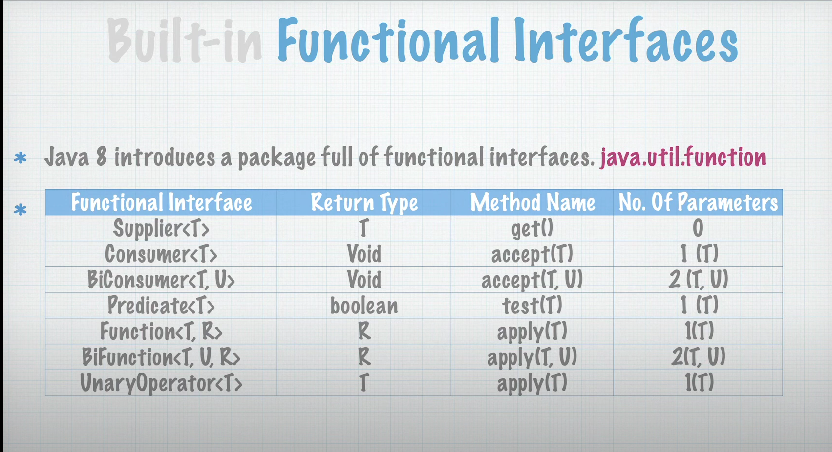
F

* Supplier<T>: generate an objects of type T (no input/parameters is required)
* Consumer<T>: return void
* Predicate<T>: the most uses as conditions checker
* Function<T,R>: take T as input and R a return value





### ****Supplier****

* **Purpose**: A Supplier<T> is used to generate or supply values when needed. It doesn't take any input but returns a value of type T.
* **Functional Method**: T get()
* **Common Usage**: Often used when you need to lazily generate values, such as creating objects or generating random numbers.

**Example: Supplier Directly in Stream**

import java.util.stream.Stream;

public class SupplierDirectExample {

public static void main(String[] args) {

// Using Supplier directly in Stream.generate() to produce random numbers

Stream.generate(() -> Math.random()) **// It doesn't take any input but returns a value of type T**

.limit(5) // Limit to 5 elements

.forEach(System.out::println); // Print each element

}

}

**Consumer**

* **Purpose**: A Consumer<T> represents an operation that takes a single input argument and performs some action, but doesn't return any result.
* **Functional Method**: void accept(T t)
* **Common Usage**: Commonly used for operations like printing values, modifying objects, or performing side-effects on elements in a stream.

**Example: Consumer Directly in Stream**

import java.util.Arrays;

import java.util.List;

public class ConsumerDirectExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Using Consumer directly in forEach() to print each name

names.stream()

.forEach(name -> System.out.println(name)); // Print each element

// that takes a single input argument and performs some action, but doesn't return any result.

} }

### ****Predicate****

* **Purpose**: A Predicate<T> is used to test a condition on an input argument and returns a boolean (true or false).
* **Functional Method**: boolean test(T t)
* **Common Usage**: Frequently used in stream operations to filter elements based on certain criteria.

import java.util.Arrays;

import java.util.List;

public class PredicateDirectExample {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Using Predicate directly in filter() to select names starting with "A"

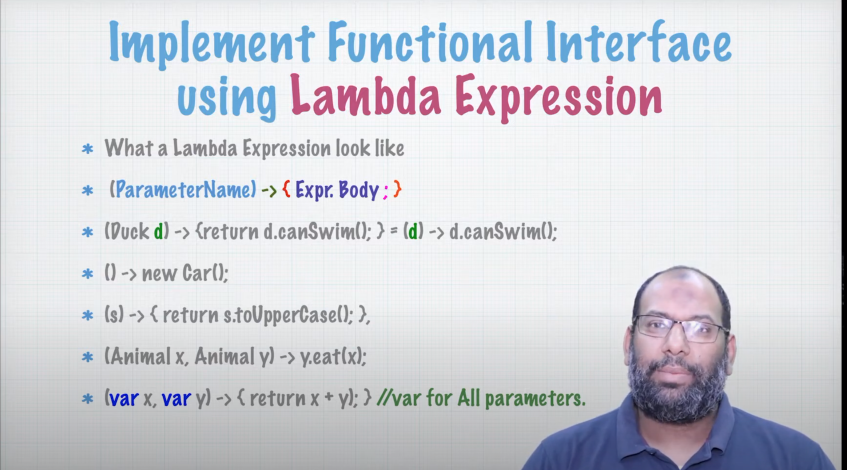
names.stream()

.filter(name -> name.startsWith("A")) // Filter names that start with "A"

.forEach(System.out::println); // Print each remaining element

}

}



import java.util.function.Consumer;

public class LambdaExample {

public static void main(String[] args) {

// Define a lambda expression (block of code) that will be executed later

Consumer<String> printMessage = message -> {

System.out.println("Message: " + message);

};

// Pass the lambda as a parameter to a method

executeLater(printMessage, "Hello, World!");

}

// Method that accepts a lambda (Consumer) and executes it later

public static void executeLater(Consumer<String> action, String parameter) {

System.out.println("Preparing to execute...");

// Execute the lambda passed in as a parameter

action.accept(parameter);

System.out.println("Execution finished.");

}

}

**Shortened Version of the Code:**

import java.util.function.Consumer;

public class LambdaExample {

public static void main(String[] args) {

// Directly pass the lambda expression to the method

executeLater(message -> System.out.println("Message: " + message), "Hello, World!");

}

// Method that accepts a lambda (Consumer) and executes it later

public static void executeLater(Consumer<String> action, String parameter) {

System.out.println("Preparing to execute...");

// Execute the lambda passed in as a parameter

action.accept(parameter);

System.out.println("Execution finished.");

}

}

Streams:

A Strems is a representation of the collection (Array, List, Map<>, Set<>) that facilitates (make it **easier) the processing of working on data >**

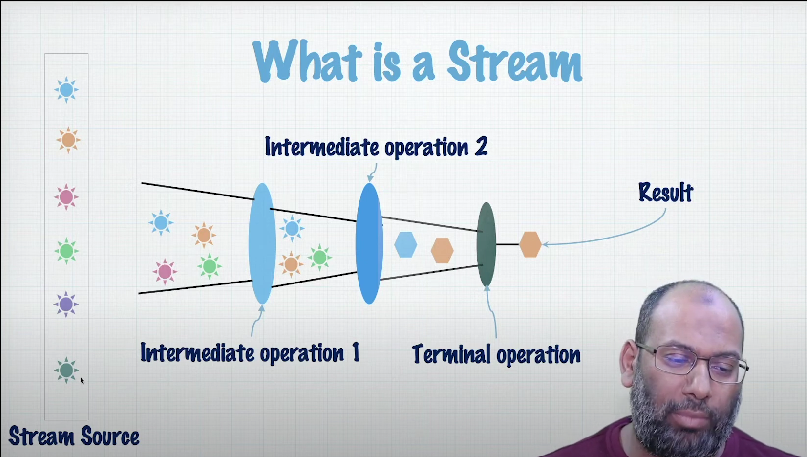
**It doesn't store the data**

**In Java, the concept of a Stream is very much like a "river" that flows through various stages, where you can apply different operations (like filtering, mapping, reducing, etc.) to transform and process the data.**

Functional Programming: You specify what is required, NOT how to achieve it

Package:

java.util.stream



Stream Source: Array, List, Set, Map….

Stream is

1- lightweight object.

2- can't be reused once processed

3- you specify What not How (Stream will do it for u in background in JVM)

4- doesn't store Data.

**Map - Filter - Reduce**

Map function is

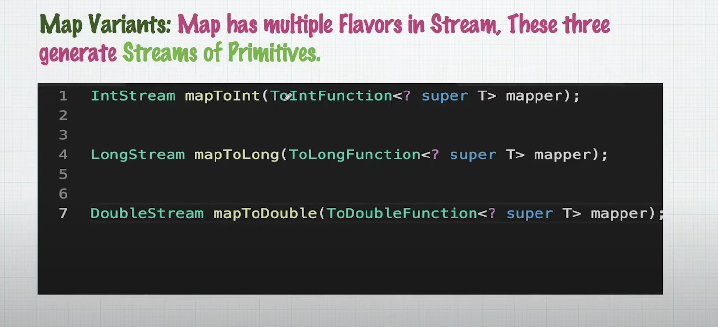
- Intermedia Operation (it will give a new stream)

- applied to each element of the Stream.

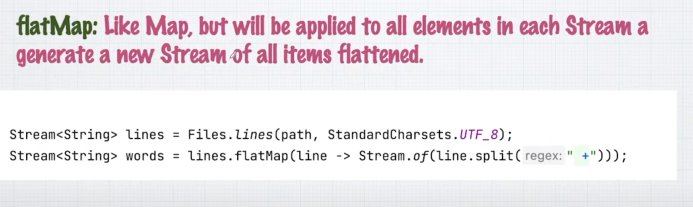
- Used to convert the Stream to a new Stream other/Same type.

- Generates a new Stream of the mapped type

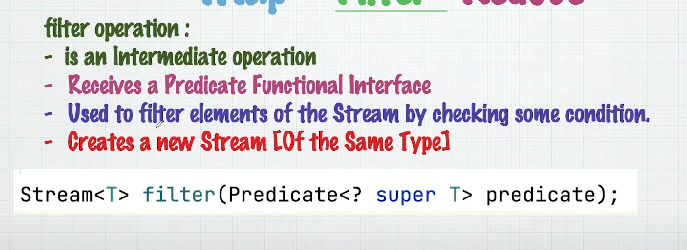
<R> Stream<**R**> map(Function<? Super T, extends **R**> mapper);



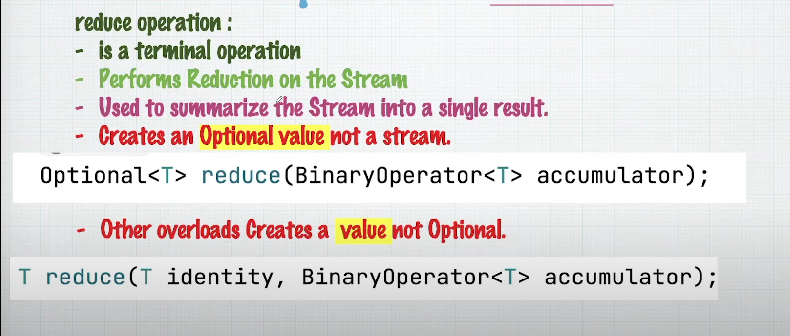
* FlatMap:



**Filter:**

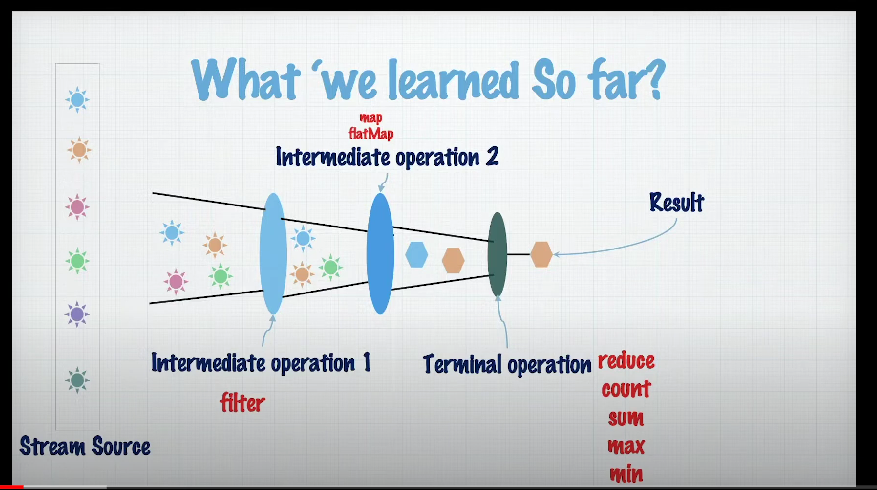


**Reduce:**



Reduce ready defined operation:

* Sum()
* Count()
* Average()
* Min()
* Max()



### How These Cover Most Use Cases

1. **Filtering Data**:
   * filter allows you to select only the elements that meet certain criteria. This can be combined with map and reduce for powerful data pipelines.
2. **Transforming Data**:
   * map is used to transform or project each element of the stream. Whether you're converting types, modifying values, or extracting information, map is versatile enough to handle it.
3. **Aggregating Data**:
   * reduce lets you condense a stream of elements into a single summary result, whether it's a sum, average, maximum, or concatenated string.

 **filter**: Select elements based on conditions.

 **map**: Transform elements into another form: It takes a Function<T, R> as a parameter, where T is the type of the input elements and R is the type of the result.

 **reduce**: Aggregate elements into a single result.

Yes, you're correct in noting that both Java Streams and LINQ (Language Integrated Query) in C# are powerful tools for querying and manipulating collections, but they differ in the number and variety of built-in functions they offer.

### Similarities Between Java Streams and LINQ

1. \*\*Declarative Approach\*\*:

- Both Java Streams and LINQ provide a declarative way to query and manipulate collections, allowing you to focus on \*what\* you want to do rather than \*how\* to do it.

2. \*\*Core Functions\*\*:

- Both have core operations that are very similar:

- \*\*Filtering\*\*:

- Java Streams: `filter()`

- LINQ: `Where()`

- \*\*Mapping/Transforming\*\*:

- Java Streams: `map()`

- LINQ: `Select()`

- \*\*Reducing/Aggregating\*\*:

- Java Streams: `reduce()`

- LINQ: `Aggregate()`, `Sum()`, `Max()`, `Min()`

3. \*\*Lazy Evaluation\*\*:

- Both Java Streams and LINQ utilize lazy evaluation, meaning that the operations are not executed until a terminal operation is performed (e.g., `collect()` in Streams or `ToList()` in LINQ).

4. \*\*Functional Programming\*\*:

- Both frameworks embrace functional programming principles, allowing you to pass functions (predicates, selectors, etc.) as arguments to various methods.

### Differences in Functionality

1. \*\*Number of Built-in Functions\*\*:

- \*\*LINQ\*\*: LINQ has a larger and more diverse set of functions compared to Java Streams. LINQ provides a wide range of operations, such as `GroupBy()`, `Join()`, `Zip()`, `Distinct()`, `Union()`, `Intersect()`, `Except()`, `OrderBy()`, `ThenBy()`, and many more.

- \*\*Java Streams\*\*: While Java Streams provide the core functions (`map`, `filter`, `reduce`, `flatMap`, etc.), they don't include as many built-in functions as LINQ. However, many of the more complex operations available in LINQ can be achieved in Streams by combining the existing functions or by using custom code.

2. \*\*Complex Queries\*\*:

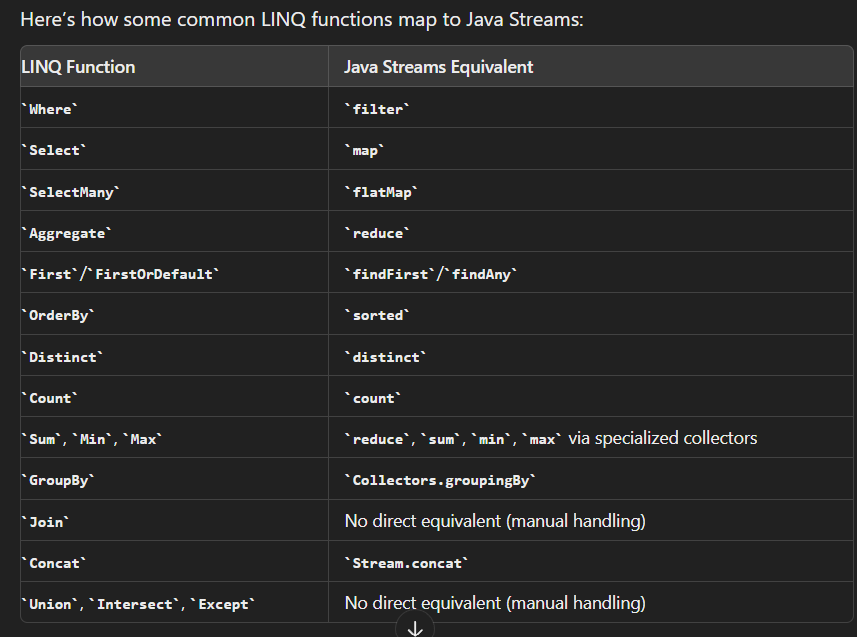
- \*\*LINQ\*\*: Allows more complex and sophisticated queries directly within its framework, often with simpler syntax. For instance, operations like `GroupBy` and `Join` are straightforward and integral to LINQ.

- \*\*Java Streams\*\*: While you can perform grouping and joining operations, they are typically less straightforward and require more manual setup (e.g., using `Collectors.groupingBy()` for grouping or merging streams manually for joins).

3. \*\*Integration with Data Sources\*\*:

- \*\*LINQ\*\*: LINQ is not limited to in-memory collections but can also query data from various data sources like databases (LINQ to SQL, Entity Framework), XML (LINQ to XML), and more.

- \*\*Java Streams\*\*: Streams are generally used for processing in-memory collections or arrays. There isn't a direct, built-in way to use Streams with a database or external data source, though you can convert data from those sources into a Stream.

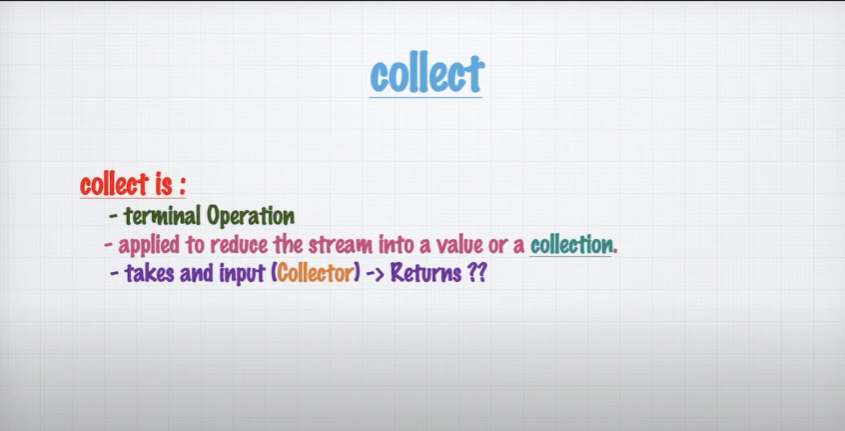


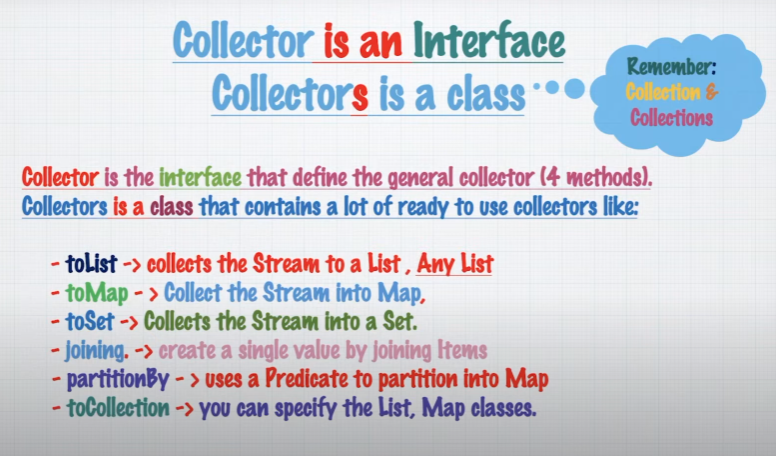
### Summary

- \*\*Java Streams\*\*: Offers a core set of functions like `map`, `filter`, and `reduce` that cover most basic use cases. While fewer in number compared to LINQ, Streams are powerful and can be extended or combined to handle more complex scenarios.

- \*\*LINQ\*\*: Provides a more extensive set of built-in functions that cover a broader range of query and manipulation tasks, making it easier to perform complex operations directly within the framework.

While Java Streams and LINQ are conceptually similar, LINQ's broader range of built-in functions gives it more out-of-the-box versatility for complex queries and data manipulation. However, Java Streams are still powerful and can achieve similar results with some additional effort or by combining existing methods.





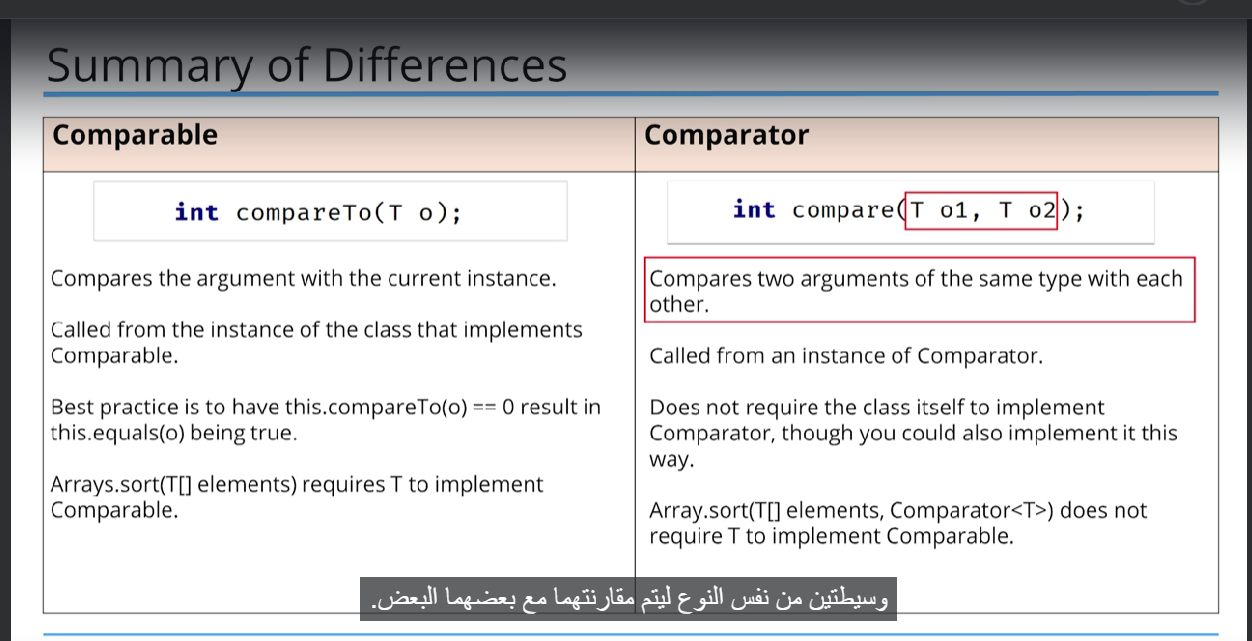
The Comparator Interface

* The Comparator Interface is similar to the Comparable interface, and the two va often be confused with each other.
* Its declaration and primary abstract method are shown here, in comparison to

Comparable.

* Notice that the method names are different, compare vs. compareTo.
* The compare method takes two arguments vs. one for compareTo, meaning that it will
* The **compare** the two arguments (Parameters) vs. one for compareTo, meaning that it will compare the two arguments to one another and not one **object** to the instance itself.
* I'll review Comparator in code, but in a slightly manufactured way.
* It's common practice to include a Comparator as a nested class.

|  |  |
| --- | --- |
| Comparator | Comparable |
| Public interface Comparator<T> {  int compare(T o1, T o2);  } | Public interface Comparable <T> {  int compareTo(To);  } |



Yes, you can compare Java functional interfaces to C# delegates to help explain their roles. Here's how you can relate the Java interfaces listed in your image to C# delegates:

1. Predicate<T>:

- \*\*Java\*\*: This interface checks if an object of type T fulfills a certain criterion and returns a boolean value.

- \*\*C#\*\*: The equivalent delegate is `Predicate<T>`, which takes an object of type T and returns a boolean.

2. \*\*Consumer<T>\*\*:

- \*\*Java\*\*: This interface processes an object of type T but does not return any result.

- \*\*C#\*\*: The equivalent delegate is `Action<T>`, which takes an object of type T and returns void.

3. \*\*Consumer<T, U>\*\*:

- \*\*Java\*\*: This interface processes two objects of types T and U and does not return any result.

- \*\*C#\*\*: The equivalent delegate is `Action<T, U>`, which takes two objects of types T and U and returns void.

4. \*\*Function<T, R>\*\*:

- \*\*Java\*\*: This interface processes an object of type T and returns an object of type R.

- \*\*C#\*\*: The equivalent delegate is `Func<T, R>`, which takes an object of type T and returns an object of type R.

5. \*\*BiFunction<T, U, R>\*\*:

- \*\*Java\*\*: This interface processes two objects of types T and U and returns an object of type R.

- \*\*C#\*\*: The equivalent delegate is `Func<T, U, R>`, which takes two objects of types T and U and returns an object of type R.

6. \*\*UnaryOperator<T>\*\*:

- \*\*Java\*\*: This interface is a specialization of `Function<T, T>`, processing an object of type T and returning an object of the same type T.

- \*\*C#\*\*: The equivalent delegate can be represented as `Func<T, T>`, which takes and returns an object of type T.

7. \*\*BinaryOperator<T>\*\*:

- \*\*Java\*\*: This interface is a specialization of `BiFunction<T, T, T>`, processing two objects of the same type T and returning an object of type T.

- \*\*C#\*\*: The equivalent delegate can be represented as `Func<T, T, T>`, which takes two objects of type T and returns an object of type T.

These comparisons should help in understanding the functional interfaces in Java in terms of similar constructs in C#.

### Similarities

Despite these differences, both functional interfaces in Java and delegates in C# provide a way to:

* Pass methods as parameters.
* Implement callback mechanisms.
* Support lambda expressions and anonymous methods.