Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Lambda expression

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

FUNZIONI NEI LINGUAGGI DI PROGRAMMAZIONE

- Tutti i linguaggi di programmazione hanno costrutti per esprimere *funzioni,* MA nei linguaggi imperativi tradizionali una funzione è *solo un costrutto che incapsula codice*
  - ha un nome, un'intestazione (signature), un corpo
  - viene *chiamata* con l'operatore **()**
- MOTIVO: tradizionalmente, il software suddivide a priori «dati» e «codice»
  - i dati stanno da una parte, il codice da un'altra
  - i dati sono «passivi», il codice «attivo»
- MA.. è veramente «giusta», «sana», questa distinzione?

FUNZIONI NEI LINGUAGGI DI PROGRAMMAZIONE

- Vedere la funzione solo come *solo un (mero) costrutto che incapsula codice* corrisponde una visione «low level», ispirata dall'hardware
- È una visione miope, peraltro difforme dalla matematica
- Perché *non* interpretare la funzione, piuttosto, come *un particolare «tipo di dato»?*
  - ha le sue proprietà (nome, intestazione, argomenti, codice)
  - potrebbe avere i suoi accessor
  - sebbene sia solitamente definita come literal (costante scrittta nel codice), potrebbe magari essere anche costruita dinamicamente (tramite **`new`**)
  - ha un'unica vera caratteristica in più: è *eseguibile*

FUNZIONI NEI LINGUAGGI DI PROGRAMMAZIONE

- Non considerare la funzione come un particolare tipo di dato, come accade nei linguaggi tradizionali, ha conseguenze:
  - *non può* essere assegnata a variabili
    - approssimazione: i puntatori a funzione del C, che però contengono solo l'indirizzo del codice, non una vera "entità funzione"..
  - *non può* essere passata come argomento a un'altra funzione
    - perché gli argomenti devono essere *valori* di un qualche *tipo*, mentre le funzioni in quei linguaggi non lo sono
  - *non può* essere restituita da un'altra funzione
    - perché non si può "sintetizzare comportamento" come fosse un valore (e le funzioni esprimono "comportamenti"…)
    - al più, si può *restituire l'indirizzo di una funzione già esistente*
  - *non può* essere definita e usata "al volo", tramite **`new`**
    - le funzioni devono essere definite nel codice, staticamente

FUNZIONI NEI LINGUAGGI FUNZIONALI

- Al contrario, i linguaggi *funzionali* si basano proprio sull'idea di *funzioni* come *fondamentale tipo di dato*
  - si usa dire come «first-class entities»
- Una funzione che sia *first-class entity* deve essere *manipolabile come ogni altro tipo di dato* e quindi:
  - deve poter essere *assegnata a variabili,* di tipo *"funzione da A a B"*
  - deve poter essere *passata come argomento a un'altra funzione*
  - deve poter essere *definibile come ogni altro oggetto*
- Diviene quindi possibile *passare una funzione (ossia, «comportamento»)* ad altre funzioni, *come argomento*
  - con la stessa facilità con cui passiamo *dati*, in questa prospettiva possiamo pensare di passare *codice eseguibile!*

LINGUAGGI A OGGETTI «BLENDED»

- Molti linguaggi a oggetti, nati su base imperativa, *stanno incorporando il concetto di funzione come first-class entity* [o qualcosa che le somigli] → linguaggi *blended*
  - Javascript: da sempre ha il costruttore **`Function`** e la keyword **`function`** per specificare funzioni "anonime"
  - Java: supporto *parziale* introdotto in Java 8
    - funzioni (anonime) come *lambda expression*
    - si possono definire come literal e passare come argomenti
    - MA non sono vere first class entities, perché *non definiscono un tipo*
    - anziché appartenere a un vero "tipo-funzione", si appoggiano a un "opportune interfacce, dette *interfacce funzionali*
  - C#: supporto efficace, tramite *delegati*
  - Kotlin, Scala: supporto *completo* molto efficace, API riorganizzate

WHAT'S NEXT

- Per cogliere appieno l'importanza e l'utilità di questo cambio di paradigma, nel seguito:
  1. Riconsidereremo l'ordinamento con *comparator*
  2. Introdurremo il concetto di *lambda expression*
     - prima in generale, poi in Java, infine negli altri linguaggi
  3. Riprenderemo quindi l'esempio dell'ordinamento riformulandolo sfruttando le *lambda expression*
  4. Estenderemo infine l'ambito di applicazione affrontando il tema dell'*iterazione interna*

UN ANTICO ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Riprendiamo la "solita" classe **`Persona`**:

Java

```
class Persona {
 private String nome, cognome;
 private int età;
 public Persona(String nome, String cognome, int età) {
    this.nome=nome; this.cognome=cognome; this.età=età;
 }
 public String getCognome(){ return cognome; }
 public String getNome(){ return nome; }
 public int getEtà(){ return età; }
}
```

- E il "solito" array di **`Persona`**:

Java

```
Persona[] persone = new Persona[]{
     new Persona("John", "Doe", 25),
     new Persona("Jane", "Doe", 45),
     new Persona("Anne", "Bee", 31),
     new Persona("Jane", "Doe", 22)
};
```

UN ANTICO ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Avevamo definito vari comparatori che incapsulavano diversi criteri:

Java

```
class CognomeComparator implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2){
      return p1.getCognome().compareTo(p2.getCognome());
    }
}

class NomeComparator implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2){
      return p1.getNome().compareTo(p2.getNome());
    }
}

class EtaComparator implements Comparator<Persona> {
    public int compare(Persona p1, Persona p2){
      return Integer.compare(p1.getEtà(), p2.getEtà());
    }
}
```

UN ANTICO ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Il mini-main di esempio ordinava un array in vari modi:

Java

```
System.out.println(String.join(" ," , persone);

Arrays.sort(persone, new CognomeComparator() );
System.out.println(String.join(" ," , persone);

Arrays.sort(persone, new NomeComparator() );
System.out.println(String.join(" ," , persone);

Arrays.sort(persone, new EtaComparator() );
System.out.println(String.join(" ," , persone);
```

- Avevamo anche considerato il caso del comparatore definito tramite *classe anonima,* inline:

Java

```
Arrays.sort(persone, new Comparator<Persona>(){
    public int compare(Persona p1, Persona p2){
      return p1.getCognome().compareTo(p2.getCognome());
    }
 );
```

IL RUOLO DEL `Comparator`

- MA.. qual è *esattamente* il ruolo del **`Comparator`**?
- A ben vedere, è solo quello permetterci di passare a **`sort`** la *funzione di confronto* (**`compare`**)

Java

```
Arrays.sort(persone,
        new Comparator<Persona>() {
                public int compare(Persona x, Persona y){...}
        }
    );
```

- **`Comparator`** è una sorta di «wrapper» per la **`compare`**
  - non essendo le funzioni delle vere first-class entity, in Java **`compare`** *non può essere manipolata come oggetto stand alone*
  - il comparator le fa quindi da «adulto accompagnatore», perché lei non può «andare in giro da sola»

PASSARE COMPORTAMENTO: ANCORA IL COMPARATOR

- L'interfaccia **`Comparator`** ha in effetti come suo *unico* scopo quello di dichiarare il metodo **`compare`**
  - non a caso, è il suo *unico* metodo
- Implementando **`Comparator`**, noi implementiamo di fatto *una specifica funzione* ***`compare`***
  - è quella che vorremmo "passare" a **`sort`** …
  - …ma che finora non abbiamo potuto passarle direttamente
- Non sarebbe più bello se potessimo scrivere qualcosa come:

```
Arrays.sort(persone, compare(…) );
```

senza tutto quell'accrocchio intorno…? ☺

FUNZIONI vs OGGETTI

- Quello che *davvero vorremmo* è passare a **`sort`** la **`compare`**
- Quello che *in realtà facciamo* è *passarle un oggetto che implementi* ***`Comparator`****, che la incapsula*

MOTIVO: nei linguaggi imperativi, classicamente,

le funzioni *non sono «first class entities»*

ossia

non sono entità «con pieno diritto di cittadinanza»

In particolare, non sono "veri oggetti".

- NUOVO OBIETTIVO: elevare le funzioni a *first-class entities,* pur nel contesto di un linguaggio mainstream a oggetti
- I linguaggi *blended* introducono a tal fine le *lambda expression*

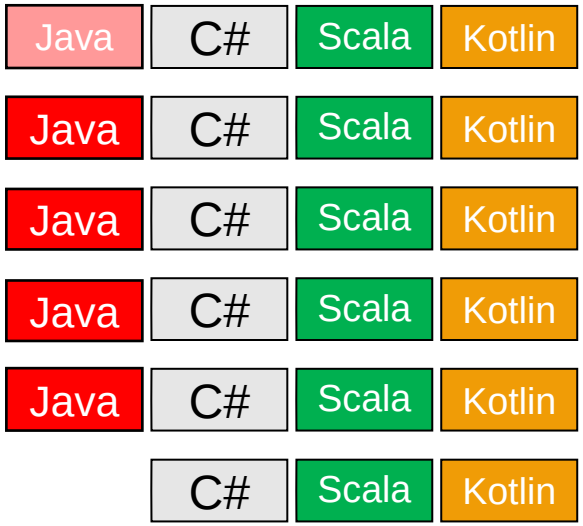
FUNZIONI COME VERI OGGETTI

- Cosa significa avere «funzioni come veri oggetti»?
  - le funzioni devono essere viste come *istanze* di un opportuno *tipo*
  - si devono poter definire *riferimenti* (a funzioni) di quel *tipo*
  - si devono poter creare *oggetti* (funzioni) di quel *tipo*
  - si devono poter *assegnare* tali oggetti (funzioni) a riferimenti di quel tipo
  - si devono poter *passare come argomenti* ad altre funzioni
  - in breve, si devono poter trattare in modo *uniforme a ogni altro oggetto*, con l'unica caratteristica ulteriore di essere *eseguibili:* oggetti su cui applicare l'operatore di chiamata `()` che *mette in esecuzione il codice*

Non tutti i linguaggi a oggetti *blended* supportano *tutte* queste caratteristiche in modo *completo.*

LINGUAGGI BLENDED

- I vari blend «at a glance»
  - funzioni viste come *istanze* di un *tipo-funzione*
  - definire *riferimenti* (a funzioni) di quel *tipo*
  - creare *oggetti* (funzioni) di quel *tipo*
  - *assegnare* tali oggetti a riferimenti
  - *passare come argomenti* ad altre funzioni
  - applicare l'operatore di chiamata `()`



- Java è un blend soltanto parziale
  - supporto introdotto in Java 8
  - non introduce un vero *tipo-funzione,* ri-sfrutta la nozione di *interfaccia*
  - *non consente* di applicare direttamente l'operatore di chiamata `()`

IL BLEND DI JAVA

- Java è arrivato *buon ultimo* a proporre il suo "blend"
- Obiettivi primari:
  - retrocompatibilità, più che innovazione totale
  - nessuno stravolgimento del linguaggio
    (in particolare: mantenimento della nozione di *nominal typing*)
- Risultato: un blend *all'80%*
  - esistono oggetti-funzione**,** ma *non esiste un vero "tipo funzione":*
    si sfrutta a tal fine la nozione di interfaccia → *interfacce funzionali* ☹
  - gli oggetti-funzione *non* sono *direttamente* eseguibili con l'operatore di chiamata `()`: occorre *invocare per nome il metodo sottostante* ☹ ☹ ☹
- Massima resa, ma risultato "non fully functional"

LAMBDA EXPRESSION

- Per definire oggetti-funzione si utilizza un *particolare tipo di espressione*, noto come *lambda expression*
- Ricorda la notazione delle funzioni in matematica:

$$x \rightarrow x+3 \qquad x,y \rightarrow 2x-y$$

Una *lambda expression* è una *funzione anonima,* espressa in modo simile alla notazione matematica:

Java

(lista argomenti) **->** **{** *corpo della funzione* **}**

Le parentesi tonde/graffe si possono *omettere* nei casi semplici:

- se c'è un solo statement nel corpo → si omettono le *graffe*
- se c'è un solo argomento in lista → si omettono le *tonde*

I *tipi degli argomenti* si possono omettere se deducibili dal contesto

LAMBDA EXPRESSION NEI VARI LINGUAGGI

- Java:

  (*lista argomenti*) -> { *corpo della funzione* } Java

- C#

  (*lista argomenti*) => { *corpo della funzione* } C#

- Scala:

  (*lista argomenti*) => { *corpo della funzione* } Scala

- Kotlin:

  { *lista argomenti* -> *corpo della funzione* } Kotlin

  – NB: da un linguaggio all'altro cambiano il *tipo di freccia* e alcuni dettagli sulla collocazione e l'obbligatorietà delle varie *parentesi,* oltre ovviamente al modo di specificare il *tipo* degli argomenti

IL TIPO DELLA LAMBDA

- In matematica, le funzioni si definiscono stabilendo *dominio e codominio*:

f: Z → Z	**g: ZxZ → Z**	**h: RxR → R**
x → x+3	x,y → 2x-y	x,y → x+πy

- Dunque, *le funzioni non hanno tutte lo stesso «tipo»*
  - una funzione sul dominio Z non è intercambiabile con una su RxR
  - il «tipo» della funzione è definito dal suo *dominio* e dal suo *codominio*

  Ad esempio, nei casi sopra:
  - f: Z → Z *«funzione a un argomento dagli interi agli interi»*
  - h: RxR → R *«funzione a due argomenti dai reali ai reali»*
- Ciò richiede di *specificare il tipo* di argomenti e risultato.

IL TIPO DELLA LAMBDA

- Nella lista argomenti occorre perciò, di norma, specificare il *tipo* di ogni argomento
  - salvo che si possa dedurlo dal contesto tramite *type inference*
- Il tipo del risultato è spesso dedotto per type inference
  - ESEMPIO: funzione che, dati $(x,y) \in Z$, calcola 2x-y

```
(int x, int y) -> 2*x-y
```
Java

```
(int x, int y) => 2*x-y
```
C#

```
(x:Int, y:Int) => 2*x-y
```
Scala

```
{ x:Int, y:Int -> 2*x-y }
```
Kotlin

- Scrivere un siffatto *lambda literal* equivale a *fare la new* di un nuovo oggetto-funzione, istanza di un "*opportuno tipo*" stabilito dal compilatore (o dichiarato dall'utente)

IL TIPO DELLA LAMBDA

- Come insegna la matematica, ogni funzione ha il suo *tipo* che specifica o riassume *dominio e codominio*
- Ad esempio:

```
(int x, int y) -> 2*x-y
```
f: ZxZ → Z

«funzione a due argomenti dagli interi agli interi»

```
(Counter c) -> c.getValue()
```
f: C → N

«funzione a un argomento dall'insieme dei Counter ai naturali»

```
(String s, int n) -> s+n
```
f: SxZ → S

«funzione a due argomenti, una stringa e un intero, a stringa»

eccetera

- *Come specificare ciò* in un linguaggio di programmazione?

TIPO NOMINALE vs. STRUTTURALE

- Posto che il *tipo della funzione* è una locuzione come:
  - f: $Z \rightarrow Z$ *«funzione a un argomento dagli interi agli interi»*
  - h: $R \times R \rightarrow R$ *«funzione a due argomenti dai reali ai reali»*
- Ci sono due possibilità:
  - *dare un nome* a ogni tipo → approccio *NOMINALE*
  - *scrivere «da dominio a codominio»* → approccio *STRUTTURALE*
- Entrambi gli approcci hanno pro e contro
  - l'approccio nominale costringe a inventarsi nomi per ogni possibile tipo di funzione ☹, MA è poco invasivo in un linguaggio esistente ☺
  - l'approccio strutturale evita di doversi inventare altri nomi ☺, MA si incastra più in profondità nel type system di un linguaggio 😐 rendendo non semplice la sua adozione in linguaggi già esistenti ☹

TIPO NOMINALE vs. STRUTTURALE

- Con il *tipo nominale* bisogna inventarsi dei *nomi* per le diverse possibili situazioni, ad esempio:
  - f: $Z \rightarrow Z$ diremo che è di tipo *IntToIntFunction* (Java) o magari *Func<int,int>* (C#)
  - h: $RxR \rightarrow R$ diremo che è di tipo *RealToRealBiFunction* (Java) o magari *Func<double,double,double>* (C#)
- Con il *tipo strutturale la notazione stessa funge da tipo:*
  - f: $Z \rightarrow Z$ diremo che è di tipo «$Z \rightarrow Z$» (o notazione analoga)
  - h: $RxR \rightarrow R$ diremo che è di tipo «$RxR \rightarrow R$» (o notazione analoga)
- Ci sono pro e contro in entrambe le scelte
  - l'approccio strutturale è la soluzione più pulita *se si parte da zero*
  - l'approccio nominale è più semplice da inserire *in un linguaggio col type system già definito,* ma implica definire e ricordare (molti) nomi

LE SCELTE DEI VARI LINGUAGGI

- Java e C# scelgono l'approccio *nominale*

C# Java

  - MOTIVO: retrocompatibilità, facilità di riconversione del legacy code
  - PRO: si sposa *naturalmente* con il concetto di *interfaccia* (eventualmente adattato, v. «*delegato*» in C#) → integrazione seamless ☺
  - CONTRO: necessità di *definire* e *ricordare* parecchie interfacce ☹ (*interfacce funzionali* in Java) o *delegati standard* (C#)

Scala Kotlin

- Scala, Kotlin optano invece per l'approccio *strutturale*
  - MOTIVO: sono stati definiti incorporando *fin dall'inizio* la nozione di funzione come «first-class entity» nel type system del linguaggio ☺
  - PRO: il tipo della lambda è evidente, non occorre ricordarsi nomi
  - CONTRO: type system più complesso, leggibilità.. *dipende* 😐

IL TIPO DELLA LAMBDA NEI VARI LINGUAGGI

- In Java e C# il tipo della lambda è dunque espresso da un *nome* che «evochi» dominio e codominio della funzione
  - in Java, da un'appropriata *interfaccia funzionale* scelta fra quelle standard definite nel package **`java.util.function`**
  - in C#, da uno dei *delegati standard* **`Func<…>`** o **`Action<…>`**
- Nel caso dei nostri due esempi:
  - f: $Z \rightarrow Z$

in Java	**`f`** è di tipo **`IntUnaryOperator`**	Java
in C#	**`f`** è di tipo **`Func<int,int>`**	C#
in Scala	**`f`** è di tipo **`Int => Int`**	Scala
in Kotlin	**`f`** è di tipo **`Int -> Int`**	Kotlin

  - h: $RxR \rightarrow R$

in Java	h è di tipo **`DoubleBinaryOperator`**	Java
in C#	h è di tipo **`Func<double,double,double>`**	C#
in Scala	h è di tipo **`Double,Double => Double`**	Scala
in Kotlin	h è di tipo **`Double,Double -> Double`**	Kotlin

ESEMPI MULTI-LINGUAGGIO

- Dichiarazioni complete con tutti i tipi specificati:

```
IntUnaryOperator f = (int x) -> x+3;
Func<int,int>    f = (int x) => x+3;
val f : Int => Int =  x:Int => x+3;
val f : (Int) -> Int = { x:Int -> x+3; }
```

Java
C#
Scala
Kotlin

- Dichiarazioni abbreviate con type inference:

```
IntUnaryOperator f = x -> x+3;
Func<int,int>    f = x => x+3;
val f : Int => Int = x => x+3;
val f = x:Int => x+3;
val f : (Int) -> Int = { x -> x+3; }
val f = { x:Int -> x+3; }
```

Java
C#
Scala
Kotlin

LAMBDA CON «VOID»

- Dubbio: *dominio o codominio* potrebbero essere il «vuoto»?
- Sì, perché hanno entrambi senso:
  - *dominio vuoto* rappresenta funzioni che *generano* valori
    - in Java, il loro tipo è la specifica interfaccia funzionale *Supplier*
    - in C#, il loro tipo è espresso normalmente da **`Func<`**…**`>`**, avente come *unico argomento il tipo del risultato*
    - in Scala e Kotlin, il loro tipo è espresso semplicemente da **`()->`** …
  - *codominio vuoto* rappresenta *procedure* che *non restituiscono nulla*
    - in Java, il loro tipo è la specifica interfaccia funzionale *Consumer*
    - in C#, il loro tipo è espresso dal delegato **`Action<`**…**`>`**, avente come *unici argomenti i tipi del dominio*
    - in Scala e Kotlin, il tipo del risultato è **`Unit`**

ESEMPI MULTI-LINGUAGGIO

- Funzioni con dominio vuoto

f: void → R *«funzione che genera un valore reale»*

```
DoubleSupplier f = () -> Math.sqrt(2);
```
Java

```
Func<double> f = () => Math.Sqrt(2);
```
C#

```
val f : () => Double = () => Math.sqrt(2);
```
Scala

```
val f : () -> Double = { Math.sqrt(2.0); }
```
Kotlin

- Funzioni con codominio vuoto

p: S → void *«procedura che, data una stringa, la stampa»*

```
Consumer<String> p = s -> System.out.println(s);
```
Java

```
Action<string> p = (s) => Console.WriteLine(s);
```
C#

```
val p : String => Unit = (s) => println(s);
```
Scala

```
val p : (String) -> Unit = { s -> println(s); }
```
Kotlin

CHIAMARE UNA FUNZIONE LAMBDA

- Ora essere stata *definita*, una funzione (lambda) deve anche poter essere *chiamata*
  - *in matematica,* una funzione viene «applicata» ad argomenti scrivendone il nome seguito dagli argomenti fra parentesi
    - ad esempio, se f = x^2+2x+1 (definizione), la sua applicazione all'argomento 3 si scrive f(3)
  - *nei linguaggi di programmazione,* una funzione «classica», ossia definita *senza* notazione lambda, segue circa la stessa sintassi
    - ad esempio, la funzione sopra si può definire scrivendo **`double f(double x){ return x*x+2*x+1;}`** e chiamare con **`f(3)`**
  - *una lambda expression* dovrebbe poter seguire la stessa sintassi, **ma questo *non è sempre vero***
    - in particolare *non è vero in Java*, perché le lambda in tale linguaggio *non sono vere «first class entities» !*

CHIAMARE UNA FUNZIONE LAMBDA

- Una lambda expression
  - può essere chiamata in modo analogo a una funzione «classica», ossia scrivendone il nome seguito dagli argomenti fra parentesi, *se il linguaggio la vede come first-class entity*
    - è il caso di C# Scala Kotlin
  - *deve invece essere chiamata ricorrendo a qualche circonvoluzione* se il linguaggio non ha tale nozione (ma la approssima soltanto)
    - è il caso di Java che deve appoggiarsi a un *metodo ausiliario*
    - MOTIVO: una lambda expression in Java è in realtà un'istanza di una opportuna interfaccia, che internamente dichiara un normale metodo: per chiamare la funzione occorre quindi invocare tale metodo
    - ad esempio, una funzione f: R → R è di tipo **`DoubleUnaryOperator`**, un'interfaccia che dichiara internamente il metodo **`applyAsDouble`**: quindi per chiamare f(3) si deve scrivere **`f.applyAsDouble(3)`**

CHIAMARE UNA LAMBDA ESEMPI MULTI-LINGUAGGIO

- Dichiarazioni complete con tutti i tipi specificati:

```
IntUnaryOperator f = (int x) -> x+3;          Java
Func<int,int>    f = (int x) => x+3;          C#
val f : Int => Int =  x:Int => x+3;           Scala
val f : (Int) -> Int = { x:Int -> x+3; }      Kotlin
```

- Esempi di chiamate (funzione da int a int):

```
int res = f.applyAsInt(3);                    Java
int res = f(3);                               C#
val res = f(3);                               Scala
val res = f(3);                               Kotlin
```

  - purtroppo, in Java il nome del metodo interno all'interfaccia non è fisso: *varia da un'interfaccia funzionale all'altra*

ESEMPI MULTI-LINGUAGGIO

Java

```java
import java.util.function.*;

public class LambdaNew1 {
    public static void main(String[] args) {

        IntUnaryOperator f = (int x) -> x+3;
        IntBinaryOperator g = (int x, int y) -> 2*x-y;
        DoubleBinaryOperator h = (double x, double y) -> x+y*Math.PI;
        System.out.println("f(4) = " + f.applyAsInt(4));                    // 7
        System.out.println("g(3,4) = " + g.applyAsInt(3,4));                // 2
        System.out.println("h(Math.PI,1) = " + h.applyAsDouble(Math.PI,1)); // 2*PI

    }
}
```

JAVA: la lambda ha un tipo *nominale* espresso da un'*interfaccia funzionale*

Inoltre, non si può usare direttamente l'operatore di chiamata `()`

C#

```csharp
public static void Main()
{
    Func<int,int> f = (int x) => x+3;
    Func<int,int,int> g = (int x, int y) => 2*x-y;
    Func<double,double,double> h = (double x, double y) => x+y*Math.PI;
    Console.WriteLine("f(4) = " + f(4));                    // 7
    Console.WriteLine("g(3,4) = " + g(3,4));                // 2
    Console.WriteLine("h(Math.PI,1) = " + h(Math.PI,1));    // 2*PI
}
```

C#: la lambda ha un tipo *nominale* espresso dal *delegato* Func<…>

Si può usare l'operatore di chiamata `()`

Scala

```scala
def main(args:Array[String]):Unit = {

  var f = (x:Int) ⇒ x+3;
  var g = (x:Int, y:Int) ⇒ 2*x-y;
  var h = (x:Double, y:Double) ⇒ x+y*Math.PI;
  println("f(4) = " + f(4));                    // 7
  println("g(3,4) = " + g(3,4));                // 2
  println("h(Math.PI,1) = " + h(Math.PI,1));    // 2*PI

}
```

Kotlin

```kotlin
fun main() {
    var f = {x:Int -> x+3};
    var g = {x:Int, y:Int -> 2*x-y};
    var h = {x:Double, y:Double -> x+y*Math.PI};
    println("f(4) = " + f(4));                    // 7
    println("g(3,4) = " + g(3,4));                // 2
    println("h(Math.PI,1) = " + h(Math.PI,1.0));  // 2*PI
}
```

Scala, Kotlin: la lambda ha un tipo *strutturale* deducibile anche dalla type inference

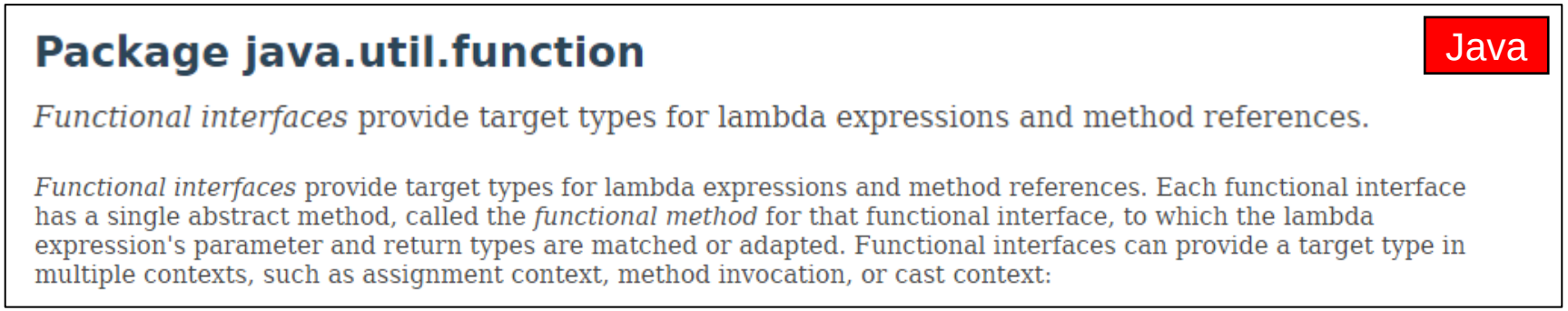
Si può usare l'operatore di chiamata `()`

INTERFACCE FUNZIONALI JAVA vs. DELEGATI C#

- Una *interfaccia funzionale* è una *normale interfaccia* che dichiara un solo metodo — Java
  - si riconosce dall'annotazione `@FunctionalInterface`
  - interfaccia «SAM» = *Single Abstract Method*
  - il metodo dichiarato è quello che *incapsulerà il codice* della funzione
  - per chiamare la funzione occorrerà *chiamare quel metodo*, di cui quindi *occorre conoscere il nome* ☹ → impossibile uso diretto di `()`
  - Java ne definisce moltissime (40+), che occorre conoscere ☹
- Un *delegato* è *simile* a un'interfaccia, ma è adattato e reso *specifico per dichiarare tipi-funzione* — C#
  - internamente anche qui un metodo incapsula il codice della funzione, *ma ciò non traspare esternamente* → possibile uso diretto di `()`
  - C# ne definisce 3 standard, che coprono il 99% dei casi pratici ☺

JAVA: FUNCTIONAL INTERFACES

- Il package **`java.util.function`** definisce *più di 40 interfacce funzionali* per tutti i casi di più largo uso



  - il nome del metodo interno tende a "richiamare l'uso tipico" della funzione (con una certa fantasia..)
    - per le funzioni da A a B («function»), solitamente il metodo si chiama **`apply`** + varianti per tipi primitivi (**`applyAs`*`Tipo`***)
    - per le funzioni da void a B («supplier»), solitamente il metodo si chiama **`get`** + varianti per tipi primitivi (**`getAs`*`Tipo`***)
    - per le funzioni da A a void («consumer»), solitamente il metodo è **`accept`** (curiosamente, *senza* varianti per tipi primitivi: il nome è sempre **`accept`**!)

JAVA: FUNCTIONAL INTERFACES

Consumer	DoubleConsumer	IntConsumer	LongConsumer
Supplier	DoubleSupplier	IntSupplier	LongSupplier
Predicate	DoublePredicate	IntPredicate	LongPredicate
Function	DoubleFunction	IntFunction	LongFunction
BiConsumer BiPredicate BiFunction			
UnaryOperator	DoubleUnaryOperator	IntUnaryOperator	LongUnaryOperator
BinaryOperator	DoubleBinaryOperator	IntBinaryOperator	LongBinaryOperator
	DoubleToIntFunction	IntToDoubleFunction	LongToDoubleFunction
	DoubleToLongFunction	IntToLongFunction	LongToIntFunction
	ToDoubleFunction	ToIntFunction	ToLongFunction
	ToDoubleBiFunction	ToIntBiFunction	ToLongBiFunction
BooleanSupplier	ObjDoubleConsumer	ObjIntConsumer	ObjLongConsumer

OSSERVA: la presenza dei tipi primitivi ci costa il 377% in più in termini di interfacce da definire e ricordare!!

Versione singola per **boolean**

Versioni specializzate per **double**

Versioni specializzate per **int**

Versioni specializzate per **long**

JAVA: FUNCTIONAL INTERFACES

Consumer	**DoubleConsumer**	**IntConsumer**	**LongConsumer**
Supplier	DoubleSupplier	IntSupplier	LongSupplier
Predicate		edicate	LongPredicate
Function		nction	LongFunction
BiConsumer			
BiPredicate			
BiFunction			
UnaryOperator	DoubleUnaryOperator	IntUnaryOperator	LongUnaryOperator
BinaryOperator	DoubleBinaryOperator	IntBinaryOperator	LongBinaryOperator
	DoubleToIntFunction	IntToDoubleFunction	LongToDoubleFunction
	DoubleToLongF		LongToIntFunction
	ToDoubleFunct		ToLongFunction
	ToDoubleBiFun		ToLongBiFunction
BooleanSupplier	**ObjDoubleConsumer**	**ObjIntConsumer**	**ObjLongConsumer**

I **consumatori** sono funzioni a *un argomento*, *che non restituiscono nulla*

Questi consumatori extra però sono funzioni a due argomenti. Anch'essi non restituiscono nulla

JAVA: FUNCTIONAL INTERFACES

Consumer	DoubleConsumer	IntConsumer	LongConsumer
Supplier	**DoubleSupplier**	**IntSupplier**	**LongSupplier**
Predicate		IntPredicate	LongPredicate
Function		IntFunction	LongFunction
BiConsumer			
BiPredicate			
BiFunction			
UnaryOperator	DoubleUnaryOperator	IntUnaryOperator	LongUnaryOperator
BinaryOperator	DoubleBinaryOperator	IntBinaryOperator	LongBinaryOperator
	DoubleToIntFunction	IntToDoubleFunction	LongToDoubleFunction
	DoubleToLongFunction	IntToLongFunction	LongToIntFunction
	ToDoubleFunction	ToIntFunction	ToLongFunction
	ToDoubleBiFunction	ToIntBiFunction	ToLongBiFunction
BooleanSupplier	ObjDoubleConsumer	ObjIntConsumer	ObjLongConsumer

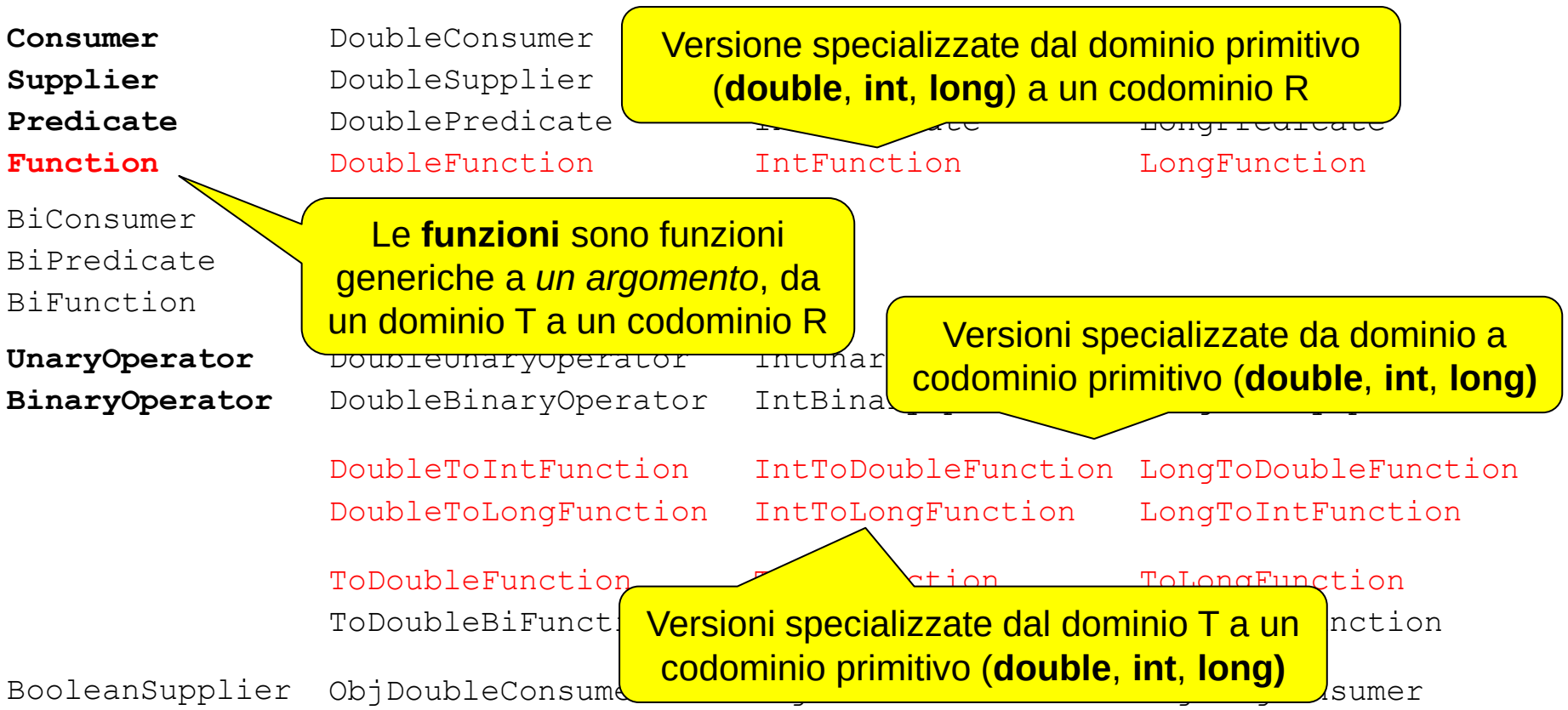
I **fornitori** sono funzioni con *zero argomenti* che *producono un valore*

JAVA: FUNCTIONAL INTERFACES

Consumer	DoubleConsumer	IntConsumer	LongConsumer
Supplier	DoubleSupplier	IntSupplier	LongSupplier
Predicate	DoublePredicate	IntPredicate	LongPredicate
Function		ntFunction	LongFunction
BiConsumer			
BiPredicate			
BiFunction			
UnaryOperator	DoubleUnaryOperator	IntUnaryOperator	LongUnaryOperator
BinaryOperator	DoubleBinaryOperator	IntBinaryOperator	LongBinaryOperator
	DoubleToIntFunction	IntToDoubleFunction	LongToDoubleFunction
	DoubleToLongFunction	IntToLongFunction	LongToIntFunction
	ToDoubleFunction	ToIntFunction	ToLongFunction
	ToDoubleBiFunction	ToIntBiFunction	ToLongBiFunction
BooleanSupplier	ObjDoubleConsumer	ObjIntConsumer	ObjLongConsumer

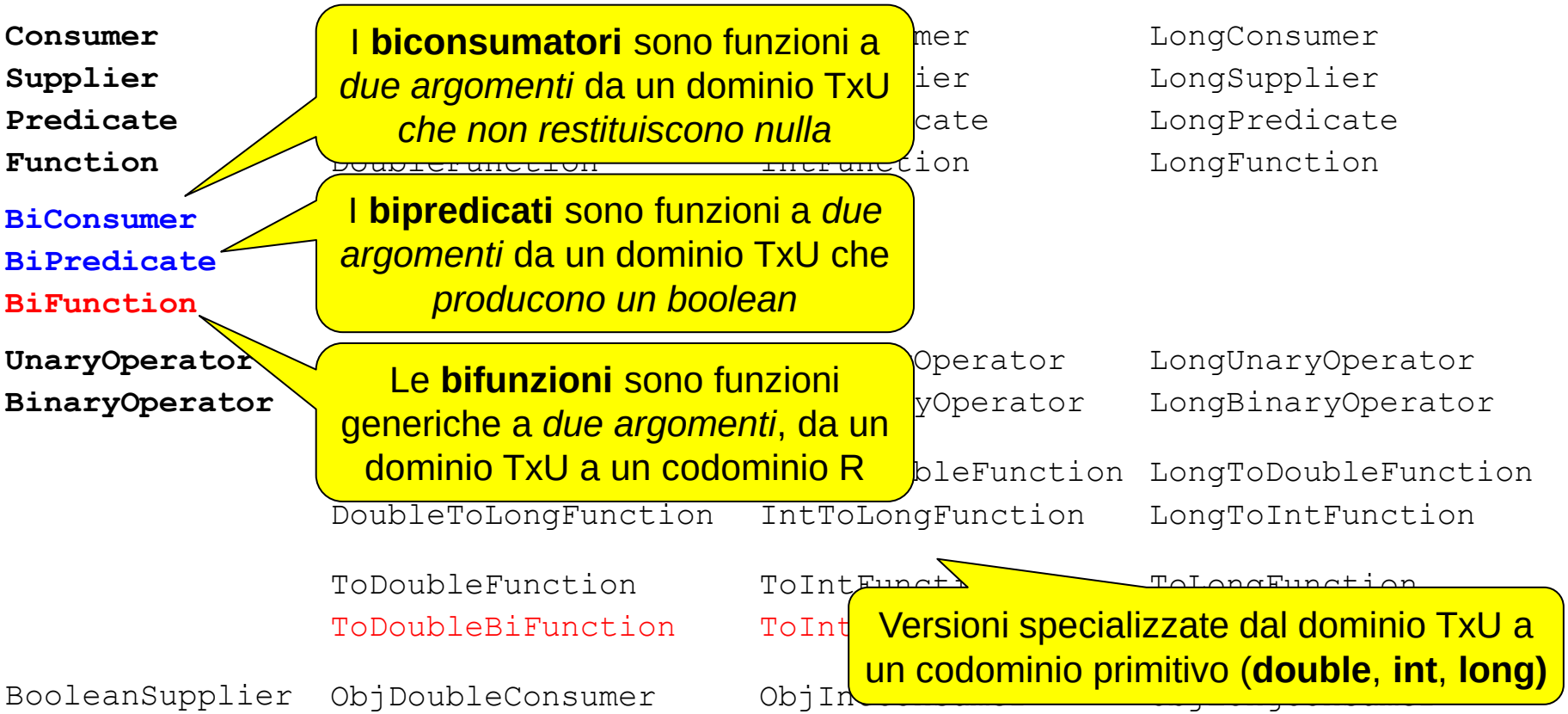
I **predicati** sono funzioni con *un argomento* che *producono un boolean*

JAVA: FUNCTIONAL INTERFACES



NB: non sono previste interfacce funzionali standard per funzioni a più di due argomenti; nel caso, dovranno essere definite dall'utente.

JAVA: FUNCTIONAL INTERFACES



NB: non sono previste interfacce funzionali standard per predicati a più di due argomenti; nel caso, dovranno essere definite dall'utente.

JAVA: FUNCTIONAL INTERFACES

Consumer	DoubleConsumer	IntConsumer	LongConsumer
Supplier	DoubleSupplier	IntSupplier	LongSupplier
Predicate	DoublePredicate	IntPredicate	LongPredicate
Function	DoubleFunction	IntFunction	LongFunction
BiConsumer			
BiPredicate			
BiFunction			
UnaryOperator	DoubleUnaryOperator	IntUnaryOperator	LongUnaryOperator
BinaryOperator	DoubleBinaryOperator	IntBinaryOperator	LongBinaryOperator
		DoubleFunction	LongToDoubleFunction
		LongFunction	LongToIntFunction
	ToDoubleFunction	ToIntFunction	ToLongFunction
	ToDoubleBiFunction	ToIntBiFunction	ToLongBiFunction
BooleanSupplier	ObjDoubleConsumer	ObjIntConsumer	ObjLongConsumer

Gli **operatori unari** sono funzioni a *un argomento* su un dominio T

Versioni specializzate su domini primitivi (**double**, **int**, **long**)

Gli **operatori binari** sono funzioni a *due argomenti* su un dominio T

NB: non sono previste interfacce funzionali standard per operatori a più di due argomenti; nel caso, dovranno essere definiti dall'utente.

JAVA: LAMBDA EXPRESSION «UNDER THE HOOD»

- In Java ogni lambda ha il tipo di un'opportuna interfaccia funzionale, in molti casi (ma non tutti!) *deducibile dal compilatore*

ESEMPIO

Java

La lambda expression

```
(int x) -> x+1
```

è un'istanza dell'interfaccia funzionale **`IntUnaryOperator`**

Poiché tale interfaccia dichiara il metodo:

```
int applyAsInt(int operand)
```

il compilatore *traduce la lambda expression in una classe anonima* che implementa tale metodo:

```
new IntUnaryOperator(){
   public int applyAsInt(int x){ return x+1; }
}
```

C#: DELEGATI

- In C# ogni lambda ha il tipo di un opportuno delegato – una sorta di interfaccia *chiamabile* tramite l'operatore `()`
- Esistono due *delegati standard* fondamentali:
  - **`Func<`T_1`,`T_2`,...`T_N`,` T_{RET}`>`** esprime l'idea di *funzione a N argomenti* (dominio) di tipo $T_1, T_2, \ldots T_N$ e tipo di ritorno (codominio) T_{RET}
    - caso particolare: **`Func<`**T_{RET}**`>`** esprime l'idea di *funzione senza argomenti* (dominio void) e tipo di ritorno (codominio) T_{RET}
  - **`Action<`**$T_1, T_2, \ldots T_N$**`>`** esprime l'idea di *funzione a N argomenti* (dominio) di tipo $T_1, T_2, \ldots T_N$ e tipo di ritorno (codominio) vuoto
- A questi si aggiungono altri *delegati standard* di complemento
  - **`Predicate<`**$T_1, T_2, \ldots T_N$**`>`** esprime l'idea di *funzione a N argomenti* (dominio) di tipo $T_1, T_2, \ldots T_N$ e tipo di ritorno (codominio) *boolean*
  - **`EventHandler<T>`** esprime l'idea di *funzione a un argomento* usata per gestire eventi

ASSEGNARE E USARE LE LAMBDA (1/5)

- Assegnare una lambda (versione con tutti i tipi specificati)

```
IntUnaryOperator f = (int x) -> x+3;
Func<int,int>    f = (int x) => x+3;
val f : Int => Int =  x:Int => x+3;
val f : (Int) -> Int = { x:Int -> x+3; }
```

In Java e C# non si può usare **var** nelle lambda: il target type deve essere *sempre esplicito*

Java C# Scala Kotlin

- Assegnare una lambda (versione con type inference)

```
IntUnaryOperator f = x -> x+3;
Func<int,int>    f = x => x+3;
val f : Int => Int = x => x+3;
val f = x:Int => x+3;
val f : (Int) -> Int = { x -> x+3; }
val f = { x:Int -> x+3; }
```

In Java e C# non si può usare **var** nelle lambda: il target type deve essere *sempre esplicito*

Java C# Scala Kotlin

ASSEGNARE E USARE LE LAMBDA (2/5)

- Chiamare una lambda

```
System.out.println( f.applyAsInt(4) ); // ARGH!
Console.WriteLine( f(4) );
println( f(4) );
println( f(4) );
```

Java C# Scala Kotlin

7

  - in Java, purtroppo, la vera natura della lambda come «istanza di una classe anonima» emerge al momento della chiamata, rendendo *impossibile l'uso diretto dell'operatore* `()`
  - per questo, in Java le lambda non sono «veri tipi funzione»
  - la necessità di doversi *ricordare il nome del metodo interno* con le ulteriori sotto-varianti dovute ai tipi primitivi peggiora le cose

ASSEGNARE E USARE LE LAMBDA (3/5)

- Accettare una lambda come argomento

```
void myCalc(IntUnaryOperator f){…}
void myCalc(Func<int,int> f){…}
def myCalc(f: Int=>Int ) : Unit = {…}
fun myCalc(f: (Int)->Int ) : Unit {…}
```

Da stabilire cosa faccia **myCalc**

Java | C# | Scala | Kotlin

- Chiamare una funzione passando una lambda

```
myCalc((int x) -> x+1); // o anche: x -> x+1
myCalc((int x) => x+1); // o anche: x => x+1
myCalc((x:Int) => x+1); // o anche: x => x+1
myCalc({x:Int -> x+1}); // o anche: {x -> x+1}
```

Java | C# | Scala | Kotlin

  - l'output dipende da cosa fa **myCalc**: se ad esempio essa stampa il risultato di **f(5)**, passando questa lambda vedremo in output 6

ASSEGNARE E USARE LE LAMBDA (4/5)

- Definire funzioni con dominio vuoto

f: void → R *«funzione che genera un valore reale»*

Java
```
DoubleSupplier f = () -> Math.sqrt(2);
```

C#
```
Func<double> f = () => Math.Sqrt(2);
```

Scala
```
val f : () => Double = () => Math.sqrt(2);
```

Kotlin
```
val f : () -> Double = { Math.sqrt(2.0); }
```

- Chiamare funzioni con dominio vuoto

Java
```
System.out.println( f.getAsDouble() );
```

C#
```
Console.WriteLine( f() );
```

Scala
```
println( f() );
```

Kotlin
```
println( f() );
```

In Java occorre conoscere il nome del metodo sottostante

ASSEGNARE E USARE LE LAMBDA (5/5)

- Definire funzioni con codominio vuoto

p: S → void *«procedura che, data una stringa, la stampa»*

```
Consumer<String> p = s -> System.out.println(s);
```
Java

```
Action<string> p = (s) => Console.WriteLine(s);
```
C#

```
val p : String => Unit = (s) => println(s);
```
Scala

```
val p : (String) -> Unit = { s -> println(s); }
```
Kotlin

- Chiamare funzioni con codominio vuoto

```
p.accept("Hello World");
```
Java

```
p("Hello World");
```
C#

```
p("Hello World");
```
Scala

```
p("Hello World");
```
Kotlin

In Java occorre conoscere il nome del metodo sottostante

SCORCIATOIE PER LAMBDA

- Spesso le lambda sono molto semplici, limitandosi a:
  1. *richiamare un metodo esistente, senza argomenti*

     Esempio: `x -> x.getValue()`
  2. *fare semplici elaborazioni, senza argomenti*

     Esempio: `x -> x+1`
- In questi casi, i vari linguaggi rendono disponibili *scorciatoie* che permettono, secondo i casi, di:
  - non scrivere proprio la lambda → *method reference*
  - scrivere solo il lato destro → *parametri impliciti*

Java Kotlin

Scala Kotlin

METHOD REFERENCE

- Se la lambda si limita a *richiamare un metodo già esistente*, senza argomenti, si può sostituire con un *riferimento al metodo originario:* una *method reference*
- La sintassi si basa sul nuovo operatore `::`

Java Kotlin

NomeClasse*`::`*nomeMetodo
NomeIstanza*`::`*nomeMetodo

  - la forma ***NomeClasse*`::`*nomemetodo*** è utilizzabile per ogni *metodo pubblico di una classe*
  - la forma ***NomeIstanza*`::`*nomemetodo*** è utilizzabile anche per *metodi privati di specifici oggetti:* in particolare,
    - in Java **`this::method`** equivale a **`this -> this.method()`**
    - in Kotlin ***NomeIstanza*** può anche essere assente (=**`this`**)

METHOD REFERENCE: ESEMPI

- ESEMPI Java Kotlin

  `Counter::getValue` sostituisce la lambda **`x -> x.getValue()`**

  `Persona::getNome` sostituisce la lambda **`p -> p.getNome()`**

  `Integer::toString` sostituisce la lambda **`i -> i.toString()`**

  NB:

  - non c'è l'operatore di chiamata `()`, solo il *nome del metodo*
  - in Java ***`NomeClasse`*** può essere anche un array, come **`Counter[]`**
  - occorre evitare ambiguità: la type inference deve poter determinare in modo univoco il tipo degli argomenti – altrimenti, conflitto!
    - esempio: **`Integer::toString`**, perché in **`Integer`** convivono sia il *metodo* **`toString`** sia la *funzione statica* **`toString`**
    - non è detto che tale compresenza dia effettivamente luogo ad ambiguità: dipende dal contesto d'uso, caso per caso

PARAMETRI IMPLICITI

- Un'altra scorciatoia consiste nello *scrivere solo il lato destro della lambda,* evitando completamente il resto
- La sintassi si basa sull'uso di un apposito *placeholder*
  - in Scala, `_` sostituisce l'i-esimo argomento Scala
  - in Kotlin, `it` sostituisce il primo (e unico) argomento Kotlin
- ESEMPI Scala

  `_.getValue()` sostituisce la lambda `c => c.getValue()`

  `_+1` sostituisce la lambda `n => n+1`
- ESEMPI Kotlin

  `{it.getValue()}` sostituisce la lambda `{c -> c.getValue()}`

  `{it+1}` sostituisce la lambda `{n -> n+1}`

PARAMETRI IMPLICITI

- In Scala, il *placeholder* **_** può anche sostituire *contempora-neamente* più argomenti
- ESEMPI Scala

```
val sum = (_: Int) + (_: Int)
```
sostituisce la lambda **`(x,y) => x+y`**

```
val mul : (Int,Int)=>Int = _*_
```
sostituisce la lambda **`(x,y) => x*y`**

Il tipo degli argomenti va specificato o negli argomenti (come in **`sum`**), o nel tipo della lambda (come in **`mul`**)

ATTENZIONE: ogni occorrenza di **_** sostituisce *un diverso* argomento

```
val pow : Int=>Int = _*_
```
NO, errata: **`_*_`** *vale come* **`x*y`**, *non come* **`x*x`** *!*

Un classico caso applicativo
Back to Comparator

ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Riprendiamo la «solita» classe **`Persona`** (non **`Comparable`**):

Java

```
class Persona {
 private String nome, cognome;
 private int età;
 ...
 public String getCognome(){ return cognome; }
 public String getNome(){ return nome; }
 public int getEtà(){ return età; }
}
```

- … e il «solito» array di persone:

Java

```
Persona[] persone = new Persona[]{
    new Persona("John", "Doe", 25),
    new Persona("Jane", "Doe", 45),
    new Persona("Anne", "Bee", 31),
    new Persona("Jane", "Doe", 22)
};
```

ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Servono tanti **`Comparator`** quanti i criteri di ordinamento:

```
class CognomeComparator implements Comparator<Persona> {
   public int compare(Persona p1, Persona p2){
    return p1.getCognome().compareTo(p2.getCognome());
   }
}
class NomeComparator implements Comparator<Persona>{
   public int compare(Persona p1, Persona p2){
    return p1.getNome().compareTo(p2.getNome());
   }
}
class EtaComparator implements Comparator<Persona>{
   public int compare(Persona p1, Persona p2){
    return Integer.compare(p1.getEtà(), p2.getEtà());
   }
}
```

Java

ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Per ordinare l'array si usare la funzione **`Arrays.sort`**:

Java

```
// versione base con comparatori
System.out.println(Arrays.asList(persone));
Arrays.sort(persone, new CognomeComparator() );
System.out.println(Arrays.asList(persone));
Arrays.sort(persone, new EtaComparator() );
System.out.println(Arrays.asList(persone));
```

- Funziona, ma:
  - costringe a creare N classi (una per ogni criterio)
  - ognuna delle quali è istanziata e usata una sola volta

ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Si possono usare le classi anonime, ma è verboso:

Java

```
// versione con comparatori e classi anonime
System.out.println(Arrays.asList(persone));
Arrays.sort(persone, new Comparator<Persona> {
   public int compare(Persona p1, Persona p2){
    return p1.getCognome().compareTo(p2.getCognome());
   }
 });
System.out.println(Arrays.asList(persone));
Arrays.sort(persone, new Comparator<Persona>{
   public int compare(Persona p1, Persona p2){
    return Integer.compare(p1.getEtà(), p2.getEtà());
   }
 });
System.out.println(Arrays.asList(persone));
```

ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- La lambda expression può fare la stessa cosa con eleganza: ogni lambda equivale a un *comparatore anonimo*

Java

```
// versione con lambda expression
System.out.println(Arrays.asList(persone));
Arrays.sort(persone,
    (p1,p2) -> p1.getCognome().compareTo(p2.getCognome()) );
System.out.println(Arrays.asList(persone));

Arrays.sort(persone,
    (p1,p2) -> Integer.compare(p1.getEtà(), p2.getEtà()) );
System.out.println(Arrays.asList(persone));
```

equivale a

```
Comparator<Persona> cmp = new Comparator<Persona> {
    public int compare(Persona p1, Persona p2){
        return p1.getCognome().compareTo(p2.getCognome());
    }
}
```

ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- La lambda expression può fare la stessa cosa con eleganza: ogni lambda equivale a un *comparatore anonimo*

Java

```
// versione con lambda expression
System.out.println(Arrays.asList(persone));
Arrays.sort(persone,
   (p1,p2) -> p1.getCognome().compareTo(p2.getCognome()) );
System.out.println(Arrays.asList(persone));

Arrays.sort(persone,
   (p1,p2) -> Integer.compare(p1.getEtà(), p2.getEtà()) );
System.out.println(Arrays.asList(persone));
```

equivale a

```
Comparator<Persona> cmp = new Comparator<Persona> {
    public int compare(Persona p1, Persona p2){
        return Integer.compare(p1.getEtà(), p2.getEtà());
    }
}
```

UN PASSO IN PIÙ

- Non è finita: osservando bene, si nota che la struttura delle varie lambda-comparatori è *sostanzialmente identica*
  - tutti i comparatori confrontano due oggetti, restituendo +1, -1 o 0
  - l'unica cosa che cambia è *il valore da confrontare*
    - nel primo caso, *il cognome* → metodo **`getCognome`**
    - nel secondo caso, *l'età* → metodo **`getEtà`**

Java

```
System.out.println(Arrays.asList(persone));
Arrays.sort(persone,
   (p1,p2) -> p1.getCognome().compareTo( p2.getCognome( )));
System.out.println(Arrays.asList(persone));


Arrays.sort(persone,
   (p1,p2) -> Integer.compare( p1.getEtà(), p2.getEtà() ));
System.out.println(Arrays.asList(persone));
```

UN COMPARATORE «GENERALE»

- In effetti, ogni comparatore ha una struttura fissa: *cambia solo la proprietà su cui si effettua il confronto*

```
(x,y) -> x.getProp().compareTo(y.getProp())
```

- Ma allora.. se fattorizziamo in una piccola lambda (*estrattore*) la proprietà di interesse per il confronto:

```
x -> x.getProp()
```

- Il comparatore può essere *sintetizzato da una factory* che riceva appunto *l'estrattore* della proprietà di interesse!

```
Comparator.comparing(estrattore)
```

LA FABBRICA DEI COMPARATORI

- Il metodo statico **`comparing`** è la factory dei comparatori
  - si chiama così perché la frase "viene bene in inglese"
  - riceve l'estrattore (mini-lambda) della *proprietà da confrontare*

PRIMA: Java

```
Arrays.sort(persone,
   (p1,p2) -> p1.getCognome().compareTo( p2.getCognome( )));
```

DOPO:

```
Arrays.sort(persone, Comparator.comparing( p -> p.getCognome() );
```

  - in inglese: sort «by means of a comparator *comparing* this property»
  - in questo caso: ordina «*confrontando per cognome*»

- Ciliegina sulla torta: l'estrattore come *method reference* ☺

```
Arrays.sort(persone, Comparator.comparing( Persona::getCognome );
```

COMPARATORS, AT LAST!

- Ora la transizione è completa
  - anziché definire una classe-comparatore ad hoc, da istanziare appositamente per quella **`sort`**:

```java
class CognomeComparator implements Comparator<Persona> {
   public int compare(Persona p1, Persona p2){
    return p1.getCognome().compareTo(p2.getCognome());
   }
}
Arrays.sort(persone, new CognomeComparator() );
```

Java

  - anziché l'intrico innestato delle classi anonime
  - un *approccio pulito, auto-documentante e leggibile,* in cui si *riconosce subito* cosa si fa

```java
Arrays.sort(persone,
   Comparator.comparing( Persona::getCognome );
```

Java

COMPARATORI PER TIPI PRIMITIVI

- Anche in questo caso, come sempre in Java, i tipi primitivi, purtroppo, richiedono un trattamento ad hoc
  - tre factory method specifici per i tre più usati: **`int`**, **`long`** e **`double`**
  - **`comparingInt`**, **`comparingLong`**, **`comparingDouble`**

PRIMA: Java

```
Arrays.sort(persone,
    (p1,p2) -> Integer.compare( p1.getEtà(), p2.getEtà() ));
```

DOPO:

```
Arrays.sort(persone,
   Comparator.comparingInt( p -> p.getEtà() );
```

COMPARATORI MULTIPLI

- Si possono perfino applicare *criteri di ordinamento multipli*
  - prima un criterio principale (ad esempio, ordinamento per cognome)
  - poi un criterio secondario (ad esempio, ordinamento per nome)
  - poi un criterio ulteriore (ad esempio, ordinamento per età)
- Basta *combinare più comparatori* con **`thenComparing`**

Java

```
Arrays.sort(persone,
   Comparator.comparing( Persona::getCognome )
             .thenComparing( Persona::getNome )
             .thenComparingInt( Persona::getEtà );
```

  - *Fluent interface* ottenuta tramite *cascading:*
    confronta per cognome *e in subordine* per nome *e poi* per età
  - Funziona perché **`comparing`** restituisce il **`Comparator`** creato, usato poi come target dalla successiva **`thenComparing`**

…BECAUSE READABILITY MATTERS

- Giusto per apprezzare la differenza…
  - con factory, lambda come extractor e method reference:

Java

```
Arrays.sort(persone,
   Comparator.comparing( Persona::getCognome )
             .thenComparing( Persona::getNome )
             .thenComparingInt( Persona::getEtà );
```

  - la stessa cosa con l'approccio classico:

Java

```
class MyComparator implements Comparator<Persona> {
   public int compare(Persona p1, Persona p2){
    boolean res = p1.getCognome().compareTo(p2.getCognome());
    if (res!=0) return res;
    res = p1.getNome().compareTo(p2.getNome());
    if (res!=0) return res;
    else return Integer.compare( p1.getEtà(), p2.getEtà() )
   }
}
```

…BECAUSE READABILITY MATTERS

- Giusto per apprezzare la differenza…
  - con factory, lambda come extractor e method reference:

```
Arrays.sort(persone,
   Comparator.comparing( Persona::getCognome )
             .thenComparing( Persona::getNome )
             .thenComparingInt( Persona::getEtà );
```

- Espressività
- Estendibilità
- Manutenibilità

  - la stessa cosa con l'approccio classico:

```
class MyComparator implements Comparator<Persona>
   public int compare(Persona p1, Persona p2){
    boolean res = p1.getCognome().compareTo(p2.getCognome());
    if (res!=0) return res;
    res = p1.getNome().compareTo(p2.getNome());
    if (res!=0) return res;
    else return Integer.compare( p1.getEtà(), p2.getEtà() )
   }
}
```

- Cripticità
- Manutenibilità faticosa

COMPARATORI MULTIPLI: LIMITI

- ATTENZIONE: talora la type inference *può non farcela* a dedurre con certezza il tipo della lambda *se il contesto non dà sufficienti informazioni*
- Ad esempio, mentre questa compila regolarmente:

```
Collections.sort(playlist, comparing(p1 -> p1.getTitle()));
```

il comparatore multiplo seguente *non* compila:

Java

```
Collections.sort(playlist,
    comparing(p1 -> p1.getTitle())
    .thenComparing(p1 -> p1.getDuration())
    .thenComparing(p1 -> p1.getArtist())   );
```

- MOTIVO: nel primo caso, il tipo del risultato di **`comparing`** è ricavato dalla signature di **`Collections.sort`**: poiché **`playlist`** è **`List<Song>`**, **`comparing`** deve fornire un **`Comparator<Song>`**, quindi **`p1`** dev'essere una **`Song`**.

COMPARATORI MULTIPLI: LIMITI

- ATTENZIONE: talora la type inference *può non farcela* a dedurre con certezza il tipo della lambda *se il contesto non dà sufficienti informazioni*
- Ad esempio, mentre questa compila regolarmente:

```
Collections.sort(playlist, comparing(p1 -> p1.getTitle()));
```

il comparatore multiplo seguente *non* compila:

Java

```
Collections.sort(playlist,
    comparing(p1 -> p1.getTitle())
    .thenComparing(p1 -> p1.getDuration())
    .thenComparing(p1 -> p1.getArtist())   );
```

- MOTIVO: nel secondo caso, **`Comparator<Song>`** è il tipo del risultato *dell'intera catena,* ma perché funzioni occorre *riuscire a inferire tutti i singoli pezzi.*
- Ora, il metodo **`thenComparing`** è invocato sul risultato di **`comparing`**, ma *poiché questa è generica*, non se ne conosce il tipo: quindi, non si può essere certi neppure del fatto che *abbia un metodo* **`thenComparing`**!

COMPARATORI MULTIPLI: LIMITI

- Per uscirne bisogna *aggiungere in qualche modo informazione di tipo*

Java

```
Collections.sort(playlist, // non compila!
    comparing(p1 -> p1.getTitle())
    .thenComparing(p1 -> p1.getDuration())
    .thenComparing(p1 -> p1.getArtist())   );
```

- Ci sono fondamentalmente tre possibilità:
  - ☺ ☺ usare una method reference specifica, come `Song::getTitle`
  - ☺ usare una lambda con argomenti tipizzati, come `(Song p1) ->` …
  - 😐 fornire la specifica di tipo in sede di chiamata di **`comparing`**, scrivendo `Comparator.<Song,String>`**`comparing(...)`**

COMPARATORI MULTIPLI: LIMITI

- ☺☺ usare una method reference specifica in tipo:

```
Collections.sort(playlist,
    comparing(Song::getTitle)
    .thenComparing(p1 -> p1.getDuration())
    .thenComparing(p1 -> p1.getArtist())   );
```
Java

- ☺ Usare una lambda con argomenti tipizzati

```
Collections.sort(playlist,
    comparing((Song p1) -> p1.getTitle())
    .thenComparing(p1 -> p1.getDuration())
    .thenComparing(p1 -> p1.getArtist())   );
```
Java

- 😐 fornire la specifica di tipo in sede di chiamata di **`comparing`**:

```
Collections.sort(playlist,
    Comparator.<Song,String>comparing(p1 -> p1.getTitle())
    .thenComparing(p1 -> p1.getDuration())
    .thenComparing(p1 -> p1.getArtist())   );
```
Java

E NEGLI ALTRI LINGUAGGI…?

- In C#
  - la fabbrica dei comparatori è **`Comparer<T>.Create`**
  - l'ordinamento *su nuovo array* via lambda lo fa il metodo **`OrderBy`**

C#

```
Array.Sort(persone, new MyComp()); // approccio classico

Array.Sort(persone, Comparer<Persona>.Create(
        (a,b) => a.GetNome().CompareTo(b.GetNome()) ));

IOrderedEnumerable<Counter> orderedArray =
        persone.OrderBy( p => p.GetNome() );
```

E NEGLI ALTRI LINGUAGGI…?

- In Scala
  - non c'è una «fabbrica di comparatori», si usano appositi metodi
  - l'ordinamento «in place» via lambda lo fa il metodo **`sortWith`**
  - l'ordinamento *su nuovo array* via lambda lo fa il metodo **`sortBy`**

Scala

```
Sorting.quickSort(persone)(MyComp); // classico

Sorting.quickSort(persone)(new Ordering[Persona] {
   def compare(a: Persona, b: Persona) : Int =
        a.getNome() compare b.getNome();
   })

val sortedPersons = persone.sortWith(
   (a:Persona,b:Persona) => a.getNome() < b.getNome() );

val sortedPersons = persone.sortBy( (p:Persona) => p.getNome());

val sortedPersons = persone.sortBy( _.getNome() );
```

E NEGLI ALTRI LINGUAGGI…?

- In Kotlin
  - la «fabbrica di comparatori» è embedded nel metodo **`compareBy`**
  - l'ordinamento «in place» via lambda lo fa il metodo **`sortWith`**

Kotlin

```
persone.sortWith(new MyComp()); // classico

persone.sortWith( object: Comparator<Persona> {// object anonimo
   override fun compare(a: Persona, b: Persona) : Int {
        if (a.getNome() < b.getNome()) return -1;
        if (a.getNome() > b.getNome()) return +1;
        /* else */ return 0;
   });

persone.sortWith( Comparator<Counter> { a,b -> // classe anonima
        if (a.getValue()%24 < b.getValue()%24) -1 else
        if (a.getValue()%24 > b.getValue()%24) +1 else
        0;
   });
```

E NEGLI ALTRI LINGUAGGI…?

- In Kotlin
  - la «fabbrica di comparatori» è embedded nel metodo **`compareBy`**
  - l'ordinamento «in place» via lambda lo fa il metodo **`sortWith`**
  - l'ordinamento *su nuovo array* lo fa l'analogo metodo **`sortedWith`**

Kotlin

```
persone.sortWith(new MyComp()); // classico

persone.sortWith( compareBy { p -> p.getNome() } );

persone.sortWith( compareBy { it.getNome() } );


val mySorted = persone.sortedWith( compareBy {
                                        it.getCognome().length });
```

E NEGLI ALTRI LINGUAGGI…?

- In Kotlin
  - nel caso in cui basti specificare la proprietà su cui ordinare, senza altre elaborazioni, si usano i metodi-scorciatoia **`sortedBy`** e **`sortedByDescending`** che accettano direttamente *l'estrattore*
  - NB: si ricorda che esiste anche il metodo **`sorted`** che opera con l'ordinamento naturale, senza argomenti

Kotlin

```
val mySorted = persone.sortedBy { it.getNome() };

val mySorted = persone.sortedByDescending { it.getCognome() };

val stndSorted = persone.sorted;
```

COMPARATORI MULTIPLI NEGLI ALTRI LINGUAGGI

- In C#
  - la coppia di metodi **`OrderBy`** / **`ThenBy`** consente di comporre facilmente ordinamenti multipli *su collection* tramite lambda:

C#

```
IOrderedEnumerable<Persona> listaOrdinata =
          listaPersone.OrderBy( p => p.GetEta());

IOrderedEnumerable<Persona> listaOrdinata =
          listaPersone.OrderBy( p => p.GetCognome() )
                      .ThenBy( p => p.GetNome() );

IOrderedEnumerable<Persona> listaOrdinata =
          listaPersone.OrderBy( p => p.GetCognome() )
                      .ThenBy( p => p.GetNome() )
                      .ThenBy( p => p.GetEta() );
```

  - NB: lavorare su array è possibile, ma **`OrderBy`** restituisce comunque un **`IOrderedEnumerable`**: quindi il risultato va ri-convertito esplicitamente in array tramite **`toArray()`**

COMPARATORI MULTIPLI NEGLI ALTRI LINGUAGGI

- In Scala
  - l'approccio è diverso: il metodo tipizzato **`Ordering.on`** accetta una lambda *«a tuple di proprietà»* di *cardinalità variabile*

Scala

```
Sorting.quickSort(elencoPersone)(
   Ordering[(String,String)].on(p => (p.cognome, p.nome) ))

Sorting.quickSort(elencoPersone)(
   Ordering[(String,Int)].on(p => (p.cognome, p.eta) ))
```

    - una tupla è una struttura del tipo **`(item1,item2,…)`** il cui tipo è **`(tipoItem1,tipoItem2`**, …**`)`**: quindi la tupla *(cognome, nome)* ha tipo *(string, string)* mentre la tupla *(cognome, età)* ha tipo *(string, int)*
    - il metodo **`on`** ordina *sugli item della tupla* nel loro ordine: ad esempio, la tupla *(cognome, età)* ordina per cognome e in subordine per età
  - NB: il tratto **`Ordering`** espone anche i metodi **`thenComparing`** di Java, ma il loro uso diretto causa spesso difficoltà di disambiguazione alla type inference

COMPARATORI MULTIPLI NEGLI ALTRI LINGUAGGI

- In Kotlin
  - basta *specificare più lambda* nella factory **`compareBy`**

Kotlin

```
val mySorted = persone.sortedWith( compareBy(
                    { it.getNome() }, { it.getEta() } ))

val mySorted = persone.sortedWith( compareBy(
                    (Persona::getNome, Persona::getEta) )
```

  - nel caso di proprietà (**`val`**/**`var`**), è ancora più semplice:

Kotlin

```
val sortedCounters = myCounterArray.sortedWith(
                          compareBy( { it.value } ))

// se nome ed età fossero proprietà anziché metodi
val mySorted = persone.sortedWith( compareBy(
                          { it.nome }, { it.eta } )
```

Method reference a costruttori (Costructor reference)

METHOD REFERENCE A COSTRUTTORI

- Abbiamo visto che le method reference seguono la sintassi
  Nomeclasse::nomemetodo
- Un caso particolare è la method reference a *costruttori*
  - è particolare perché il costruttore non ha un «vero nome» suo, quindi serve una convenzione su "come chiamarlo"
- In Java, la «grande idea» è stata chiamarlo **`new`**
  - sintassi: ***`Nomeclasse::new`***
  - OCCHIO: non significa che si stia creando qualcosa, è solo un «*bel nome evocativo*» per «riferirsi» a un costruttore

Java

- In Kotlin, il nome… è la classe, senza nulla davanti
  - sintassi: ***`::NomeClasse`***

Kotlin

METHOD REFERENCE A COSTRUTTORI

- Con la sintassi della method reference a costruttori
  - Java: ***`Nomeclasse::new`*** Java
  - Kotlin: ***`::NomeClasse`*** Kotlin

  si può fare riferimento a tutti i costruttori di una classe
- Per distinguerli occorre *specificare il tipo* della funzione
  - Java: specificando l'interfaccia funzionale attesa Java
  - Kotlin: specificando il tipo strutturale atteso Kotlin
- Più precisamente, in base alla lista degli argomenti:
  - Java: il tipo è **`Supplier<T>`**, **`Function<U,T>`**, etc. Java
  - Kotlin: il tipo è **`()->T`**, **`(U)->T`**, **`(U,V)->T`**, etc. Kotlin

METHOD REFERENCE A COSTRUTTORI

- Più in dettaglio:
  - in Java, per una classe T: Java
    - il tipo del costruttore a 0 argomenti è `Supplier<T>`
    - il tipo del costruttore a 1 argomento U è `Function<U,T>`
    - il tipo del costruttore a 2 argomenti U,V è `BiFunction<U,V,T>`
    - per costruttori a 3+ argomenti bisogna definirsi le proprie interfacce:

```
interface TriFunction<T, U, V, R> {
  R apply(T t, U u, V v);
}
```

  - in Kotlin, il tipo è strutturale, quindi il tipo del costruttore è semplicemente `(Tipo1,Tipo2,…) -> T` Kotlin
    - il costruttore a 0 argomenti usa la lista vuota: `() -> T`

ESEMPIO (1/4)

- Si supponga di avere una classe **`Persona`** con 5 costruttori:
  - la versione Kotlin, del tutto simile, non viene mostrata per brevità

Java

```
class Persona{
 private String cognome, nome;
 private int eta;
 private boolean sex;
 public Persona(String nome, String cognome, int eta, boolean sex){
   this.nome=nome; this.cognome=cognome; this.eta=eta;
   this.sex=sex;
 }
 public Persona(String nome, String cognome, int eta){
   this(nome, cognome, 1, true); }
 public Persona(String cognome, String nome){ this(nome, cognome, 1); }
 public Persona(String cognome){ this("boh", cognome, 1); }
 public Persona(){ this("boh", "nnn");  }
}
```

ESEMPIO (2/4)

- Quei cinque costruttori hanno i seguenti tipi:

Java

```
Persona(String nome, String cognome, int eta, boolean sex)
```
tipo Java: `???`
tipo Kotlin: `(String,String,Int,Boolean)->Persona`

```
Persona(String nome, String cognome, int eta)
```
tipo Java: `???`
tipo Kotlin: `(String,String,Int)->Persona`

```
Persona(String cognome, String nome)
```
tipo Java: `BiFunction<String,String,Persona>`
tipo Kotlin: `(String,String)->Persona`

```
Persona(String cognome)
```
tipo Java: `Function<String,Persona>`
tipo Kotlin: `(String)->Persona`

```
Persona()
```
tipo Java: `Supplier<Persona>`
tipo Kotlin: `()->Persona`

ESEMPIO (2/4)

- Quei cinque costruttori hanno i seguenti tipi:

```
Persona(String nome, String cognome, int eta, boolean sex)
```

Java

tipo Java: `???`

tipo Kotlin: `(String,` ona

Non esiste un'interfaccia funzionale standard a 3 o 4 argomenti

```
Persona(String nome, String cognome, int eta)
```

tipo Java: `???`

tipo Kotlin: `(String,`

Non esiste un'interfaccia funzionale standard a 3 o 4 argomenti

```
Persona(String cognome, String nome)
```

tipo Java: `BiFunction<String,String,Persona>`

tipo Kotlin: `(String,String)->Persona`

```
Persona(String cognome)
```

tipo Java: `Function<String,Persona>`

tipo Kotlin: `(String)->Persona`

```
Persona()
```

tipo Java: `Supplier<Persona>`

tipo Kotlin: `()->Persona`

ESEMPIO (3/4)

- Quei cinque costruttori hanno i seguenti tipi:

Java

```
Persona(String nome, String cognome, int eta, boolean sex)
```

tipo Java: `QuadriFunction<String,String,Integer,Boolean,Persona>`

tipo Kotlin: `(String,String,Int,Boolean)->Persona`

```
Persona(String nome, String cognome, int eta)
```

tipo Java: `TriFunction<String,String,Integer,Persona>`

tipo Kotlin: `(String,String,Int)->Persona`

```
Persona(Stri
                                                          tsona>
Persona(St
Persona()
```

Ma, se servono, possiamo sempre definirle noi:

```
interface TriFunction<T, U, V, R> {
  R apply(T t, U u, V v);
}
interface QuadriFunction<T, U, V, W, R> {
  R apply(T t, U u, V v, W w);
}
```

ESEMPIO (4/4)

- Ora possiamo catturarli con opportune method reference:

Java

```
Supplier<Persona>                          ctor0 = Persona::new;
Function<String,Persona>                   ctor1 = Persona::new;
BiFunction<String,String,Persona>          ctor2 = Persona::new;

// nostre interfacce funzionali «custom»
TriFunction<String,String,Integer,Persona>
                                           ctor3 = Persona::new;

QuadriFunction<String,String,Integer,Boolean,Persona>
                                           ctor4 = Persona::new;
```

Kotlin

```
val ctor0: () -> Persona = ::Persona;
val ctor1: (String) -> Persona = ::Persona;
val ctor2: (String,String) -> Persona = ::Persona;
val ctor3: (String,String,Int) -> Persona = ::Persona ;
val ctor4: (String,String,Int,Boolean) -> Persona = ::Persona;
```

A COSA SERVONO?

- Method reference a costruttori sono utili quando occorre passare a un'altra funzione il *«modo di costruire qualcosa»*
  - una factory «parametrica» che costruisce «cose» secondo un suo schema, dove però «cosa» costruire è un parametro
  - scenario tipico in Java: *negli stream di operazioni* Java
  - ESEMPIO: il metodo factory **`Stream.toArray`** costruisce un array, ma ha bisogno di sapere esattamente *di che tipo*
    → riceve la reference al costruttore del tipo di array richiesto
    → per un array di interi, **`Integer[]::new`**
    → per un array di stringhe, **`String[]::new`**
    → per un array di Persona, **`Persona[]::new`**

Iterazione interna

ITERAZIONE INTERNA

- Le *lambda expression* non servono solo nei comparatori: un altro caso di grande rilevanza è l'*iterazione interna*
- È un cambio di paradigma: cambia *il modo stesso di concepire un ciclo* su una collection
  - anziché una struttura di controllo (**`for`**, **`while`**) che *comanda come iterare* su una struttura dati, *si chiede alla struttura* di ripetere su se stessa una operazione *ricevuta come lambda*
- A tal fine, le collection (e talvolta anche gli array) definiscono un metodo *foreach* **(*`operation`*)**
  - Java: **`forEach`** (non su array)
  - C#: **`ForEach`** (anche su array, come static method di `Array`)
  - Scala: **`foreach`** (anche su array)
  - Kotlin: **`forEach`** (anche su array)

Java

C#

Scala

Kotlin

ITERAZIONE INTERNA

- L'iterazione interna
  - astrae dal controllo: non dobbiamo più preoccuparci di governare noi l'iterazione in prima persona, "saprà la collection come fare"
  - astraendo dal controllo, rende possibile cambiare l'algoritmo di iterazione interno in modo trasparente all'esterno
  - potenzialmente, potrebbe anche agire in parallelo….
- Filosoficamente, abitua a uno stile più funzionale
  - uscire dall' "ossessione del controllo": non focalizzarsi in primis sul *come* fare le cose, ma su *cosa* si desidera ottenere
  - stile di specifica *dichiarativo* anziché imperativo
  - maggiore espressività & leggibilità, maggiori gradi di libertà futuri

ESEMPIO 1

OBIETTIVO: stampare tutte le persone di una collezione

Java

Approccio classico: iterazione *esterna*

```
for(Persona p : persone)
        System.out.println(p);
```

```
John Doe ha 25 anni
Jane Doe ha 45 anni
Anne Bee ha 31 anni
Jane Doe ha 22 anni
```

Nuovo approccio: iterazione *interna*

```
persone.forEach( p -> System.out.println(p) );
```

o, meglio, con method reference:

```
persone.forEach( System.out::println );
```

La lambda *operation* è un ***Consumer*** che consuma gli elementi uno ad uno.

- NB: in Java, `persone` dev'essere una `Collection`, *non un array*
  Per iterare in modo interno su array occorre quindi prima convertirli in lista:
  `Arrays.asList(arrayPersone).forEach(System.out::println);`

ESEMPIO 2

OBIETTIVO: calcolare l'età media di un elenco (array) di persone

Java

In teoria, verrebbe da scrivere così

```
int ageSum = 0;
Arrays.asList(persone).forEach(p -> ageSum += p.getEta());
System.out.println("età media = " + ageSum/persone.length);
```

MA *in Java non è possibile modificare variabili all'interno di una lambda*
(le variabili di «chiusura» devono essere **`final`** o *effectively final*)

La soluzione è incapsulare la variabile in un qualche wrapper

- non può essere **`Integer`**, perché non permette modifiche al valore incapsulato
- conviene definire al volo una *classe anonima* estendendo **`Object`**:

```
var wrapper = new Object(){ int ageSum = 0; };
Arrays.asList(persone).forEach(p -> wrapper.ageSum += p.getEta());
System.out.println("media = " + wrapper.ageSum/persone.length);
```

Così, la variabile (riferimento) in sé non viene modificata, ma tramite essa si registra comunque la variazione richiesta.

GLI ESEMPI NEGLI ALTRI LINGUAGGI

In C#

C#

- per gli array esiste il metodo statico tipizzato **ForEach** nella libreria **Array**: riceve due argomenti: l'array su cui operare e la lambda corrispondente (una **Action**)

```
Array.ForEach<Persona>(persone, p => Console.WriteLine(p));
```

- a differenza di Java, è possibile anche modificare le variabili nella lambda

```
int ageSum = 0;
Array.ForEach<Persona>(persone, p => ageSum += p.GetEta());
Console.WriteLine("media = " + ageSum/persone.Length);
```

- è inoltre disponibile il classico metodo **ForEach** (a un argomento) per le collection

```
List<Persona> lista = persone.ToList();
lista.ForEach(p => Console.WriteLine(p));
```

- In più, C# definisce alcuni metodi *fluent* per elaborare la lista, come il filtro **FindAll**: questo esempio stampa solo le persone di età minore di 40 anni:

```
lista.FindAll(p => p.GetEta()<40)
     .ForEach(p => Console.WriteLine(p));
```

In Java lo faremo con gli *stream di operazioni*

GLI ESEMPI NEGLI ALTRI LINGUAGGI

In Scala

Scala

- gli array sono assimilati a collection, quindi è applicabile direttamente anche su di essi il classico metodo **`ForEach`**:

```
persone.foreach(println(_)); // con parametro implicito
```

- come in C# e a differenza di Java, è possibile modificare le variabili nella lambda

```
var ageSum = 0;
persone.foreach( p => ageSum += p.eta );
println("media = " + ageSum/persone.size);
```

- ovviamente, il metodo **`foreach`** opera anche sulle collection

```
val lista = persone.toList
lista.foreach(println);
```

- anche Scala definisce alcuni metodi *fluent* per elaborare la lista, come il filtro **`filter`**, qui utilizzato con la comoda notazione a parametri impliciti (placeholder):

```
lista.filter(_.eta<40)
     .foreach(println(_));
```

In Java lo faremo con gli *stream di operazioni*

GLI ESEMPI NEGLI ALTRI LINGUAGGI

In Kotlin

Kotlin

- come in Scala, anche qui gli array sono assimilati a collection:

```
persone.forEach( {println(it)} ); // con parametro implicito
```

- anche qui è possibile modificare le variabili nella lambda:

```
var ageSum = 0;
persone.forEach({ p -> ageSum += p.eta });
println("media = " + ageSum/persone.size);
```

- ovviamente, il metodo **`foreach`** opera anche sulle collection

```
val lista = persone.toList()
lista.foreach({println(it)});
```

- anche Kotlin definisce alcuni metodi *fluent* per elaborare la lista, come il filtro **`filter`**, qui utilizzato con la comoda notazione a parametri impliciti (placeholder):

```
lista.filter({it.eta<40})
     .foreach({println(it)});
```

In Java lo faremo con gli *stream di operazioni*

COMPOSIZIONE DI FUNZIONI

- Alcuni linguaggi consentono di *comporre due lambda* per definire una nuova funzione composta
  - in Java: **`andThen`** e **`compose`** (solo su tipi funzione omogenei)
  - in Scala: **`andThen`** e **`compose`** (solo su funzioni a 1+ argomenti)
- Più precisamente:
  - **`f.andThen(g)`** definisce la funzione «sequenza» **`g(f(…))`**
  - **`f.compose(g)`** definisce la funzione «composta» **`f(g(…))`**

Java

```
IntUnaryOperator mulBy3 = x -> x * 3;
IntUnaryOperator incBy1 = x -> x + 1;

var y = mulBy3.andThen(incBy1).applyAsInt(4); // dà (3x4)+1 = 13
var z = mulBy3.compose(incBy1).applyAsInt(4); // dà 3x(4+1) = 15
```

Scala

```
val mulBy3 : (Int) => Int = (x) => x*3;
val incBy1 : (Int) => Int = (x) => x+1;

val y = mulBy3.andThen(incBy1)(4) // dà (3x4)+1 = 13
val z = mulBy3.compose(incBy1)(4) // dà 3x(4+1) = 15
```

COMPOSIZIONE DI FUNZIONI ANALOGIE & DIFFERENZE

- In Java
  - **`andThen`** e **`compose`** possono essere usate per comporre solo tipi funzione omogenei (ad esempio, due **`IntUnaryOperator`**, due **`IntConsumer`**, etc.): non è ammesso comporre entità diverse, anche quando la composizione avrebbe senso
  - però, **`andThen`** è definito anche sui consumer, intendendo che entrambi «mangiano» la stesso valore, in sequenza
- In Scala
  - **`andThen`** e **`compose`** possono essere usate per comporre anche tipi funzione disomogenei, purché la composizione abbia senso
  - però, la composizione non è ammessa con i consumer (funzioni a codominio **`Unit`**), in quanto appunto «consumano» il valore, neppure nella forma **`andThen`** supportata invece da Java

ESEMPIO

- Fase 1: definizione e test di due consumer

Java

```
IntConsumer printIfEven = (x) -> {
              if(x%2==0) System.out.println("Even " + x);};
IntConsumer printIfOdd  = (x) -> {
              if(x%2!=0) System.out.println("Odd  " + x);};
List.of(22,21,13,42,-5,-6).forEach(
          (Integer x) -> printIfEven.accept(x.intValue()) );
List.of(22,21,13,42,-5,-6).forEach(
          (Integer x) -> printIfOdd.accept(x.intValue())  );
```

Scala

```
val printIfEven : (Int) => Unit =
              (x) => if(x%2==0) println("Even " + x);
val printIfOdd  : (Int) => Unit =
              (x) => if(x%2!=0) println("Odd  " + x);
List(22,21,13,42,-5,-6).foreach(printIfEven(_))
List(22,21,13,42,-5,-6).foreach(printIfOdd(_))
```

ESEMPIO

- Fase 1: definizione e test di due consumer

Java

```
IntConsumer printIfEven = (x) -> {
                if(x%2==0) System.out.println("Even " + x);};
IntConsumer printIfOdd  = (x) -> {
                if(x%2!=0) System.out.println("Odd  " + x);};
List.of(22,21,13,42,-5,-6).forEach(printIfEven::accept);
List.of(22,21,13,42,-5,-6).forEach(printIfOdd::accept);
```

Method reference a oggetti funzione: BELLO! ☺☺
MA serve sapere il nome del metodo interno…☹☹

Scala

```
val printIfEven : (Int) => Unit =
                (x) => if(x%2==0) println("Even " + x);
val printIfOdd  : (Int) => Unit =
                (x) => if(x%2!=0) println("Odd  " + x);
List(22,21,13,42,-5,-6).foreach(printIfEven(_))
List(22,21,13,42,-5,-6).foreach(printIfOdd(_))
```

ESEMPIO

- Fase 2: composizione fra consumer
  - in Java, possibile **`andThen`** fra consumer (non ammessa in Scala)
  - ovviamente, invertendo l'ordine qui non cambia niente..

Java

```
printIfOdd.andThen(printIfEven).accept(4); // Even 4
printIfOdd.andThen(printIfEven).accept(5); // Odd 5
printIfEven.andThen(printIfOdd).accept(7); // Odd 7
printIfEven.andThen(printIfOdd).accept(6); // Even 6
```

- Fase 3: composizione fra entità diverse, purché sensata
  - in Scala, possibile mixare consumer e funzioni sia con **`andThen`**…
  - ovviamente, qui l'ordine è fondamentale: non si può fare viceversa

Scala

```
mulBy3.andThen(printIfEven)(4) // 12
mulBy3.andThen(printIfEven)(5) // non stampa nulla
mulBy3.andThen(printIfOdd)(7)  // 21
mulBy3.andThen(printIfOdd)(6)  // non stampa nulla
```

ESEMPIO

- Fase 4: composizione fra entità diverse, purché sensata
  - in Scala, possibile mixare consumer e funzioni anche con **`compose`**
  - ovviamente, qui l'ordine è invertito rispetto al caso **`andThen`**!

```scala
printIfEven.compose(incBy1)(4) // non stampa nulla
printIfEven.compose(incBy1)(5) // 6
printIfOdd.compose(incBy1)(7)  // non stampa nulla
printIfOdd.compose(incBy1)(6)  // 7
```

Scala

- Fase 3: composizione fra entità diverse, purché sensata
  - in Scala, possibile mixare consumer e funzioni sia con **`andThen`**…
  - ovviamente, qui l'ordine è fondamentale: non si può fare viceversa

```scala
mulBy3.andThen(printIfEven)(4) // 12
mulBy3.andThen(printIfEven)(5) // non stampa nulla
mulBy3.andThen(printIfOdd)(7)  // 21
mulBy3.andThen(printIfOdd)(6)  // non stampa nulla
```

Scala

ITERAZIONI COMPOSTE

- Le funzioni composte sono particolarmente comode in accoppiata con l'iterazione interna, *foreach*
  - quando una singola lambda come *operation* non basta più, se ne può creare una composta tramite **`andThen`** e/o **`compose`**
  - con l'unico limite di *non portare agli estremi la type inference*
- Esempio: ancora Persone
  - immaginiamo di ampliare la classe **`Persona`** aggiungendo due semplici consumer che filtrino e stampino solo:
    - una persona «giovane» (under 25) → **`printIfYoung`**
    - una persona «matura» (over 40) → **`printIfMature`**
  - OBIETTIVO: stampare le persone *tranne* quelle fra 25 e 40 anni
  - IDEA: *comporre opportunamente,* tramite **`andThen`**, i due consumer

ITERAZIONI COMPOSTE

- Esempio (segue)
  - OBIETTIVO: stampare le persone *tranne* quelle fra 25 e 40 anni
  - IDEA: comporre **`printIfYoung`** e **`printIfMature`**
- Prima possibilità
  - definiamo due alias di tipo **`Consumer<Persona>`**

```
Consumer<Persona> printIfOver40 = Persona::printIfMature;
Consumer<Persona> printIfUnder25 = Persona::printIfYoung;

elencoPersone.forEach(
    printIfOver40.andThen(printIfUnder25));
```

stampa completa

```
John Doe ha 25 anni
Jane Doe ha 45 anni
Anne Bee ha 31 anni
Jane Doe ha 22 anni
```

stampa composta

```
Jane Doe ha 45 anni
Jane Doe ha 22 anni
```

(non stampa la 31enne)
(non stampa il 25enne)

Java

ITERAZIONI COMPOSTE

- Perché questi alias?
  - perché l'uso diretto di **`printIfYoung`** e **`printIfMature`** mette-rebbe in crisi la type inference, causando errore di compilazione

```
elencoPersone.forEach( // NON COMPILA
  Persona::printIfMature.andThen(Persona::printIfYoung));
```
Java

  - MOTIVO: una method reference da sola non può fare da target per una chiamata di metodo, perché non dà sufficienti informazioni: è un problema di *target type non sufficientemente specificato*

TARGET TYPE

- Il compilatore deve poter dedurre *con certezza* il tipo della lambda (*target type*) e per farlo ha bisogno di informazioni adeguate *su ogni singolo passaggio*
  - non sempre le informazioni che ha bastano allo scopo
  - in particolare, servono *tipo degli argomenti* e *contesto d'uso*
- Le catene di comparatori (con **`thenComparing`**) e il concatenamento di funzioni (con **`andThen`**) *spingono al limite* le capacità inferenziali del compilatore
  - MOTIVO: spesso si conosce il tipo atteso del *risultato nel suo complesso*, ma *non dei singoli pezzi* (se sono più d'uno)
  - ergo, spesso una singola inferenza riesce, una catena no
  - le method reference aggravano il problema, in quanto forniscono solo il *nome* del metodo ma *non gli argomenti*

TARGET TYPE

- Il problema
  - se le informazioni in mano al compilatore non bastano per determinare la signature *di ogni singolo pezzo*, la type inference fallisce
  - per risolvere, bisogna dare al compilatore tali informazioni *per qualche altra via*
- Possibili cure
  - inserire un cast al *tipo di method reference* specifica, così da dargli «un aiutino» al punto giusto (Java only)
  - spostare la **`andThen`** in una funzione ausiliaria, i cui argomenti *tipizzati* diano al compilatore l'informazione che manca
  - incapsulare la *prima* method reference in una *funzione-identità* (es. **`itself`**) il cui argomento, di nuovo, abbia il solo scopo di *completare* l'informazione di tipo.

ITERAZIONI COMPOSTE
REFACTORING per target type

- Versione che non compila:

```Java
elencoPersone.forEach(
  Persona::printIfMature.andThen(Persona::printIfYoung));
```

- Chiamata riformulata tramite cast

```Java
elencoPersone.forEach(
    ((Consumer<Persona>)Persona::printIfMature)
                    .andThen(Persona::printIfYoung) );
```

- **andThen** spostata in una funzione ausiliaria (es. **combina**)

```Java
elencoPersone.forEach(
 combina(Persona::printIfMature, Persona::printIfYoung));
```

- Prima method reference incapsulata in una funzione-identità

```Java
elencoPersone.forEach(
      itself(Persona::printIfMature)
            .andThen(Persona::printIfYoung));
```

ITERAZIONI COMPOSTE
REFACTORING per target type

- Versione che non compila:

```
elencoPersone.forEach(
  Persona::printIfMature.andThen(Persona::printIfYoung));
```
Java

- Obiettivo: ...

```
private static <T> Consumer<T>
 combina(Consumer<T> f1, Consumer<T> f2){
    return f1.andThen(f2);
}
```

```
...ature)
...IfYoung) );
```
Java

- **andThe...** ...ata in una funzione ausiliaria (es. **combina**)

```
elenco...rsone.forEach(
 combina(Persona::printIfMatu...
```

```
private static <T> Consumer<T>
 itself(Consumer<T> f){
    return f;
 }
```

- Prima method reference incapsu...

```
elencoPersone.forEach(
      itself(Persona::printIfMature)
            .andThen(Persona::printIfYoung));
```
Java

ITERAZIONI COMPOSTE in SCALA

- In Scala
  - non esistendo method reference, si utilizzano in loro vece i parametri impliciti, scrivendo:

    `_.printIfYoung()` al posto di **`Persona::printIfYoung`**

    `_.printIfMature()` al posto di **`Persona::printIfMature`**

  - non potendo combinare consumer, occorre cambiare il tipo delle due funzioni **`printIfYoung`** e **`printIfMature`**

    → adottiamo il pattern *cascading*, restituendo *la persona stessa*

```
case class Persona(val nome:String, val cognome:String, val eta:Int){
  def printIfMature() : Persona = { if(eta>40) println(this); this; }
  def printIfYoung()  : Persona = { if(eta<25) println(this); this; }
}
```

ITERAZIONI COMPOSTE in SCALA

- In Scala
  - l'esempio assume allora la seguente forma:

Scala

```
val printIfOver40  = (p:Persona) => p.printIfMature();
// OR: val printIfOver40 : (Persona) => Persona = _.printIfMature();
val printIfUnder25 = (p:Persona) => p.printIfYoung();
// OR: val printIfUnder25 : (Persona) => Persona = _.printIfYoung();

persone.foreach(printIfOver40.andThen(printIfUnder25)); // OK
```

  - come prevedibile, la seguente forma NON compila:

```
persone.foreach((_.printIfMature()).andThen(_.printIfYoung()));
// NON COMPILA: missing parameter for expanded function
```

  - CURA: il cast in Scala ha la forma **`asInstanceOf[T]`**, che però non è applicabile al placeholder **_**
    Quindi, restano solo le altre due opzioni.

ITERAZIONI COMPOSTE in SCALA
REFACTORING per target type

- Versione che non compila:

```
persone.foreach(                                                    Scala
     (_.printIfMature()).andThen(_.printIfYoung()));
```

- **andThen** spostata in una funzione ausiliaria (es. **combina**)

```
persone.foreach(                                                    Scala
     combina(_.printIfMature() , _.printIfYoung()));
def combina(a: (Persona)=>Persona, b: (Persona)=>Persona):
     (Persona)=>Persona = a.andThen(b);
```

- Prima method reference incapsulata in una funzione-identità

```
persone.foreach(                                                    Scala
     itself(_.printIfMature()).andThen(_.printIfYoung()));
def itself(p: (Persona)=>Persona) :
     (Persona)=>Persona = p;
```