UNIVERSITÀ DEGLI STUDI DI BRESCIA · UNIBS.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

LO2

Dal nome alla funzione

Classi anonime, lambda, method reference · le interfacce, a fondo



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **Il problema**: classi usa-e-getta, nel motore della calcolatrice
2. **La scala**: classe → classe anonima → lambda → method reference
3. **Perché**: iniettare un comportamento in un metodo
4. `java.util.function`: le interfacce già pronte
5. **Le interfacce, a fondo**: `default`, `static`, `private`, `sealed`

01

Classi usa-e-getta



La calcolatrice ha bisogno di operatori

```java
public interface BinaryOperator {
    int eval(int a, int b);
}
```

```java
public class Somma implements BinaryOperator {
    public int eval(int a, int b) { return a + b; }
}
```

```java
Map<String, BinaryOperator> operatori = new HashMap<>();
operatori.put("+", new Somma());
operatori.put("-", new Sottrazione());
operatori.put("^", new Potenza());

BinaryOperator operator = operatori.get(op);
```

La classe anonima

CLASSE CON NOME

```
public class Somma implements BinaryOperator {
    public int eval(int a, int b) {
        return a + b;
    }
}

BinaryOperator somma = new Somma();
```

Dichiarata in un file, istanziata altrove.

CLASSE ANONIMA

```
BinaryOperator somma = new BinaryOperator() {
    public int eval(int a, int b) {
        return a + b;
    }
};
```

Dichiarata **nel punto in cui serve**, senza nome.

Anatomia di una classe anonima

```
BinaryOperator somma = new BinaryOperator() {
    public int eval(int a, int b) {
        return a + b;
    }
};
```

- `new BinaryOperator()`: il tipo che si implementa (o si estende) e gli argomenti per il costruttore del padre
- il blocco `{ ... }`: il **corpo della classe**, con i metodi da implementare
- il `;` finale: è un'**espressione**, produce un oggetto
- un nome ce l'ha comunque: il compilatore genera `Calc$1.class` (lo rivedremo con la reflection)

Che cosa può, che cosa non può

Può	Non può
usare campi e metodi della classe che la contiene	dichiarare un costruttore: non ha nome
dichiarare campi e metodi propri	essere istanziata una seconda volta
avere blocchi di inizializzazione `{ ... }`	implementare più di un'interfaccia
catturare le variabili locali del contesto	modificare le variabili catturate

> Le variabili catturate devono essere **effectively final**: assegnate una volta sola, mai più riassegnate. Il compilatore lo verifica.

Closure: il codice porta con sé il contesto

```
static BinaryOperator sommaScalata(int k) {
    return new BinaryOperator() {
        public int eval(int a, int b) {
            return k * (a + b);           // k viene catturata
        }
    };
}
```

```
BinaryOperator doppia = sommaScalata(2);
doppia.eval(3, 4);      // 14: 2 è stato catturato nel codice della classe anonima
```

- **closure**: blocco di codice + parametri + variabili libere del contesto
- `k` viene copiata nell'oggetto: per questo deve essere *effectively final*

02

La scala



Lambda: quando il metodo è uno solo

Se l'interfaccia ha **un solo metodo astratto**, il compilatore sa già quale stiamo implementando. Restano parametri e corpo:

```
BinaryOperator somma = new BinaryOperator() {
    public int eval(int a, int b) {
        return a + b;
    }
};
```

```
BinaryOperator somma = (a, b) -> { return a + b; };
```

```
BinaryOperator somma = (a, b) -> a + b;
```

`(parametri) -> espressione` oppure `(parametri) -> { istruzioni }`. I tipi dei parametri li **deduce** il compilatore dall'interfaccia.

La sintassi, tutta

```
() -> 42                                  // nessun parametro
x -> x * 2                                // un parametro: parentesi facoltative
(x, y) -> x + y                           // due parametri
(int x, int y) -> x + y                   // tipi espliciti, se servono
x -> { log(x); return x * 2; }            // blocco: return esplicito
```

> Una lambda non ha un tipo proprio: lo prende dal contesto (**target typing**). La stessa `(a, b) -> a + b` vale come la nostra `BinaryOperator`, come `IntBinaryOperator`, come `BinaryOperator<Integer>`.

Method reference: quando la lambda chiama solo un metodo

Forma	Esempio	La lambda equivalente
`oggetto::metodo`	`System.out::println`	`x -> System.out.println(x)`
`Classe::metodoStatico`	`Math::sqrt`	`x -> Math.sqrt(x)`
`Classe::metodoDiIstanza`	`String::compareTo`	`(s, t) -> s.compareTo(t)`
`Classe::new`	`ArrayList::new`	`() -> new ArrayList<>()`

Nella terza forma il **primo parametro** della lambda diventa l'oggetto su cui si chiama il metodo.

La scala completa

Lo stesso comportamento, quattro modi di scriverlo:

```
operatori.put("+", new Somma());                                    // 1. classe con nome

operatori.put("+", new BinaryOperator() {                           // 2. classe anonima
    public int eval(int a, int b) { return a + b; }
});

operatori.put("+", (a, b) -> a + b);                                // 3. lambda

operatori.put("+", Integer::sum);                                   // 4. method reference
```

Sono tutti oggetti che implementano `BinaryOperator`. Cambia solo quanto rumore c'è attorno al **comportamento**.

03

Perché



Due metodi quasi uguali

```java
static void interi(int n) {
    for (int i = 0; i < n; i++)
        System.out.println(i);
}

static void pari(int n) {
    for (int i = 0; i < n; i++)
        if (i % 2 == 0) System.out.println("pari: " + i);
}
```

Il ciclo è lo stesso; cambia **che cosa si fa** a ogni passo. Vorremmo passare quel "che cosa" come parametro.

Iniettare il comportamento

```java
static void ripeti(int n, IntConsumer azione) {    // IntConsumer: void accept(int)
    for (int i = 0; i < n; i++)
        azione.accept(i);
}
```

```java
ripeti(5, i -> System.out.println(i));                                  // interi
ripeti(5, i -> { if (i % 2 == 0) System.out.println("pari: " + i); });  // pari
```

```java
static void ripeti(int n, Runnable azione) {     // quando l'indice non serve
    for (int i = 0; i < n; i++) azione.run();
}
ripeti(3, () -> System.out.println("Hello!"));                         // hello
ripeti(3, () -> System.out.print("*"));                                 // star
```

A che cosa serve, in generale

Una lambda dichiara un blocco di codice **qui**, per eseguirlo **altrove, dopo**, magari più volte:

- eseguire lo stesso codice in punti diversi (`ripeti`)
- iniettare logica in metodi di terzi (`list.sort(...)`, `map.computeIfAbsent(...)`)
- reagire a un evento (`button.addActionListener(e -> ...)`, tra qualche settimana)
- eseguire codice solo se serve (`log.debug(() -> messaggioCostoso())`)
- eseguire codice su un altro thread (`new Thread(() -> ...)`)

java.util.function

Interfaccia	Metodo	Firma	Esempio
`Supplier<T>`	`get`	`() → T`	`() -> new Random()`
`Consumer<T>`	`accept`	`T → void`	`s -> System.out.println(s)`
`Function<T,R>`	`apply`	`T → R`	`s -> s.length()`
`Predicate<T>`	`test`	`T → boolean`	`s -> s.isEmpty()`
`UnaryOperator<T>`	`apply`	`T → T`	`s -> s.toUpperCase()`
`BinaryOperator<T>`	`apply`	`(T, T) → T`	`(a, b) -> a + b`

> Per i primitivi, senza boxing: `IntConsumer`, `IntBinaryOperator` (la nostra `BinaryOperator`)…
> `Runnable` sta in `java.lang`.

`@FunctionalInterface`

```java
@FunctionalInterface
public interface BinaryOperator {
    int eval(int a, int b);
}
```

- non cambia il comportamento: **documenta** l'intenzione
- il compilatore rifiuta l'interfaccia se i metodi astratti diventano due
- vale per qualunque interfaccia con un solo metodo astratto, anche le vostre
- i metodi `default` e `static` **non contano**: `Comparator` ne ha decine e resta funzionale

04

Le interfacce, a fondo



Un’interfaccia, dieci anni dopo

Fino a Java 7 un’interfaccia era un elenco di metodi astratti. Oggi può contenere:

	Da	A che cosa serve
metodi astratti	sempre	il contratto
costanti `static final`	sempre	con parsimonia
metodi `default`	Java 8	comportamento condiviso; evolvere senza rompere
metodi `static`	Java 8	factory e utilità legate al tipo
metodi `private`	Java 9	codice comune ai `default`
`sealed` … `permits`	Java 17	l’elenco chiuso delle implementazioni

Metodi `default`: comportamento nell'interfaccia

```java
public interface Shape2d {
    double area();
    double perimeter();

    default String describe() {
        return getClass().getSimpleName() + " · area " + area();
    }
}
```

- `Rectangle`, `Circle` e `Diamond` ereditano `describe()` senza scrivere nulla
- una classe può comunque **sovrascriverlo**
- il motivo storico: aggiungere `forEach` a `Iterable` senza rompere le classi esistenti

Metodi `static` e `private` JAVA 9

```java
public interface Shape2d {
    double area();

    static Shape2d square(double side) {          // Shape2d.square(3)
        return new Rectangle(side, side);
    }
    default String describe() { return name() + " · area " + area(); }
    default String brief()    { return name(); }
    private String name()     { return getClass().getSimpleName(); }
}
```

- `static`: si chiama sul tipo, `Shape2d.square(3)`; non si eredita
- `private`: codice condiviso tra i `default`, invisibile alle implementazioni

Due `default` in conflitto

```
interface Drawable { default String describe() { return "disegnabile"; } }
interface Shape2d  { default String describe() { return "forma"; } }

class Circle implements Shape2d, Drawable {
    public String describe() {                                  // obbligatorio
        return Shape2d.super.describe() + ", " + Drawable.super.describe();
    }
}
```

- il compilatore **rifiuta** `Circle` finché non sceglie
- `Interfaccia.super.metodo()` richiama la versione voluta
- un metodo ereditato da una **classe** padre vince sempre su un `default`

`Comparator`: funzionale e ricca insieme

```
shapes.sort((a, b) -> Double.compare(a.area(), b.area()));     // compare
```

```
shapes.sort(Comparator.comparing(Shape2d::area));                    // static
shapes.sort(Comparator.comparing(Shape2d::area).reversed());         // default
shapes.sort(Comparator.comparing(Shape2d::perimeter)
                      .thenComparing(Shape2d::area));                // default
```

Un solo metodo astratto per la lambda; `static` e `default` per **comporre**. È lo stile delle API moderne, e quello che useremo con gli stream.

Un mondo chiuso: `sealed` JAVA 17

Se le forme del bimondo sono un insieme fissato lo possiamo dichiarare al compilatore:

```
public sealed interface Shape2d
        permits Rectangle, Circle, Diamond {
    double area();
}
public final class Rectangle implements Shape2d { ... }
public final class Circle    implements Shape2d { ... }
public final class Diamond   implements Shape2d { ... }
```

- `permits` elenca **tutte** le implementazioni ammesse; ogni altra è rifiutata
- ciascuna deve dichiararsi `final`, `sealed` o `non-sealed`
- è utile se vogliamo limitare l'insieme di classi che implementano un'interfaccia e creare un sismtema di controllo automatico negli switch-case.

`switch` esaustivo sui tipi JAVA 21

```
static String describe(Shape2d s) {
    return switch (s) {
        case Rectangle r -> "rettangolo " + r.width() + "×" + r.height();
        case Circle c    -> "cerchio di raggio " + c.radius();
        case Diamond d   -> "rombo " + d.d1() + "×" + d.d2();
    };   // senza default: una forma in più = errore di compilazione
}
```

- lo `switch` sui tipi sostituisce la catena di `instanceof`
- l'esaustività è verificata dal compilatore, perché il mondo è chiuso
- un `default` resta possibile, ma rinuncia proprio a questa verifica

Aperto o chiuso?

	Gerarchia aperta	`sealed`
chi aggiunge un tipo	chiunque, ovunque	solo chi possiede l'interfaccia
aggiungere un'operazione	un metodo in ogni classe	un `case` in ogni `switch`, segnalato dal compilatore
quando	plugin, librerie, estensioni di terzi	domini fissati: forme, comandi, token, messaggi

> Un `enum` è il caso limite: insieme chiuso di **istanze**, non di tipi. `sealed` serve quando i casi portano dati diversi.