Polimorfismo in Java

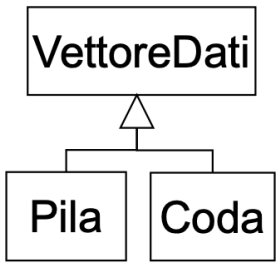
Prima di introdurre tutta la discussione dietro il polimorfismo nel linguaggio Java, è necessario tenere sempre a mente il **principio di sostituzione di Liskov**, formulazione logica che fonda le basi della pratica del polimorfismo in programmazione: "*se X è un sottotipo di T allora variabili di tipo T in un programma possono essere sostituite da variabili di tipo X senza alterare alcuna proprietà desiderabile del programma"*.

I concetti di ereditarietà e polimorfismo, in OOP, sono **fondamentali**, in quanto permettono di scegliere di volta in volta la "vista" con cui trattare un oggetto (specifica o più generale) permettendo sempre di mantenere una classe "aperta" ossia disponibile per eventuali estensioni o modifiche.

Processo decisionale

Prendiamo come esempio il seguente pezzo di codice:

```
public static void main(String a[]){
    VettoreDati p;
    // leggi k
    if (k==1) p = new Pila();
    else p = new Coda();
    p.inserisci(1);
    p.inserisci(2);
    p.estrai();
}
```

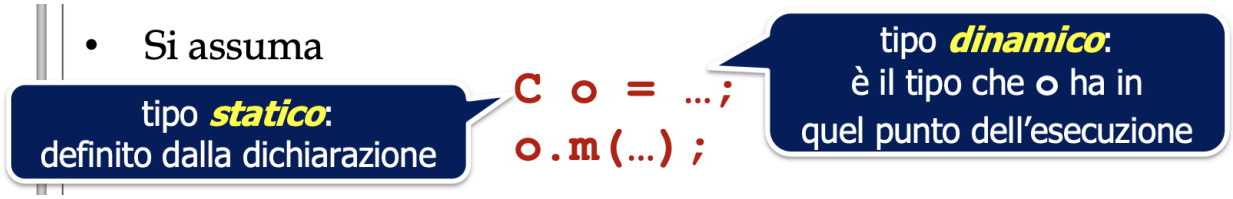


Si nota evidentemente che il tipo "**p**" è **deciso a runtime** e quindi possiamo affermare che il legame tra un oggetto e il suo tipo è un **legame dinamico**, da qui deriva il termine inglese **dynamic binding**.

Criteri e regole del polimorfismo

In presenza di polimorfismo viene operata una distinzione tra **tipo statico** (dichiarato a *compilation time*) **e tipo dinamico** (dichiarato a *runtime*) di una variabile o anche parametro formale. Come prima restrizione si ha che il tipo dinamico deve essere un **sottotipo** del tipo statico o tuttalpiù lo stesso. Inoltre è necessario mettere in evidenza che un'**invocazione** di un metodo su un oggetto **dipende** sia **dal tipo statico** di quell'oggetto che **dal tipo dinamico**, sostanzialmente il problema che ci si pone è **quale implementazione scegliere** se ci si trova di fronte una ridefinizione.

Il binding dinamico segue quindi le seguenti regole:



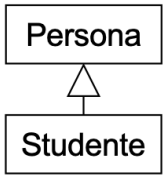
il metodo scelto **dipende dal tipo dinamico** e viene quindi DECISO A RUNTIME secondo determinati criteri:

1) Si cerca all'interno della **classe del tipo statico** [C] il metodo [m] con la **firma più simile all'invocazione**, tenendo conto del fatto che le firme da considerare sono quelle fissate a *compilation time* quindi guardando il tipo statico dei parametri.
2) Si verifica che il **tipo dinamico sia un sottotipo di quello statico**.
3) Si verifica se il **tipo dinamico ridefinisce** (Override, quindi tassativamente stessa firma del metodo analizzato) il metodo chiamato [m]: in caso **affermativo** si usa l'**implementazione del tipo dinamico**. In caso **negativo** si usa l'**implementazione del tipo statico.**

Quindi in poche parole il tipo statico determina quali metodi possono essere invocati, il tipo dinamico suggerisce quale ridefinizione invocare.
Per fare chiarezza si considerino i seguenti esempi:

|
|
|
V

```
Persona  p = new Persona("Ada");
Studente s = new Studente("Leo");
Persona ps = new Studente("Ugo");
```




```
chiSei()
chiSiete(Persona p)
```

```
chiSei()
chiSiete(Persona p)
chiSiete(Studente s)
```

```
p.chiSei();
s.chiSei();
```
tipo dinamico = tipo statico: non c'è scelta

```
ps.chiSei();
```
tipo dinamico ≠ tipo statico: uso la ridefinizione in **Studente**

```
p.chiSiete(p);
p.chiSiete(s);
p.chiSiete(ps);
```
tipo dinamico = tipo statico: non c'è scelta

```
s.chiSiete(p);
```
tipo dinamico = tipo statico: uso il metodo con input **Persona**

```
s.chiSiete(s);
```
tipo dinamico = tipo statico: uso il metodo con input **Studente**

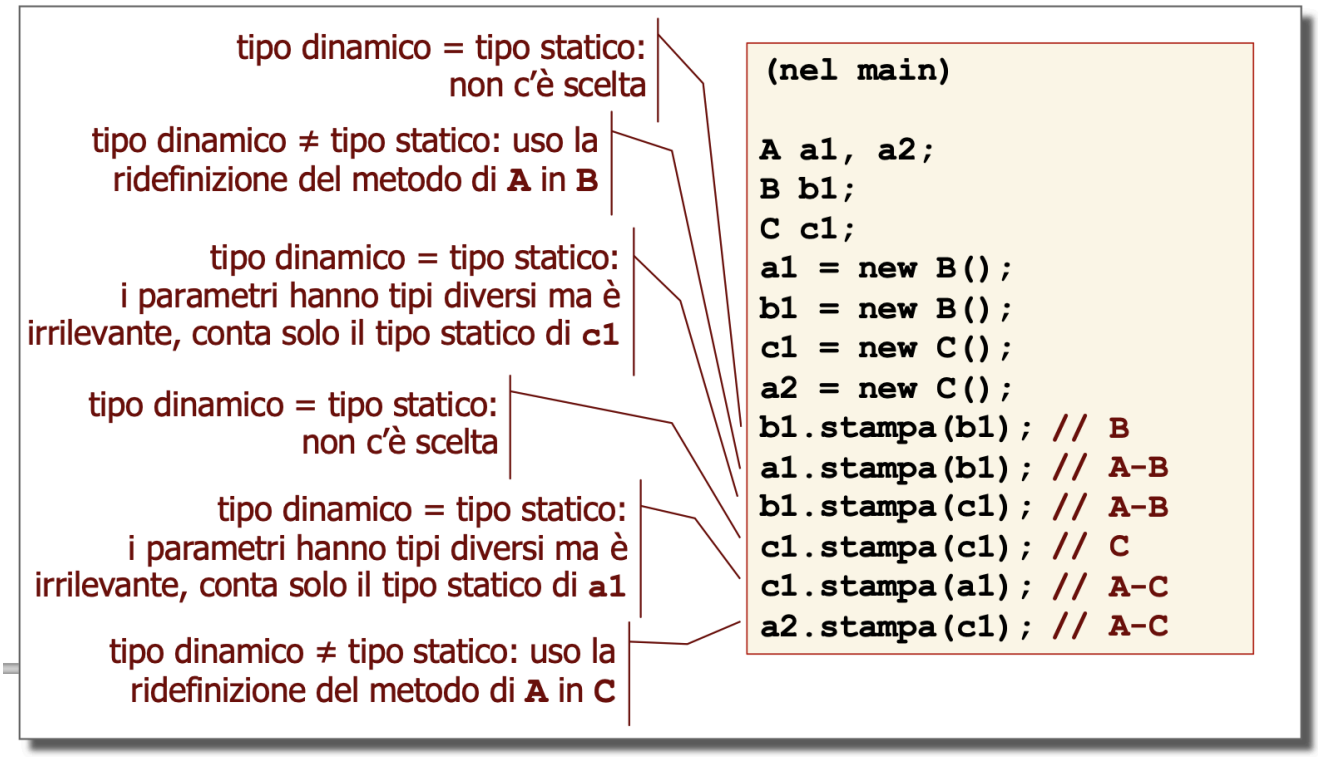
```
s.chiSiete(ps);
```
tipo dinamico = tipo statico: uso il metodo con input **Persona**
perché devo attenermi al tipo statico del parametro

```
ps.chiSiete(p);
ps.chiSiete(s);
ps.chiSiete(ps);
```
tipo dinamico ≠ tipo statico: posso usare soltanto il metodo visibile nel tipo statico, con input **Persona**

```
public class A {
  public void stampa(A p) {
    System.out.println("A");
  }
}
public class B extends A {
  public void stampa(B p) {
    System.out.println("B");
  }
  public void stampa(A p) {
    System.out.println("A-B");
  }
}
public class C extends A {
  public void stampa(C p) {
    System.out.println("C");
  }
  public void stampa(A p) {
    System.out.println("A-C");
  }
}
```

```
(nel main)

A a1, a2;
B b1;
C c1;
a1 = new B();
b1 = new B();
c1 = new C();
a2 = new C();
b1.stampa(b1);
a1.stampa(b1);
b1.stampa(c1);
c1.stampa(c1);
c1.stampa(a1);
a2.stampa(c1);
```

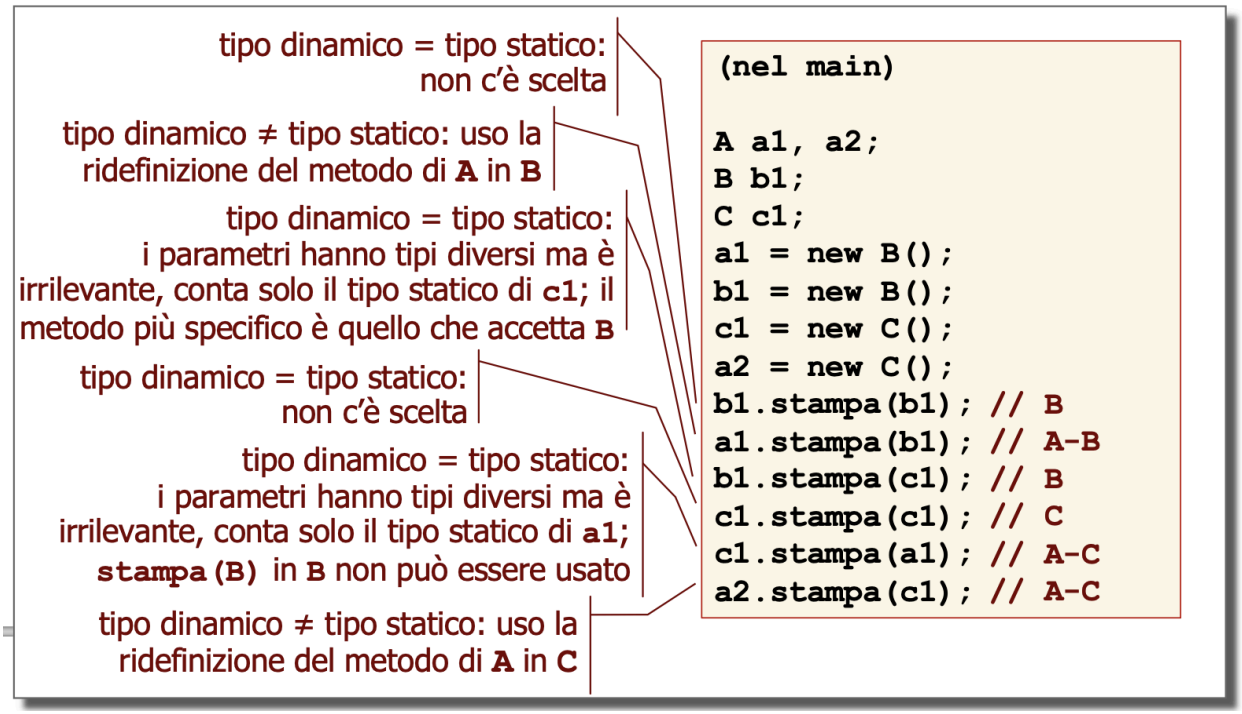


Oppure, cambiando leggermente il codice si ottiene un nuovo tipo di esercizio:

```
public class A {
  public void stampa(A p) {
    System.out.println("A");
  }
}
public class B extends A {
  public void stampa(B p) {
    System.out.println("B");
  }
  public void stampa(A p) {
    System.out.println("A-B");
  }
}
public class C extends B {
  public void stampa(C p) {
    System.out.println("C");
  }
  public void stampa(A p) {
    System.out.println("A-C");
  }
}
```

```
(nel main)

A a1, a2;
B b1;
C c1;
a1 = new B();
b1 = new B();
c1 = new C();
a2 = new C();
b1.stampa(b1);
a1.stampa(b1);
b1.stampa(c1);
c1.stampa(c1);
c1.stampa(a1);
a2.stampa(c1);
```



Infine un ultimo esempio pratico che coinvolge più funzioni:

```
class A {
  private int val;
  public A(int v) { val = v; }
  public int valore() { return val; }
  public int somma(A o) { return valore() + o.valore(); }
}
class B extends A {
  B(int v) { super(v); }
  public int somma(A o) {
    return valore() +
      o.valore() + 2;
  }
  public int somma(B o) {
    return valore() +
      o.valore() + 1;
  }
}
```

```
public class Prova {
  public static void main(String[] args) {
    A a1, a2;
    B b;
    a1 = new A(4);
    a2 = new B(5);
    b = new B(6);
    System.out.println(a1.somma(a2)); // 9
    System.out.println(a2.somma(b));  // 13
    System.out.println(b.somma(a1));  // 12
    System.out.println(b.somma(b));}  // 13
}
```

Possiamo concludere affermando che in Java le decisioni di base sono **sempre decise a runtime** salvo quando è possibile una decisione automatica a compilation time, questo è il caso di:

- costruttori
- metodi **static**, **private**, **final**

Classi e metodi final

In Java è possibile impedire l'ulteriore creazione di nuove sottoclassi aggiungendo la clausola " final " prima della denominazione della classe. Analogamente porre il *final* davanti ad un metodo ne **impedisce l'overriding**. Nel caso di una mancata attenzione a queste clausole l'errore che viene riportato è un errore a COMPILATION TIME. Una possibile alternativa per ovviare a questo errore è agire sul metodo precedentemente classificato come *final* e classificarlo come *private*, in questo modo esso non potrà essere visto dalle sottoclassi e verrà chiamato per forza di cose il metodo della superclasse.

Overriding di metodi static

Quando si pratica un override dei metodi static le **regole del binding dinamico cadono** e si fa riferimento esclusivamente a un **binding statico**, ciò è osservabile nell'esempio sotto: