10 Ways to Create a Stream in Java

The Stream API, introduced in Java 8, it is used to process collections of objects. Stream is a sequence of objects, that supports many different methods which can be pipe lined to produce the desired result.

The features of Java stream are –

- A stream is not a data structure alternatively it takes input from the Collections, Arrays or I/O channels.
- A Streams does not change the original data structure, they only provide the result as the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipe lined. Terminal operation mark the end of the stream and return the result.

Different way to create Streams:

1. **Using Collection**
   Approach:
   1. Get the collection
   2. Construct a Sequential Stream from the collection using Collection.stream() method
   3. Print the Stream

   Below is the implementation of the above approach:

   Program:

```
// Java program to create Stream from Collections

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream(List<T> list)
    {

        // Create stream object with the List
        Stream<T> stream = list.stream();

        // Iterate list first to last element
        Iterator<T> it = stream.iterator();

        // Iterate stream object
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
    }

    public static void main(String[] args)
    {

        // Create ArrayList of String
        List<String> list = new ArrayList<>();

        // Add element in list
        list.add("Geeks");
        list.add("for");
        list.add("Geeks");

        // Get the Stream from the List
        getStream(list);
    }
}
```

Output:

```
Geeks for Geeks
```

2. **Create a stream from specified values**

   Stream.of(T…t) method can be used to create a stream with the specified t values, where t are the elements. This method returns a sequential Stream containing the t elements.

   Below is the implementation of the above approach:

   Program:

```
// Java program to create Stream from values
```

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

class GFG {

    // Function convert a List into Stream
    private static void getStream()
    {

        // Create a stream from specified values
        Stream<Integer> stream
            = Stream.of(1, 2,
                        3, 4,
                        5, 6,
                        7, 8,
                        9);

        // Displaying the sequential ordered stream
        stream.forEach(p -> System.out.print(p + " "));
    }

    public static void main(String[] args)
    {

        // Get the Stream from the values
        getStream();
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 9
```

3. **Create stream from an array:**
   The Stream.of() and Arrays.stream() are two commonly used methods for creating a sequential stream from a specified array. Both these methods returns a Stream when called with a non-primitive type T.
   Integer array

   - **Create stream using Arrays.stream()**
     Program:

```
// Java program to create Stream from Collections

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream(T[] arr)
    {
```

```
        // Create stream from an array
        // using Arrays.stream()
        Stream<T> streamOfArray
            = Arrays.stream(arr);

        // Iterate list first to last element
        Iterator<T> it
            = streamOfArray.iterator();

        // Iterate stream object
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
    }

    public static void main(String[] args)
    {

        // Get the array
        String[] arr
            = new String[] { "a", "b", "c" };

        // Get the Stream from the Array
        getStream(arr);
    }
}
```

Output:

```
a b c
```

- **Create stream using Stream.of()**
  A non interfering action to be perform on elements as they are consumed from the stream and returns also a new stream.

Program:

```
// Java program to create Stream from Collections

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream(T[] arr)
    {

        // Create stream from an array
        // using Stream.of()
```

```
            Stream<T> streamOfArray = Stream.of(arr);

            // Iterate list first to last element
            Iterator<T> it = streamOfArray.iterator();

            // Iterate stream object
            while (it.hasNext()) {
                System.out.print(it.next() + " ");
            }
        }

        public static void main(String[] args)
        {

            // Get the array
            String[] arr
                = new String[] { "a", "b", "c" };

            // Get the Stream from the Array
            getStream(arr);
        }
    }
```

Output:

```
a b c
```


4. **Create an empty stream using Stream.empty()**

   The empty() method is used upon creation to avoid returning null for streams with no element.

   Program:

```
// Java program to create empty Stream

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

class GFG {

    // Function convert a List into Stream
    private static void getStream()
    {

        // Create stream from an array using Stream.empty()
```

```
            Stream<String> streamOfArray
                = Stream.empty();

            // Iterate list first to last element
            Iterator<String> it
                = streamOfArray.iterator();

            // Iterate stream object
            while (it.hasNext()) {
                System.out.print(it.next() + " ");
            }
        }

        public static void main(String[] args)
        {

            // Get the empty Stream
            getStream();
        }
    }
```

Output:

5. **Create a Stream using Stream.builder()**

   The builder() method is used when the desired type should be additionally specified in the right part of the statement, otherwise the build() method will create an instance of the Stream.

   Program:

```
// Java program to create Stream from Collections

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream()
    {

        // Create stream using Stream builder()
        Stream.Builder<String> builder
            = Stream.builder();

        // Adding elements in the stream of Strings
        Stream<String> stream = builder.add("a")
                                       .add("b")
                                       .add("c")
                                       .build();

        // Iterate list first to last element
```

```
        Iterator<String> it = stream.iterator();

        // Iterate stream object
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
    }

    public static void main(String[] args)
    {

        // Get the Stream using Builder
        getStream();
    }
}
```

Output:

```
a b c
```

6. **Create an infinite Stream using Stream.iterate()**

   The iterate() method returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed. In below example, First element of the resulting stream is a first parameter of the iterate method. For creating every following element the function is applied to the previous element. In the example below the second element will be 4.

   Program:

```
// Java program to create infinite Stream
// using Stream.iterate() method

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

class GFG {

    // Function convert a List into Stream
    private static <T> void
    getStream(int seedValue, int limitTerms)
    {

        // Create infinite stream
        // using Stream.iterate() method
        Stream.iterate(seedValue,
                       (Integer n) -> n * n)
            .limit(limitTerms)
```

```
                .forEach(System.out::println);
    }

    public static void main(String[] args)
    {

        // Get the seed value
        int seedValue = 2;

        // Get the limit for number of terms
        int limitTerms = 5;

        // Get the Stream from the function
        getStream(seedValue, limitTerms);
    }
}
```

Output:

```
2
4
16
256
65536
```

7. **Create an infinite Stream using Stream.generate() method**
   The generate() method accepts a Supplier for generating elements and the resulting stream is infinite. So to restrict it, specify the desired size or the generate() method will work until it reaches the memory limit.

   Program:

```
// Java program to create infinite Stream
// using Stream.generate() method

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream(int limitTerms)
    {

        // Create infinite stream
        // using Stream.generate() method
        Stream.generate(Math::random)
            .limit(limitTerms)
            .forEach(System.out::println);
    }

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

```
        // Get the limit for number of terms
        int limitTerms = 5;

        // Get the Stream from the function
        getStream(limitTerms);
    }
}
```

Output:

```
0.2293502475696314
0.5650334795948209
0.3418138293253522
0.36831074763500116
0.4864408670097241
```

8. **Create stream from a Pattern using Predicate**

   In java 8, the Predicate asPredicate() method of Pattern creates a predicate boolean-valued function that is used for pattern matching.

   Program:

```
// Java program to create Stream from Collections

import java.util.*;
import java.util.stream.*;
import java.util.regex.Pattern;

class GFG {

    // Function convert a List into Stream
    private static void
    getStream(List<String> list, Pattern p)
    {

        list.stream()
            .filter(p.asPredicate())
            .forEach(System.out::println);
    }

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

```
            // Create ArrayList of String
            // that is backed by the specified array
            List<String> list
                = Arrays
                      .asList("Geeks",
                              "For",
                              "Geek",
                              "GeeksForGeeks",
                              "A Computer Portal");

            // Get the pattern
            Pattern p = Pattern.compile("^G");

            // Get the Stream from the List matching Pattern
            getStream(list, p);
        }
    }
```

Output:

```
Geeks
Geek
GeeksForGeeks
```


9. **Create stream from Iterator**

   Iterators, in Java, are used in Collection Framework to retrieve elements one by one. Spliterator is the key to create the sequential stream. Hence in this method also, Spliterator is used. But in this method, the source of Spliterator is set to an Iterable created from the Iterator. So first the Iterable is created from the Iterator. Then the Spliterator is passed to the stream() method directly as Iterable.spliterator().

   Program:

```
// Java program to create Stream from Collections

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream(Iterator<T> itr)
```

```
    {

        // Convert the iterator into a Spliterator
        Spliterator<T> spitr
            = Spliterators
                  .spliteratorUnknownSize(itr,
                                          Spliterator.NONNULL);

        // Convert spliterator into a sequential stream
        Stream<T> stream
            = StreamSupport.stream(spitr, false);

        // Iterate list first to last element
        Iterator<T> it = stream.iterator();

        // Iterate stream object
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
    }

    public static void main(String[] args)
    {

        // Get the Iterator
        Iterator<String> iterator = Arrays
                                        .asList("a", "b", "c")
                                        .iterator();

        // Get the Stream from the Iterator
        getStream(iterator);
    }
}
```

Output:

```
a b c
```

10. **Create stream from Iterable**

    Iterable interface is designed keeping in mind and does not provide any stream() method on its own. Simply it can be passed into StreamSupport.stream() method, and get a Stream from the given Iterable object. It is easier to turn an Iterable into a Stream. Iterable has a default method spliterator(), which can be used to get a Spliterator instance, which can be in turn then converted to a Stream.

    Note: The Iterable is not a instance of Collection, this method internally calls StreamSupport.stream() to get a sequential Stream from Spliterator else it simply calls Collection.stream() method.

    Program:

```
// Java program to create Stream from Collections

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

class GFG {

    // Function convert a List into Stream
    private static <T> void getStream(Iterable<T> iterable)
    {

        // Convert the iterator into a Stream
        Stream<T> stream
            = StreamSupport
                  .stream(iterable.spliterator(),
                          false);

        // Iterate list first to last element
        Iterator<T> it = stream.iterator();

        // Iterate stream object
        while (it.hasNext()) {
            System.out.print(it.next() + " ");
        }
    }

    public static void main(String[] args)
    {

        // Get the Iterable
        Iterable<String> iterable
            = Arrays.asList("a", "b", "c");

        // Get the Stream from the Iterable
        getStream(iterable);
    }
}
```

Output:

```
a b c
```

Recommended Posts:

Different ways to create objects in Java

Different ways to create an Object in C#

Different ways to create Pandas Dataframe

Difference between Stream.of() and Arrays.stream() method in Java