LPP - +3CFU (corso da 9 CFU)

Cenni sui metodi di default

Viviana Bono

Referenze

Capitolo 9 (cenni) di **Java 8 in action**

API:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Compatibility

"There are three main kinds of compatibility when introducing a change to a Java program: **binary**, **source**, and **behavioural** compatibilities. [...]

Binary compatibility means existing binaries running without errors continue to link (which involves verification, preparation, and resolution) without error after introducing a change. For example, just adding a method to an interface is binary compatible because if it's not called, existing methods of the interface can still run without problems. In its simplest form, **source compatibility** means an existing program will still compile after introducing a change. For example, adding a method to an interface isn't source compatible; existing implementations won't recompile because they need to implement the new method. Finally, **behavioural compatibility** means running a program after a change with the same inputs results in the same behaviour. For example, adding a method to an interface is behavioural compatible because the method is never called in the program (or it gets overridden by an implementation)."

Un esempio di metodo di default

```
// prima di Java 8
interface Iterator<T> {
  boolean hasNext();
  T next();
  void remove();
  // poco usato, ma ogni classe che non ha bisogno di remove() deve
  // cmq implementarlo, es. con body vuoto (boilerplate code!)
  // per avere source compatibility
}

// da Java 8 in poi
interface Iterator<T> {
  boolean hasNext();
  T next();
  default void remove() {
      throw new UnsupportedOperationException();
  } // non va più implementato per forza
}
```

Ragione principale per i default methods: modificate molte interfacce delle Java collection per sfruttare le lambda $\Rightarrow$ mantenere la backward (source) compatibility.

Abstract classes vs. interfaces in Java 8

"So what's the difference between an abstract class and an interface? They both can contain abstract methods and methods with a body. First, a class can extend only from one abstract class, but a class can implement multiple interfaces ⇒ A FORM OF MULTIPLE INHERITANCE ***. Second, an abstract class can enforce a common state through instance variables (fields). An interface can't have instance variables."

*** V. Bono, E. Mensa, M. Naddeo. *Trait-oriented Programming in Java 8*.

Regole di disambiguazione dei metodi

Method with the same signature from multiple places:
1. Classes always win (also abstract classes).
2. Otherwise, sub-interfaces win: if B extends A, B is more specific than A.
3. Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method to use by overriding it and calling the desired method explicitly:

```
public interface A {
    default void hello() {
      System.out.println("Hello from A");
     }
}

public interface B {
    default void hello() {
      System.out.println("Hello from B");
     }
}

public class C implements B, A {
   void hello(){
       B.super.hello()
       // we call the hello() implementation explicitly
   }
}
```

The `Function<T, R>` functional interface

Consideriamo l'interfaccia funzionale `Function<T, R>`:

https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

dove `T` è il tipo dell'input della funzione, `R` quello del risultato, e il metodo `apply()` è il metodo funzionale.
Tra i suoi metodi di default c'è il metodo `andThen()`:

```
default<V> Function<T,V> andThen(Function<? super R, ? extends V> after)
```

funzione composta che prima applica la funzione 'this" e poi quella passata come parametro al risultato (detta funzione 'after').

```
Function <Integer, Integer> f = x ->x+1;
Function <Integer, Integer> g = x -> x*2;
Function <Integer, Integer> h = f.andThen(g); // corrisponde a g(f())

int res = h.apply(1); // corrisponde a g(f(1))
```

andThen() dal punto di vista dei tipi

```
default<V> Function<T,V> andThen(Function<? super R, ? extends V> after)
```

Vediamo l'esempio dal punto di vista dei tipi della signature:

```
g(f): T -> V;      f: T -> R (funzione this);      g : R -> V (fuzione after)
```

Perché nella signature abbiamo "? super R" e "? extends V"? nel tipo di `after` ("?" è la wildcard, una forma di tipo parametrico.) Per poter sfruttare la flessibilità dei tipi parametrici, mantenendo la correttezza. Infatti:

- "? super R" vuol dire "qualsiasi tipo più grande di R" $\Rightarrow$ il tipo dell'input di una funzione può variare solo in modo contro-variante;
- "? extends V" vuol dire "qualsiasi tipo più piccolo di V" $\Rightarrow$ il tipo del risultato di una funzione può variare solo in modo co-variante,

come abbiamo visto.