JUG Genova

Practical functional programming

Java

bonfante.fabio@gmail.com

INDEX

- Functional Programming **intro**
- Usage in **Java API** (Stream e Optional)
- Practical benefits in **your API**
  - Performance: Lazy evaluation
  - Closure + Readability
  - Composition over Inheritance

Disclaimer

Non ho esperienza con linguaggi **funzionali puri**, quello che so deriva dall’uso delle lambda da Java8, un po' di Groovy e si... pure da JavaScript.

Non aspettatevi definizioni formali di monadi, high-order functions etc..

Intro to Functional Programming

Cos’è una funzione?

(in java)

Sostanzialmente
è sempre un metodo

```
void main(String[] args)
```

caratterizzato dalla
sua firma

Quando un **linguaggio** supporta la **programmazione funzionale**?

Quando una funzione può essere

- salvata in una **variabile**
- passata come **parametro**
- un **valore di ritorno**

e ovviamente... **eseguita** (prima o poi)

Cosa ho a disposizione in Java?

- java.util.function.Function (and friends) *dichiarazione*
- lambda expression *implementazione*
- method reference *“puntatore” all’implementazione*

che “funzionano” grazie alle **Functional Interfaces**

java.util.function.Function (and friends)

> Represents a function that accepts one argument and produces a result.
> This is a functional interface whose functional method is `apply(Object)`.
>
> Since: 1.8
>
> Type parameters: `<T>` – the type of the input to the function
> `<R>` – the type of the result of the function

```
@FunctionalInterface
public interface Function<T, R> {
```

> Applies this function to the given argument.
>
> Params: `t` – the function argument
>
> Returns: the function result

```
    R apply(T t);
```

java.util.function.Function (and friends)

Represents a function that accepts two arguments and produces a result. This is the two-arity specialization of `Function`.
This is a functional interface whose functional method is `apply(Object, Object)`.

Since: 1.8

See Also: `Function`

Type parameters: `<T>` – the type of the first argument to the function
`<U>` – the type of the second argument to the function
`<R>` – the type of the result of the function

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
```

Applies this function to the given arguments.

Params: `t` – the first function argument
`u` – the second function argument

Returns: the function result

```
    R apply(T t, U u);
```

java.util.function.Function (and friends)

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, `Consumer` is expected to operate via side-effects.
This is a functional interface whose functional method is `accept(Object)`.

Since: 1.8

Type parameters: `<T>` – the type of the input to the operation

```
@FunctionalInterface
public interface Consumer<T> {
```

Performs this operation on the given argument.

Params: `t` – the input argument

```
    void accept(T t);
```

java.util.function.Function (and friends)

Represents a supplier of results.
There is no requirement that a new or distinct result be returned each time the supplier is invoked.

This is a functional interface whose functional method is `get()`.

Since: 1.8

Type parameters: `<T>` – the type of results supplied by this supplier

```
@FunctionalInterface
public interface Supplier<T> {
```

Gets a result.

Returns: a result

```
    T get();
}
```

java.util.function.Function (and friends)

Represents a predicate (boolean-valued function) of one argument.
This is a functional interface whose functional method is `test(Object)`.

Since: 1.8

Type parameters: `<T>` – the type of the input to the predicate

```
@FunctionalInterface
public interface Predicate<T> {
```

Evaluates this predicate on the given argument.

Params: `t` – the input argument

Returns: `true` if the input argument matches the predicate, otherwise `false`

```
    boolean test(T t);
```

java.util.function.Function (and friends)

Represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of `Function` for the case where the operand and result are of the same type.
This is a functional interface whose functional method is `apply(Object)`.

Since: 1.8

See Also: `Function`

Type parameters: `<T>` – the type of the operand and result of the operator

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
```

Returns a unary operator that always returns its input argument.

Returns: a unary operator that always returns its input argument

```
    static <T> UnaryOperator<T> identity() { return t → t; }
```

java.util.function.Function (and friends)

Represents an operation upon two operands of the same type, producing a result of the same type as the operands. This is a specialization of `BiFunction` for the case where the operands and the result are all of the same type.
This is a functional interface whose functional method is `apply(Object, Object)`.

Since: 1.8

See Also: `BiFunction`, `UnaryOperator`

Type parameters: `<T>` – the type of the operands and result of the operator

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
```

Functional Interfaces

Any interface with a SAM (Single Abstract Method) is a functional interface, and its implementation may be treated as **lambda expressions.**

source baeldung.com

java.util.function.**Function** non è “speciale”

è solo una `@FunctionalInterface` che può essere **implementata con una lambda**

Usage in Java API

Stream

Optional

Stream

API che riesce ad essere “general purpose”

grazie alle funzioni

Dichiarativa

tramite le operazione intermedie e terminali

(collect,anyMatch,findFirst,count,forEach,reduce, ...)

permette diverse modalità di esecuzione (es. parallelStream)

Optional

è più facile considerarlo uno

Stream di dimensione 1

Optional

Operazioni intermedie

filter(Predicate<? super T> predicate)

map(Function<? super T, ? extends U> mapper)

flatMap(Function<? super T, Optional<U>> mapper)

Operazioni terminali

get()

orElse(T other)

orElseGet(Supplier<? extends T> other)

orElseThrow(Supplier<? extends Throwable> exceptionSupplier)

ifPresent(Consumer<? super T> action)

ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)

Practical benefits in **your API**

Performance: Lazy execution

Closure + Readability

Composition over inheritance

Performance: Lazy execution

Se posso evitare di eseguire del codice
⇒ sono più veloce

Una lambda è un “insieme di codice da eseguire”…
⇒ **dove scrivo una lambda non è “quando verrà eseguita”** 🤔

Functional Programming non rende il tuo codice più veloce...
ma se fai eseguire solo quello che serve, si!

Closure + Readability

Le lambda non sono sempre il massimo della leggibilità

⇒ ma se gli diamo un nome...

e se la lambda si basa su variabili “fuori” dal suo scope?

Composition over Inheritance

L’ereditarietà NON È
il modo “per eccellenza” di fattorizzare il codice

L’ereditarietà != Object Oriented Programming

Oggetti composti da

un *behaviour* generale
ben definito

funzioni che completano
dettagli implementativi

+ pratici nell'utilizzo

+ riutilizzabili

+ manutenibili

Functional Programming

(real*) OOP

* Dr. Alan Kay on the Meaning of “Object-Oriented Programming”