

# Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

## Capitolo 8: Interfacce e la classe `Object`

Autore: Tommaso C.

---

Appunti tratti dalle lezioni in aula e dal materiale del corso

---

Argomenti: Interfacce · Interfacce come tipo · Interfacce e classi astratte  
La classe `Object` · La classe `Class`

## Indice

|   |                              |    |
|---|------------------------------|----|
| 1 | Interfacce                   | 1  |
| 2 | Interfacce come tipo         | 2  |
| 3 | Interfacce e classi astratte | 4  |
| 4 | La classe <b>Object</b>      | 5  |
| 5 | La classe <b>Class</b>       | 7  |
| 6 | Esercizi                     | 10 |
| 7 | Soluzioni                    | 11 |

## 1 Interfacce

### Definizione – Interfaccia

Un'**interfaccia**, dichiarata **interface** al posto di **class**, è un elenco di metodi pubblici di cui si dà la segnatura ma non il corpo. Una classe **implementa** l'interfaccia, con **implements**, quando definisce tutti i suoi metodi.

L'interfaccia dice *che cosa* si può fare con un oggetto, non *come*. È un contratto: elenca dei servizi, e ogni classe che la implementa si impegna a fornirli tutti. Si può pensare come una classe astratta portata all'estremo:

- non ha campi, salvo costanti statiche;
- tutti i metodi sono astratti, e implicitamente **public**.

### Nota – Metodi con il corpo

Da Java 8 un'interfaccia può avere anche metodi **static** e metodi **default**, con il corpo, che le classi ereditano se non li ridefiniscono. In questo capitolo le interfacce hanno solo metodi astratti.

```
public interface Confrontabile {  
    boolean maggiore(Confrontabile c);    // implicitamente public  
    boolean paritetico(Confrontabile c);  
}
```

`c1.maggiore(c2)` dice se `c1` è maggiore di `c2`, e `c1.paritetico(c2)` se i due sono equivalenti. Il parametro è di tipo **Confrontabile**, quindi un oggetto confrontabile si confronta con un altro oggetto confrontabile.

```
public class Edificio implements Confrontabile {
    protected int altezza;
    public Edificio(int a) { altezza = a; }
    public int getAltezza() { return altezza; }
    @Override
    public boolean maggiore(Confrontabile e) {
        if (e != null && e instanceof Edificio)
            return altezza > ((Edificio) e).altezza;
        else return false;
    }
    @Override
    public boolean paritetico(Confrontabile e) {
        if (e != null && e instanceof Edificio)
            return altezza == ((Edificio) e).altezza;
        else return false;
    }
}
```

I metodi ricevono un **Confrontabile**, perché la segnatura deve essere quella dell'interfaccia. Per leggere l'altezza dell'altro edificio serve quindi il cast **(Edificio)** e (Capitolo 7, Sezione 2), e **e instanceof Edificio** controlla prima che l'oggetto sia davvero un edificio (Sezione 5). **Persona** ha la stessa struttura, con il campo **eta** al posto di **altezza** e il getter **getEta()**.

#### Proposizione – Implementare un'interfaccia

- La classe deve definire tutti i metodi dell'interfaccia, con la stessa segnatura e **public**. Altrimenti è un errore di compilazione, salvo che la classe sia **abstract**.
- Un'interfaccia non ha istanze: **new Confrontabile()** non compila.

## 2 Interfacce come tipo

#### Proposizione – Un'interfaccia è un tipo

Un'interfaccia si usa come tipo di un parametro, di una variabile o di un array. Un riferimento di tipo **Confrontabile** denota un oggetto di una classe che implementa **Confrontabile**.

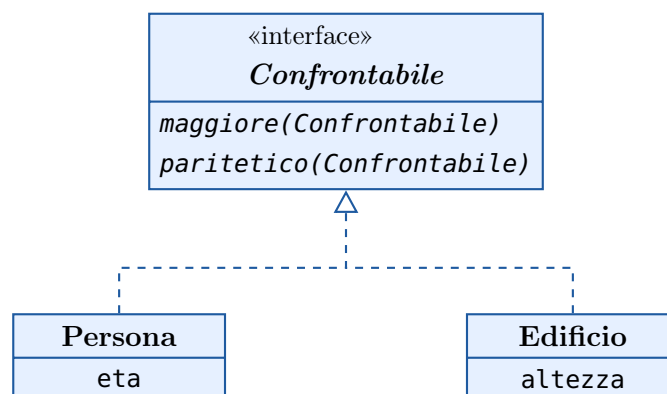
Così si scrivono funzioni polimorfe, come nel Capitolo 7, Sezione 9, che usano un'interfaccia senza sapere quali classi la implementano:

```
public class ServiziConfrontabile {  
    public static Confrontabile maggioreTraTre(Confrontabile c1,  
        Confrontabile c2, Confrontabile c3) {  
        if ((c1.maggiore(c2) || c1.paritetico(c2))  
            && (c1.maggiore(c3) || c1.paritetico(c3)))  
            return c1;  
        else if ((c2.maggiore(c1) || c2.paritetico(c1))  
            && (c2.maggiore(c3) || c2.paritetico(c3)))  
            return c2;  
        else return c3;  
    }  
}
```

```

Persona p1 = new Persona(30), p2 = new Persona(35), p3 = new Persona(32);
Edificio e1 = new Edificio(12), e2 = new Edificio(5), e3 = new Edificio(100);
Persona pp = (Persona) ServiziConfrontabile.maggioreTraTre(p1, p2, p3);
Edificio ee = (Edificio) ServiziConfrontabile.maggioreTraTre(e1, e2, e3);
System.out.println(pp.getEta());           // 35
System.out.println(ee.getAltezza());       // 100
    
```

La stessa funzione, scritta una volta, confronta persone ed edifici, e varrà per ogni classe che implementerà **Confrontabile**. Restituisce un **Confrontabile**, quindi chi vuole usare il risultato come persona deve fare il cast (**Persona**).



*Due classi senza nulla in comune implementano la stessa interfaccia. In UML l'implementazione è una freccia tratteggiata con la punta vuota.*

### 3 Interfacce e classi astratte

#### Proposizione – Che cosa è e che cosa sa fare

- Una **classe astratta** astrae un insieme di oggetti simili, quelli delle sue derivate: dice che cosa un oggetto è.
- Un'**interfaccia** astrae un insieme di funzionalità: dice che cosa un oggetto *sa fare*.

Rettangoli, triangoli e cerchi sono figure, con un colore e una struttura comuni, quindi **Figura** è una classe astratta (Capitolo 7, Sezione 8). Persone ed edifici hanno in comune solo la capacità di confrontarsi: una classe base comune direbbe che sono lo stesso tipo di cosa, ed è una forzatura. In più una classe ha una sola classe base, mentre le interfacce possono essere molte.

### Proposizione – Più interfacce

1. Una classe può implementare più interfacce, elencate dopo **implements** e separate da virgole.
2. Un'interfaccia può derivare da altre interfacce con **extends**, anche da più di una: per le interfacce l'ereditarietà multipla è ammessa.
3. Una classe che implementa un'interfaccia derivata definisce i metodi di tutte le interfacce da cui deriva.

```
public interface I { void g(); }
public interface J { void h(); }
public interface M extends I, J { void k(); }

public class C1 implements I, J { // definisce g() e h()
    public void g() { ... }
    public void h() { ... }
}
public class C2 implements M { // definisce g(), h() e k()
    public void g() { ... }
    public void h() { ... }
    public void k() { ... }
}
```

Se **J** estendesse **I**, una classe che implementa **J** definirebbe sia **h()** sia **g()**, ereditato da **I**. L'ereditarietà multipla, vietata fra le classi (Capitolo 7, Sezione 1), qui non crea ambiguità: due interfacce che dichiarano lo stesso metodo astratto non portano due corpi fra cui scegliere, e la classe ne scrive uno solo.

|                 | Classe astratta                   | Interfaccia                            |
|-----------------|-----------------------------------|----------------------------------------|
| Astrae          | che cosa un oggetto è             | che cosa un oggetto sa fare            |
| Campi           | sì                                | solo costanti statiche                 |
| Metodi          | astratti e con il corpo           | astratti, <b>public</b>                |
| Per ogni classe | una sola base, con <b>extends</b> | quante si vuole, con <b>implements</b> |

## 4 La classe **Object**

### Definizione – La classe **Object**

Tutte le classi Java derivano da **Object**: direttamente, se non hanno **extends**, oppure attraverso la loro classe base. Ogni oggetto è quindi un'istanza di **Object** e ne eredita i metodi.

**Object** è la radice di tutte le gerarchie di classi: `class B { ...}` è come `class B extends Object { ...}`, e una variabile di tipo **Object** può denotare qualunque oggetto.

| Metodo di <b>Object</b>                      | Che cosa fa                                   |
|----------------------------------------------|-----------------------------------------------|
| <code>public String toString()</code>        | una stringa che rappresenta l'oggetto         |
| <code>public final Class getClass()</code>   | la classe dell'oggetto (Sezione 5)            |
| <code>public boolean equals(Object o)</code> | l'uguaglianza con <code>o</code> (Capitolo 9) |
| <code>public int hashCode()</code>           | un intero calcolato dall'oggetto (Capitolo 9) |
| <code>protected Object clone()</code>        | una copia dell'oggetto (Capitolo 9)           |

### Definizione – `toString`

`toString()` restituisce una stringa stampabile che rappresenta l'oggetto. Java la invoca da solo quando l'oggetto va stampato con `println` o concatenato a una stringa con `+`.

```
public class B {
    private int i;
    public B(int x) { i = x; }
    @Override
    public String toString() { return "i: " + i; }
}
...
B b = new B(5);
System.out.println(b);    // i: 5
```

La versione di **Object** dà il nome della classe, `@` e un codice esadecimale, come `B@58644d46`. Il codice cambia da un'esecuzione all'altra e non dice nulla del contenuto, quindi conviene ridefinire `toString`.

In una derivata, `super.toString()` riusa la descrizione della base:

```
public class D extends B {
    private int j;
    public D(int x, int y) {
        super(x);
        j = y;
    }
    @Override
    public String toString() { return super.toString() + " j: " + j; }
}
...
System.out.println(new D(5, 10));    // i: 5 j: 10
```

`i` è `private` in `B`, quindi `D` non lo potrebbe nemmeno leggere. Con `super.toString()` ne ottiene la descrizione senza conoscerla e senza duplicare codice, e se `B` aggiunge un campo e aggiorna il suo `toString`, `D` non va cambiata.

## 5 La classe `Class`

### Definizione – La classe `Class`

Durante l'esecuzione, per ogni classe e interfaccia del programma la JVM ha un oggetto di tipo `Class` che la rappresenta. Lo denotano:

- il letterale `C.class`, per la classe `C`;
- un riferimento di tipo `Class`, come `Class c = C.class`; oppure `Class c = o.getClass()`;

L'oggetto `Class` conosce nome, metodi e classe base della classe che rappresenta, per esempio `d.getClass().getName()` dà `"D"`. Gli oggetti `Class` li crea la JVM quando carica le classi: `Class` non ha costruttori accessibili, e `new Class()` non compila.

### Definizione – `isInstance`

`c.isInstance(o)` restituisce `true` se e solo se `o` denota un oggetto istanza della classe, o dell'interfaccia, rappresentata da `c`.

«Istanza» va inteso secondo i principi della derivazione (Capitolo 7, Sezione 1): un oggetto **D** è anche un **B**.

```
public class B {}
public class D extends B {}
...
B b1 = new B();
D d1 = new D();
System.out.println(B.class.isInstance(b1)); // true
System.out.println(B.class.isInstance(d1)); // true: un D è un B
System.out.println(D.class.isInstance(d1)); // true
System.out.println(D.class.isInstance(b1)); // false: un B non è un D
```

Il controllo riguarda l'oggetto, non il riferimento, cioè guarda il tipo dinamico (Esercizio 1).

#### Definizione – instanceof

L'operatore **o instanceof C** verifica se **o** è istanza della classe **C**, o implementa l'interfaccia **C**. Equivale a **C.class.isInstance(o)**.

**instanceof** non usa un oggetto **Class** ma il nome della classe scritto nel codice, che quindi deve essere noto durante la compilazione. Con **isInstance** la classe si può scegliere durante l'esecuzione: se **c** è una variabile di tipo **Class**, **c.isInstance(o)** non si può scrivere con **instanceof**. Se **o** vale **null**, **o instanceof C** è **false**: in Edificio il controllo **e != null** è ridondante.

#### Definizione – getClass

**o.getClass()**, ereditato da **Object**, restituisce l'oggetto **Class** della classe con cui **o** è stato creato, la più specifica, qualunque sia il tipo del riferimento usato.

`instanceof` accetta anche le derivate, `getClass` dà la classe esatta:

```
B b = new D();
System.out.println(b instanceof B);           // true: un D è un B
System.out.println(b.getClass() == D.class); // true
System.out.println(b.getClass() == B.class); // false
```

`getClass` serve a confrontare la classe di due oggetti, come fa `equals` nel Capitolo 9.

### Riepilogo

- **Interfaccia:** metodi `public` senza corpo, e nessuna istanza. Chi la implementa definisce tutti i metodi.
- **Interfaccia come tipo:** per parametri, variabili e array. Le funzioni che la usano valgono per ogni classe che la implementa.
- **Classe astratta e interfaccia:** che cosa un oggetto è, e che cosa sa fare.
- **Più interfacce:** `implements I, J`. `interface M extends I, J` è ereditarietà multipla.
- **Object:** la radice di ogni gerarchia. Tutte le classi ne ereditano i metodi.
- **toString:** si ridefinisce per descrivere l'oggetto, e nelle derivate riusa `super.toString()`.
- **Class:** `C.class` e `o.getClass()`. `isInstance` e `instanceof` accettano le derivate.
- **getClass:** la classe esatta dell'oggetto, qualunque sia il riferimento.

## 6 Esercizi

### Esercizio 1 – `isInstance` con un'interfaccia

Che cosa stampa ciascuna istruzione, e perché?

```
public interface I {}  
public class D implements I {}  
  
...  
I i1 = new D();  
D d1 = new D();  
System.out.println(I.class.isInstance(i1));  
System.out.println(I.class.isInstance(d1));  
System.out.println(D.class.isInstance(d1));  
System.out.println(D.class.isInstance(i1));
```

*Soluzione a pagina 11 →*

## 7 Soluzioni

### Soluzione dell'Esercizio 1 – `isInstance` con un'interfaccia

Tutte e quattro stampano `true`, perché `i1` e `d1` denotano due oggetti `D`, e `D` implementa `I`:

1. ogni oggetto `D` è istanza di `I`, quindi le prime due sono vere;
2. ogni oggetto `D` è istanza di `D`, quindi lo sono anche le ultime due.

L'ultima conta: `i1` ha tipo statico `I`, ma `isInstance` guarda l'oggetto, che è un `D`. Nell'esempio con `B` e `D` della Sezione 5 la chiamata analoga era falsa perché l'oggetto era davvero un `B`.

[\*← Torna all'Esercizio 1, pagina 10\*](#)