**Stream.distinct()  :**

**distinct()** returns a stream consisting of distinct elements in a stream. distinct() is the method of **Stream** interface. This method uses *hashCode()* and *equals()* methods to get distinct elements.

**Syntax :**

Stream<**T**> distinct()

Where, Stream is an interface and the function

returns a stream consisting of the distinct

elements.

EXAMPLE :

import java.util.\*;

public class Disticts{

    public static void main(String[] args)

    {

        // Creating a list of integers

        List<Integer> list = Arrays.asList(1, 1, 2, 3, 3, 4, 5, 5);

        System.out.println("The distinct elements are :");

        // Displaying the distinct elements in the list

        list.stream().distinct().forEach(System.out::println);

    }

}

Output :

The distinct elements are :

1

2

3

4

5

**Stream peek() :**

Stream peek(Consumer action) returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

**Syntax :**

**Stream<T> peek(Consumer<? super T> action)**

Where, Stream is an interface and T is the type of

stream elements. **action** is a [non-interfering](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterference) action

to perform on the elements as they are consumed

from the stream and the function returns the new stream.

Example :

import java.util.\*;

class Peeksss{

    // Driver code

    public static void main(String[] args)

    {

        // Creating a list of Integers

        List<Integer> list = Arrays.asList(0, 2, 4, 6, 8, 10);

        // Using peek without any terminal

        // operation does nothing. Hence this

        // code will produce no output.

        list.stream().peek(System.out::println);

    }

}

Output:

EX2 :

import java.util.\*;

class GFG {

    // Driver code

    public static void main(String[] args)

    {

        // Creating a list of Integers

        List<Integer> list = Arrays.asList(0, 2, 4, 6, 8, 10);

        // Using peek with count(), which

        // is a terminal operation

        list.stream().peek(System.out::println).count();

    }

}

Output :

0

2

4

6

8

10

**IntStream parallel() :**

**IntStream parallel()** is a method in java.util.stream.IntStream. This method returns a parallel IntStream, i.e, it may return itself, either because the stream was already present, or because the underlying stream state was modified to be parallel.

IntStream parallel() is an intermediate operation. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

Syntax

IntStream parallel()

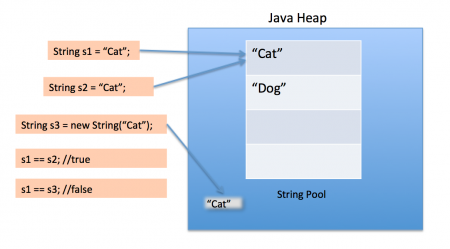
Where, IntStream is a sequence of

primitive int-valued elements and the function

returns a parallel IntStream.

**String Pool in Java**

Here is a diagram which clearly explains how String Pool is maintained in  [java heap](https://www.journaldev.com/4098/java-heap-space-vs-stack-memory) space and what happens when we use different ways to create Strings.

[](https://cdn.journaldev.com/wp-content/uploads/2012/11/String-Pool-Java1.png)

**Stream filter() :**

**Stream filter(Predicate predicate)** returns a stream consisting of the elements of this stream that match the given predicate. such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.

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

Where, Stream is an interface and T is the

type of the input to the predicate.

The function returns the new stream.

Example :

import java.util.stream.Stream;

class Filters {

    public static void main(String[] args)

    {

        // Creating a Stream of Strings

        Stream<String> stream = Stream.of("sreemas", "foR",

                                          "Siddhu", "patros");

        // elements ending with s

        stream.filter(str -> str.endsWith("s"))

            .forEach(System.out::println);

    }

}

Ouput :

sreemas

patros

**Stream flatMap() :**

**Stream flatMap(Function mapper)** returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Stream flatMap(Function mapper) is an intermediate operation. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

**Syntax :**

**<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**

where, R is the element type of the new stream.

Stream is an interface and T is the type

of stream elements. mapper is a stateless function

which is applied to each element and the function.

**Min and Max in Streams ;**

We use the min() and max() methods to find the min & max value in streams. These methods are used for finding min & max values in different types of streams such as stream of chars, strings, dates etc. We just have to change the parameter that we pass in this method based on the type of stream.

For example:  
max(Comparator.comparing(Integer::valueOf)): To get the max value from stream of numbers.  
max(Comparator.comparing(LocalDate::toEpochDay)): To get the max date from stream of dates.

import java.util.Comparator;

import java.util.stream.\*;

public class JavaExample{

public static void main(String args[]) {

//getting max number

Integer maxnum = Stream.of(10, 13, 4, 9, 2, 100)

.max(Comparator.comparing(Integer::valueOf))

.get();

//getting min number

Integer minnum = Stream.of(10, 13, 4, 9, 2, 100)

.min(Comparator.comparing(Integer::valueOf))

.get();

.get();

System.out.println("Max number is: " + maxnum);

System.out.println("Min number is: " + minnum);

}

}

Output :

Max number is : 100

Min number is : 2

To create a userdefined unchecked exception :

public class RegistrationServiceClient { public static void main(String[] args)

{

RegistrationService service = new RegistrationService();

try {

service.validateEmail("abc@gmail.com");

}

catch (EmailNotUniqueException e)

{

*// logging and handling the situation*

}

}

}

Output :

mynotes.custom.checked.exception.EmailNotUniqueException: Email Already Registered at mynotes.custom.checked.exception.RegistrationService.validateEmail(RegistrationService.java:12) at mynotes.custom.checked.exception.RegistrationServiceClient.main(RegistrationServiceClient.ja