Difference Between Function and Predicate Interfaces in Java 8

Both **Function** and **Predicate** are functional interfaces introduced in Java 8. They are part of the **java.util.function** package, but they serve different purposes. Here's a breakdown of their differences:

Feature	`Function<T, R>`	`Predicate<T>`
Purpose	Represents a function that accepts one argument and produces a result.	Represents a boolean-valued function (condition) that accepts one argument.
Method	`R apply(T t)`	`boolean test(T t)`
Input Type	Takes one input of type `T`.	Takes one input of type `T`.
Return Type	Returns a value of type `R`.	Returns a boolean (`true`/`false`).
Use Case	Used when you want to transform or map a value to another value (e.g., transforming an object or mapping one type to another).	Used when you want to test a condition (filtering, validation, etc.).
Chaining	You can chain multiple `Function`s using `andThen()` and `compose()`.	You can chain multiple `Predicate`s using `and()`, `or()`, and `negate()`.

Key Points:

- **Predicate** is used to encapsulate the condition (checking if a list is empty).
- **Reusability**: You can reuse the isListEmpty predicate to check multiple lists.

- **Functional Programming**: This style aligns with Java 8's functional programming paradigm, where behavior (the condition for emptiness) is passed as a lambda expression.

1. **Predicate**: Represents a condition to filter the elements. In this case, it checks if the string is non-empty (!str.isEmpty()).
2. **Streams**: Provide a powerful way to perform transformations and operations on collections, like filtering, mapping, and reducing.
3. **Filter Operation**: Only elements that satisfy the predicate pass through the filter.
4. **Collectors.toList()**: Collects the filtered elements back into a List.

Why Use Predicate?

- **Predicate** is useful when you want to encapsulate some condition for filtering or validation.
- **Reusability**: You can reuse the Predicate across multiple filters or validations.
- **Functional Programming**: It's part of the functional style of programming introduced in Java 8, where behavior is passed around as parameters.

Method References in Java 8

Method references in Java 8 provide a shorthand way to refer to methods or constructors without executing them. They are a form of lambda expression and make the code more readable by eliminating verbosity.

Types of Method References:

1. **Static Method Reference**: Refers to a static method.
    - Syntax: ClassName::staticMethod
2. **Instance Method Reference of a Particular Object**: Refers to an instance method of a specific object.

- Syntax: instance::instanceMethod

3. **Instance Method Reference of an Arbitrary Object of a Particular Type**: Refers to an instance method of an arbitrary object of a specific type.
    - Syntax: ClassName::instanceMethod
4. **Constructor Reference**: Refers to a constructor to create a new object.
    - Syntax: ClassName::new