Interfaces Funcionales en Java

El paquete `java.util.function` en Java proporciona una colección de interfaces funcionales diseñadas para apoyar la programación funcional, introducida en Java 8. Estas interfaces tienen cada una un método abstracto único, lo que las hace compatibles con expresiones lambda y referencias de método. Esta respuesta explica cómo usar algunas de las interfaces funcionales más comunes en este paquete—`Function<T, R>`, `Predicate<T>`, `Consumer<T>` y `Supplier<T>`—con ejemplos prácticos.

¿Qué son las Interfaces Funcionales?

Una interfaz funcional es una interfaz con exactamente un método abstracto. El paquete `java.util.function` ofrece interfaces funcionales predefinidas para tareas comunes, por lo que no necesitas crear las tuyas propias. Estas interfaces se utilizan ampliamente con expresiones lambda, referencias de método y la API de Streams para escribir código conciso y expresivo.

Aquí te explicamos cómo usar las interfaces clave:

---

1. `Function<T, R>`: Transformar Entrada en Salida

La interfaz `Function<T, R>` representa una función que toma una entrada de tipo `T` y produce una salida de tipo `R`. Su método abstracto es `apply`.

Ejemplo: Obtener la Longitud de una Cadena

```
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();
        System.out.println(stringLength.apply("Hello")); // Muestra: 5
    }
}
```

- **Explicación**: La expresión lambda `s -> s.length()` define una `Function` que toma una `String` (`T`) y devuelve un `Integer` (`R`). El método `apply` ejecuta esta lógica.

---

2. `Predicate<T>`: Probar una Condición

La interfaz `Predicate<T>` representa una función de valor booleano que toma una entrada de tipo `T`. Su método abstracto es `test`.

Ejemplo: Verificar si un Número es Par

```
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println(isEven.test(4)); // Muestra: true
        System.out.println(isEven.test(5)); // Muestra: false
    }
}
```

- **Explicación**: La lambda `n -> n % 2 == 0` define un `Predicate` que devuelve `true` si la entrada es par. El método `test` evalúa esta condición.

---

3. `Consumer<T>`: Realizar una Acción

La interfaz `Consumer<T>` representa una operación que toma una entrada de tipo `T` y no devuelve ningún resultado. Su método abstracto es `accept`.

Ejemplo: Imprimir una Cadena

```
import java.util.function.Consumer;

public class Main {
    public static void main(String[] args) {
        Consumer<String> printer = s -> System.out.println(s);
        printer.accept("Hello, World!"); // Muestra: Hello, World!
    }
}
```

- **Explicación**: La lambda `s -> System.out.println(s)` define un `Consumer` que imprime su entrada. El método `accept` realiza la acción.

---

4. `Supplier<T>`: Generar un Resultado

La interfaz `Supplier<T>` representa un proveedor de resultados, que no toma ninguna entrada y devuelve un valor de tipo `T`. Su método abstracto es `get`.

Ejemplo: Generar un Número Aleatorio

```
import java.util.function.Supplier;
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        Supplier<Integer> randomInt = () -> new Random().nextInt(100);
        System.out.println(randomInt.get()); // Muestra un entero aleatorio entre 0 y 99
    }
}
```

- **Explicación**: La lambda `() -> new Random().nextInt(100)` define un `Supplier` que genera un entero aleatorio. El método `get` recupera el valor.

Usar Interfaces Funcionales con Streams

Estas interfaces brillan en la API de Streams de Java, donde permiten un procesamiento de datos conciso. Aquí tienes un ejemplo que filtra, transforma e imprime una lista de cadenas:

Ejemplo: Procesar una Lista de Cadenas

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList("a", "bb", "ccc", "dddd");

        Predicate<String> longerThanTwo = s -> s.length() > 2;       // Filtrar cadenas más largas que 2
        Function<String, String> toUpperCase = s -> s.toUpperCase(); // Convertir a mayúsculas
```

```
        Consumer<String> printer = s -> System.out.println(s);       // Imprimir cada resultado

        strings.stream()
                .filter(longerThanTwo)    // Mantiene "ccc" y "dddd"
                .map(toUpperCase)         // Convierte a "CCC" y "DDDD"
                .forEach(printer);        // Muestra: CCC, DDDD (en líneas separadas)
    }
}
```

- **Explicación**:
  - `filter` usa un `Predicate` para mantener cadenas con longitud > 2.
  - `map` usa una `Function` para transformar cadenas a mayúsculas.
  - `forEach` usa un `Consumer` para imprimir cada resultado.

Usando Referencias de Método Puedes hacer esto aún más corto con referencias de método:

```
strings.stream()
        .filter(s -> s.length() > 2)
        .map(String::toUpperCase)      // Referencia de método para Function
        .forEach(System.out::println); // Referencia de método para Consumer
```

Componer Interfaces Funcionales

Algunas interfaces permiten la composición para operaciones más complejas: - **Composición de Funciones**: Usa `andThen` o `compose`. `java Function<String, Integer> toLength = s -> s.length(); Function<Integer, String> toString = i -> "Longitud es " + i; Function<String, String> combined = toLength.andThen(toString); System.out.println(combined.apply("Hello")); // Muestra: Longitud es 5` - **Combinación de Predicados**: Usa `and`, `or` o `negate`. `java Predicate<String> isLong = s -> s.length() > 5; Predicate<String> startsWithA = s -> s.startsWith("A"); Predicate<String> isLongAndStartsWithA = isLong.and(startsWithA); System.out.println(isLongAndStartsWithA.test("Avocado")); // Muestra: true`

Resumen

Aquí tienes cuándo y cómo usar estas interfaces: - `Function<T, R>`: Transformar una entrada en una salida (por ejemplo, `apply`). - `Predicate<T>`: Probar una condición (por ejemplo, `test`). - `Consumer<T>`: Realizar una

acción en una entrada (por ejemplo, `accept`). - **`Supplier<T>`**: Generar un valor sin entrada (por ejemplo, `get`). - **Con Streams**: Combinarlas para un procesamiento de datos potente. - **Composición**: Encadenarlas o combinarlas para una lógica compleja.

Puedes implementar estas interfaces usando expresiones lambda (por ejemplo, `s -> s.length()`) o referencias de método (por ejemplo, `String::toUpperCase`). Ellas permiten un estilo de programación funcional en Java, haciendo tu código más conciso, legible y reutilizable, especialmente con la API de Streams.