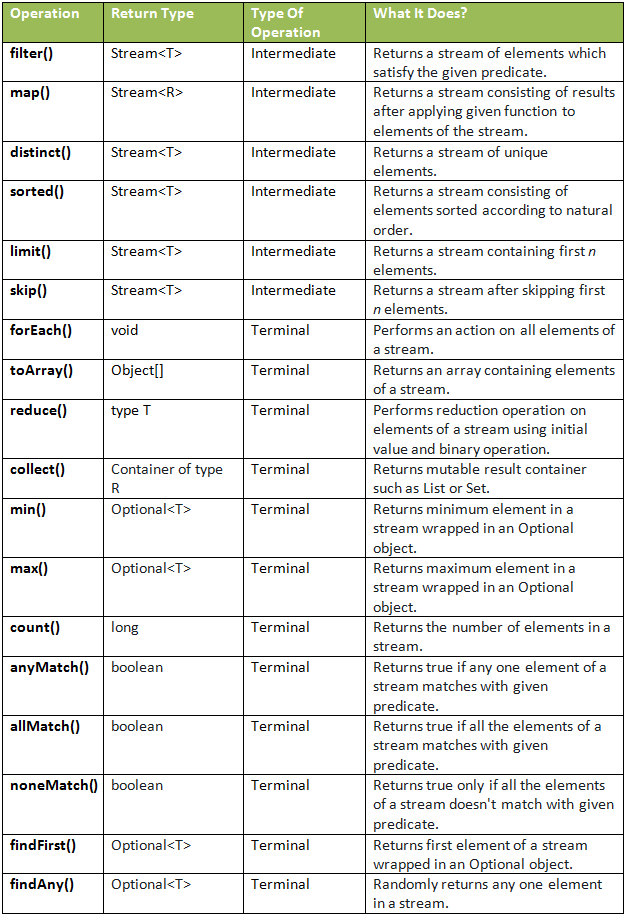
}

}

The **terminal operations** return a result of a certain type and **intermediate operations** return the stream itself so we can chain multiple methods in a row to perform the operation in multiple steps.

Streams are created on a source, e.g. a java.util.Collection like List or Set. The Map is not supported directly, we can create stream of map keys, values or entries.

Stream operations can either be executed sequentially or parallel. when performed parallelly, it is called a parallel stream



# A Guide to Java Streams in Java 8: In-Depth Tutorial With Examples

First of all, Java 8 Streams should not be confused with Java I/O streams (ex: *FileInputStream* etc); these have very little to do with each other.

Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

**A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**

This functionality – *java.util.stream* – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

#### **Java Stream Creation**

Let’s first obtain a stream from an existing array:

private static Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

Stream.of(arrayOfEmps);

We can also obtain a stream from an existing list:

private static List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream();

Note that **Java 8 added a new *stream()*method to the *Collection* interface.**

And we can create a stream from individual objects using *Stream.of()*:

### **Java Stream Operations**

#### **forEach**

forEach() is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

s.forEach(action -> syso(action));

**forEach() is a terminal operation**, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used. We’ll talk more about terminal operations in the next section.

#### **map**

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

public void whenMapIdToEmployees\_thenGetEmployeeStream() {

Integer[] empIds = { 1, 2, 3 };

List<Employee> employees = Stream.of(empIds)

.map(employeeRepository::findById)

.collect(Collectors.toList());

assertEquals(employees.size(), empIds.length);

}

Maps have a different structure, with a mapping from keys to values, without sequence. However, this doesn't mean that we can't convert a Map structure into different sequences which then allow us to work in a natural way with the Stream API.

Map<String, Integer> someMap = **new** HashMap<>();

We can obtain a set of key-value pairs:

Set<Map.Entry<String, Integer>> entries = someMap.entrySet();

We can also get the key set associated with the *Map*:

Set<String> keySet = someMap.keySet();

Or we could work directly with the set of values:

Collection<Integer> values = someMap.values();

These each give us an entry point to process those collections by obtaining streams from them:

Stream<Map.Entry<String, Integer>> entriesStream = entries.stream();

Stream<Integer> valuesStream = values.stream();

Stream<String> keysStream = keySet.stream();

### Retrieving a Match

Map<String, String> books = **new** HashMap<>();

books.put( "978-0201633610", "Design patterns : elements of reusable object-oriented software");

books.put( "978-1617291999", "Java 8 in Action: Lambdas, Streams, and functional-style programming");

books.put("978-0134685991", "Effective Java");

Optional<String> optionalIsbn = books.entrySet().stream() .filter(e -> "Effective Java".equals(e.getValue())) .map(Map.Entry::getKey) .findFirst();