Interfaces Funcionales en Java

Por definición, una interface funcional es aquella que tiene un único método abstracto. Así de sencillo. No importa que además tenga métodos estáticos o implementados por defecto.

Java 8 añadió la anotación `@FunctionalInterface` para marcar este tipo de interfaces. Es una anotación puramente informativa, usada para reforzar la intención de las interfaces, pero que permite a los compiladores generar un error si se aplica sobre una interface que no cumpla con la definición de interface funcional, o sobre una clase, enumerado u otra anotación.

```
@FunctionalInterface
public interface Computer {
 public void compute();
 }
}
```

Las interfaces funcionales se pueden instanciar con expresiones lambda, referencias a métodos o referencias a constructores. Todos ellos conceptos añadidos en Java 8.

```
Computer lambda = () -> {};

Computer metodo = System::gc;

Computer constructor = String::new;
```

Como se observa, la declaración de la interface no tiene que estar relacionada de manera directa con la expresión que la instancia. Ni siquiera tiene que aparecer el nombre del método funcional. Lo único que debe cumplirse es que la expresión cumpla con la interface, es decir que la expresión lambda, o la referencia al método o constructor tenga el mismo número de argumentos y tipo retornado.

Java 8 añadió una gran cantidad de interfaces funcionales dentro del paquete **java.util.function**, tanto para uso interno como para su uso general. Las interfaces principales son **Consumer<T>**, **Supplier<T>**, **Function<T, R>** y **Predicate<T>**, el resto son especializaciones de estas.

La nomenclatura utilizada para el nombre de las clases especializadas indica cómo se diferencian de la interfaces base que especializan:

- El prefijo Bi indica que admite dos parámetros.
- El prefijo Boolean, Int, Long y Double indica que admite un parámetro o genera un resultado de tipo boolean, int, long o double respectivamente.

- El prefijo ObjInt, ObjLong y ObjDouble indica que admite dos parámetros, el primero del mismo tipo que la interface, y el segundo de tipo int, long o double respectivamente.
- El prefijo ToInt, ToLong y ToDouble indica que produce un resultado de tipo int, long o double respectivamente.
- El prefijo IntToLong, IntToDouble, LongToInt, LongToDouble, DoubleToInt y DoubleToLong indica que admite un parámetro del primer tipo indicado y produce un resultado del segundo.
- El sufijo Operator indica una especialización de **Function**.

Consumer<T>

Consumer<T> representa una función con un argumento que no retorna nada. Tal y como su nombre indica, consume un valor y no genera nada. Un ejemplo de este tipo de funciones es el método de impresión por la salida estándar, que admite el texto a imprimir y no retorna nada:

```
Consumer<String> consumidor = System.out::print;
```

La interface **Consumer<T>** ofrece el método funcional **accept**, e implementa por defecto el método **andThen** que permite concatenar varias acciones en secuencia.

```
Consumer<Date> reset = fecha -> fecha.setTime(0);
Consumer<Date> print = System.out::println;

Date fecha = new Date();
reset.andThen(print).accept(fecha);
```

Notar que la interface **Consumer<T>** permite trabajar en base a efectos laterales provocados al modificar el estado de los objetos consumidos, si estos no son inmutables, para que la siguiente operación indicada en el método **andThen** tenga sentido.
Interfaces especializadas de **Consumer<T>**:

```
BiConsumer<T, U>
DoubleConsumer
IntConsumer
LongConsumer
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
```

Supplier<T>

Supplier<T> representa una función sin argumentos que retorna un resultado. Tal y como su nombre indica, genera un valor. Un ejemplo de este tipo de funciones es el método de generación de números aleatoriosde **Math**, que no requiere argumentos:

```
Supplier<Double> suministrador = Math::random;
```

La interface **Supplier<T>** ofrece el método funcional **get** y no implementa ningún otro método.

```
DoubleSupplier generador = Math::random;
DoubleConsumer print = System.out::println;

DoubleStream.generate(generador).limit(10).forEach(print);
```

Interfaces especializadas de **Supplier<T>**:

```
BooleanSupplier
DoubleSupplier
IntSupplier
LongSupplier
```

Function<T, R>

Function<T, R> representa una función con un argumento que retorna un resultado. Representa la expresión más habitual de lo que se entiende por función. Un ejemplo de este tipo de funciones es el método que convierte una cadena de texto en mayúsculas:

```
Function<String, String> funcion = String::toUpperCase;
```

La interface **Function<T, R>** ofrece el método funcional **apply** y el método estático **identity** que retorna una función que devuelve el propio parámetro que se le pasa. E implementa por defecto el método **compose**, que aplica una función dada al parámetro pasado al método funcional y después ejecuta el método funcional sobre el resultado, y **andThen**, que permite concatenar varias acciones en secuencia.

```
IntUnaryOperator square = valor -> valor * valor;
IntUnaryOperator negate = valor -> -valor;
IntConsumer print = System.out::println;

IntStream.rangeClosed(1, 10).
                    map(negate.compose(square)).forEach(print);
```

Interfaces especializadas de **Function<T, R>** :

```
BiFunction<T, U, R>
DoubleFunction<R>
DoubleToIntFunction
DoubleToLongFunction
IntFunction<R>
IntToDoubleFunction
IntToLongFunction
LongFunction<R>
LongToDoubleFunction
LongToIntFunction
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
```

Predicate<T>

Predicate<T> representa una función con un argumento que retorna un **boolean**. Tal y como su nombre indica, evalúa una expresión y retorna **true** o **false** para indicar si se cumple o no la expresión. Un ejemplo de este tipo de funciones es el método que comprueba si una carácter es un espacio en blanco o no:

```
Predicate<Character> predicado = Character::isWhitespace;
```

La interface **Predicate<T>** ofrece el método funcional **test** y el método estático **isEqual** , que retorna un predicado que compara los dos parámetros que se le pasan con el método **Objects.equals**. E implementa por defecto los métodos **and**, **or** y **negate**, para la composición y negación de predicados.

```
IntPredicate isWhitespace = Character::isWhitespace;
IntPredicate isUpper = Character::isUpperCase;
IntConsumer print = System.out::println;

"aBcDeF".chars().
                filter(isWhitespace.negate(). and(isUpper)).
                forEach(print);
```

Interfaces especializadas de **Predicate<T>**:

```
BiPredicate<T, U>
DoublePredicate
IntPredicate
LongPredicate
BinaryOperator<T>
DoubleBinaryOperator
DoubleUnaryOperator
IntBinaryOperator
IntUnaryOperator
LongBinaryOperator
LongUnaryOperator
UnaryOperator<T>
```

Además de esta interfaces funcionales de uso genérico, Java 8 añadió interfaces especializadas en distintos paquetes para cubrir necesidades concretas, pero las cuatro básicas son las importantes porque explica el funcionamiento de todas las demás.