Streams and collections are both powerful features in Java for working with collections of data, but they have distinct characteristics and serve different purposes. Here's a comparison between streams and collections in Java:

Collections:

* Collections are data structures that hold a collection of elements in memory, such as lists (ArrayList, LinkedList), sets (HashSet, TreeSet), and maps (HashMap, TreeMap).
* Collections are mutable, meaning you can add, remove, or modify elements within the collection.
* Collections provide direct access to elements using methods like **get()** (for lists) or **contains()** (for sets and maps).
* Collections are eager in nature, meaning they store all the elements in memory at once.
* Collections are typically used for storing and manipulating data in memory.
* Collections provide various operations like sorting, filtering, and transforming the data using methods defined in their respective interfaces.

Streams:

* Streams are a sequence of elements that can be processed in a pipeline-like manner.
* Streams are based on the concept of functional programming and enable functional-style operations on data.
* Streams are immutable, meaning they do not modify the underlying data source.
* Streams are lazily evaluated, meaning they process elements on-demand as the stream operations are performed.
* Streams provide a wide range of powerful operations like filtering, mapping, sorting, reducing, and aggregating data.
* Streams allow for parallel execution, enabling efficient processing of large datasets across multiple threads.
* Streams can be created from various data sources like collections, arrays, I/O channels, and generators.
* Streams provide a declarative and concise way of expressing data processing operations.

Here's an example that illustrates the difference between collections and streams:

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamsVsCollectionsExample {

public static void main(String[] args) {

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

// Collections: Filtering even numbers

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println("Even Numbers (Collections): " + evenNumbers);

// Streams: Filtering even numbers

List<Integer> evenNumbersStream = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

System.out.println("Even Numbers (Streams): " + evenNumbersStream);

}

}

Java 8 introduced several built-in functional interfaces in the **java.util.function** package to support functional programming and lambda expressions. These functional interfaces represent common function types and provide a foundation for working with lambda expressions and method references. Here are some of the commonly used functional interfaces introduced in Java 8:

1. **Supplier<T>**: Represents a supplier of results, with no input arguments. It provides a single method **T get()** to retrieve a result.
2. **Consumer<T>**: Represents an operation that accepts a single input argument of type **T** and returns no result. It provides a single method **void accept(T t)** to perform the operation.
3. **Function<T, R>**: Represents a function that takes an argument of type **T** and produces a result of type **R**. It provides a single method **R apply(T t)** to apply the function.
4. **Predicate<T>**: Represents a predicate (boolean-valued function) that takes an argument of type **T** and returns a boolean result. It provides a single method **boolean test(T t)** to test the predicate.
5. **UnaryOperator<T>**: Represents an operation on a single operand of type **T** that produces a result of the same type **T**. It extends **Function<T, T>**.
6. **BinaryOperator<T>**: Represents an operation on two operands of type **T** that produces a result of the same type **T**. It extends **BiFunction<T, T, T>**.
7. **BiPredicate<T, U>**: Represents a predicate that takes two arguments of types **T** and **U** and returns a boolean result. It provides a single method **boolean test(T t, U u)** to test the predicate.

These are just a few examples of the functional interfaces introduced in Java 8. The **java.util.function** package provides a range of other functional interfaces to handle various scenarios, including **BiFunction**, **BiConsumer**, **BiPredicate**, and specialized interfaces for primitive types.

Functional interfaces serve as the building blocks for working with lambda expressions and provide a standardized way to express common functional patterns. They can be used as method parameters, return types, or variables to represent behavior that can be passed around and composed easily.