**Spring Functional Interfaces: ¿Qué son y para qué sirven?**

1. **¿Qué es una *Functional Interface*?**

Una **Functional Interface** en Java es una interfaz que tiene **exactamente un método abstracto** (un único contrato funcional). Esto permite que pueda ser usada como objetivo de una **expresión lambda** o referencia a método.

* Son la base para programación funcional en Java.
* El compilador garantiza que solo hay un método abstracto.
* Pueden tener métodos default o static además del método abstracto.

**2. ¿Qué es una función? ¿Qué son Consumer, Supplier y Function?**

Spring, y Java en general, usan algunas interfaces funcionales comunes para manejar datos o procesos funcionales:

**a) Function<T, R>**

* Representa una función que recibe un argumento de tipo T y devuelve un resultado de tipo R.
* Firma: R apply(T t)

**Ejemplo:**

Function<String, Integer> lengthFunction = s -> s.length(); System.out.println(lengthFunction.apply("Hola")); // 4

**b) Consumer<T>**

* Representa una operación que recibe un argumento de tipo T y **no devuelve nada** (consume el dato).
* Firma: void accept(T t)

**Ejemplo:**

Consumer<String> printer = s -> System.out.println("Mensaje: " + s); printer.accept("Hola Mundo"); // Imprime: Mensaje: Hola Mundo

**c) Supplier<T>**

* Representa un proveedor o fuente que **no recibe argumentos** y devuelve un valor de tipo T.
* Firma: T get()

**Ejemplo:**

Supplier<Double> randomSupplier = () -> Math.random(); System.out.println(randomSupplier.get()); // Devuelve un número aleatorio

**d) Predicate<T> (aunque no pediste, es muy común)**

* Representa una función que recibe un argumento T y devuelve un booleano (verdadero/falso).
* Firma: boolean test(T t)

**3. Uso en Spring Framework**

Spring Framework (sobre todo a partir de Spring 5 y Spring Boot) fomenta el uso de programación funcional para:

* Declarar beans usando lambdas.
* Crear pipelines de procesamiento.
* Trabajar con funciones en aplicaciones reactivas y Spring Cloud Function.

**4. Ejemplos claros en Spring Boot usando Functional Interfaces**

**a) Function example**

Supongamos que definimos un bean que transforma un string:

@Bean public Function<String, String> uppercase() { return value -> value.toUpperCase(); }

Este bean puede ser invocado en otros lugares para convertir textos a mayúsculas.

**b) Consumer example**

Bean que consume un string y hace una acción (por ejemplo, imprimir):

@Bean public Consumer<String> logConsumer() { return value -> System.out.println("Recibido: " + value); }

**c) Supplier example**

Bean que provee valores sin recibir argumentos:

@Bean public Supplier<String> currentTimeSupplier() { return () -> Instant.now().toString(); }

**5. Integración con Spring Cloud Function**

Spring Cloud Function permite desplegar funciones Function, Consumer o Supplier y exponerlas fácilmente como servicios REST, eventos, etc.

Por ejemplo, si defines un bean Function<String, String> uppercase, puedes invocarlo vía HTTP, eventos o mensajes.

**6. Código completo ejemplo Spring Boot**

@SpringBootApplication public class FunctionalExampleApp { public static void main(String[] args) { SpringApplication.run(FunctionalExampleApp.class, args); } @Bean public Function<String, String> uppercase() { return value -> value.toUpperCase(); } @Bean public Consumer<String> logConsumer() { return value -> System.out.println("Logging: " + value); } @Bean public Supplier<String> timeSupplier() { return () -> Instant.now().toString(); } }

**7. ¿Por qué usar Functional Interfaces?**

* **Simplicidad y claridad:** puedes definir funciones como lambdas sin crear clases aparte.
* **Compatibilidad con programación reactiva** y flujos de datos.
* **Integración natural con APIs modernas y Spring Cloud Function.**
* **Testeo sencillo:** funciones son fáciles de testear unitariamente.

**Cómo crear tus propias interfaces funcionales en Java (para usar con Spring o en general)**

**1. Definición básica**

Una **Functional Interface** debe tener **exactamente un método abstracto** (sin implementar).

Para indicarle al compilador que una interfaz es funcional (y evitar errores accidentales), se suele usar la anotación:

@FunctionalInterface public interface MiInterfazFuncional {

// Un único método abstracto void ejecutar();

}

* La anotación @FunctionalInterface es opcional pero recomendada.
* La interfaz puede tener métodos default o static además del método abstracto.
* Puede extender otra interfaz funcional mientras no añada métodos abstractos nuevos.

**2. Ejemplo simple**

@FunctionalInterface public interface MiFuncion {

int operar(int a, int b);

}

Esto define una función que toma dos int y devuelve un int.

**3. Usar tu interfaz con expresiones lambda**

MiFuncion suma = (a, b) -> a + b;

System.out.println(suma.operar(3, 4)); // Imprime 7

**4. Interfaces funcionales con parámetros y retorno genéricos**

Para que sea más flexible, podés usar genéricos:

@FunctionalInterface public interface MiFuncionGenerica<T, R> {

R aplicar(T t);

}

Ejemplo de uso:

MiFuncionGenerica<String, Integer> longitud = s -> s.length(); System.out.println(longitud.aplicar("Hola")); // Imprime 4

**5. Ejemplo de interfaz funcional con múltiples parámetros**

@FunctionalInterface public interface MiBiFuncion<T, U, R> {

R aplicar(T t, U u);

}

Uso:

MiBiFuncion<Integer, Integer, Integer> multiplicar = (a, b) -> a \* b; System.out.println(multiplicar.aplicar(3, 5)); // 15

**6. Buenas prácticas al crear interfaces funcionales**

* Usá la anotación @FunctionalInterface para mayor claridad y seguridad.
* Evitá métodos abstractos adicionales.
* Documentá claramente el propósito y contrato del método.
* Elegí nombres descriptivos para la interfaz y método (ej: operate, apply, test, etc).
* Aprovechá genéricos para máxima reutilización.

**7. Integración con Spring**

* Podés definir beans con tus propias interfaces funcionales.
* Spring las tratará igual que las estándar si cumplen con la definición (único método abstracto).
* Podés inyectarlas, usarlas en lambdas, o exponerlas en Spring Cloud Function (con algo de configuración).