**import** java.util.stream.Stream;

**public** **class** Str1 {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Stream<Integer> stream = Stream.*of*(1,2,3,4,5,6,7,8,9);

stream.forEach(p -> System.***out***.println(p));

}

}

**List.stream()**

In the given example, we are creating a stream from the [List](https://howtodoinjava.com/java-arraylist/). The elements in the stream are taken from the List.

List<Integer> list = **new** ArrayList<Integer>();

**for**(**int** i = 1; i< 10; i++){

list.add(i);

}

Stream<Integer> stream = list.stream();

stream.forEach(p -> System.out.println(p));

## Stream Operations

Stream abstraction has a long list of useful functions. Let us look at a few of them.

Before moving ahead, let us build a List of strings beforehand. We will build our examples on this list so that it is easy to relate and understand.

List<String> memberNames = **new** ArrayList<>();

memberNames.add("Amitabh");

memberNames.add("Shekhar");

memberNames.add("Aman");

memberNames.add("Rahul");

memberNames.add("Shahrukh");

memberNames.add("Salman");

memberNames.add("Yana");

memberNames.add("Lokesh");

### Intermediate Operations

Intermediate operations return the stream itself so you can chain multiple methods calls in a row. Let’s learn important ones.

#### 4.1.1. Stream.filter()

The filter() method accepts a [Predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/) to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. [forEach()](https://howtodoinjava.com/java8/java-stream-foreach/" \t "_blank)) on the result.

memberNames.stream().filter((s) -> s.startsWith("A"))

.forEach(System.out::println);

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Stream;

**public** **class** Str2 {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

List<String> memberNames = **new** ArrayList<>();

memberNames.add("Amitabh");

memberNames.add("Shekhar");

memberNames.add("Aman");

memberNames.add("Rahul");

memberNames.add("Shahrukh");

memberNames.add("Salman");

memberNames.add("Yana");

memberNames.add("Lokesh");

Stream<String> s=memberNames.stream();

memberNames.forEach((i)->System.***out***.println(i));

System.***out***.println("filtering operation");

memberNames.stream().filter((i)->i.startsWith("A"))

.forEach(s1->System.***out***.println(s1));

}

}

#### 4.1.2. Stream.map()

The map() intermediate operation converts each element in the stream into another object via the given function.

The following example converts each string into an UPPERCASE string. But we can use map() to transform an object into another type as well.

memberNames.stream().filter((s) -> s.startsWith("A"))

.map(String::toUpperCase)

.forEach(System.out::println);

#### 4.1.2. Stream.sorted()

The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom [Comparator](https://howtodoinjava.com/java/collections/java-comparator/).

memberNames.stream().sorted()

.map(String::toUpperCase)

.forEach(System.out::println);

### 4.2. Terminal operations

Terminal operations return a result of a certain type after processing all the stream elements.

Once the terminal operation is invoked on a Stream, the iteration of the Stream and any of the chained streams will get started. Once the iteration is done, the result of the terminal operation is returned.

#### 4.2.1. Stream.forEach()

The forEach() method helps in iterating over all elements of a stream and perform some operation on each of them. The operation to be performed is passed as the lambda expression.

memberNames.forEach(System.out::println);

#### 4.2.2. Stream.collect()

The collect() method is used to receive elements from a steam and store them in a collection.

List<String> memNamesInUppercase = memberNames.stream().sorted()

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.print(memNamesInUppercase);

#### 4.2.3. Stream.match()

Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.

**boolean** matchedResult = memberNames.stream()

.anyMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult); *//true*

matchedResult = memberNames.stream()

.allMatch((s) -> s.startsWith("A"));

System.out.println(matchedResult); *//false*

#### 4.2.4. Stream.count()

The count() is a terminal operation returning the number of elements in the stream as a long value.

**long** totalMatched = memberNames.stream()

.filter((s) -> s.startsWith("A"))

.count();

System.out.println(totalMatched); *//2*

#### 4.2.5. Stream.reduce()

The reduce() method performs a reduction on the elements of the stream with the given function. The result is an [Optional](https://howtodoinjava.com/java8/java-8-optionals-complete-reference/) holding the reduced value.

In the given example, we are reducing all the strings by concatenating them using a separator #.

Optional<String> reduced = memberNames.stream()

.reduce((s1,s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);