Java: Basilari di Gerarchie di Classi

Sommario: 2 Maggio, 2018

- Classi e Sottoclassi:
  Oggetti, Costruttori e il costrutto **new**
- Sottoclassi:
  Interfacce, Ereditarietà e Shadowing
- Overriding di metodi
- Binding dinamico dei metodi:
  Late Binding
- Overloading e Overriding:
  Rischi e Cautele.
- Esercizi

Classi e Sottoclassi: Oggetti e Costruttori

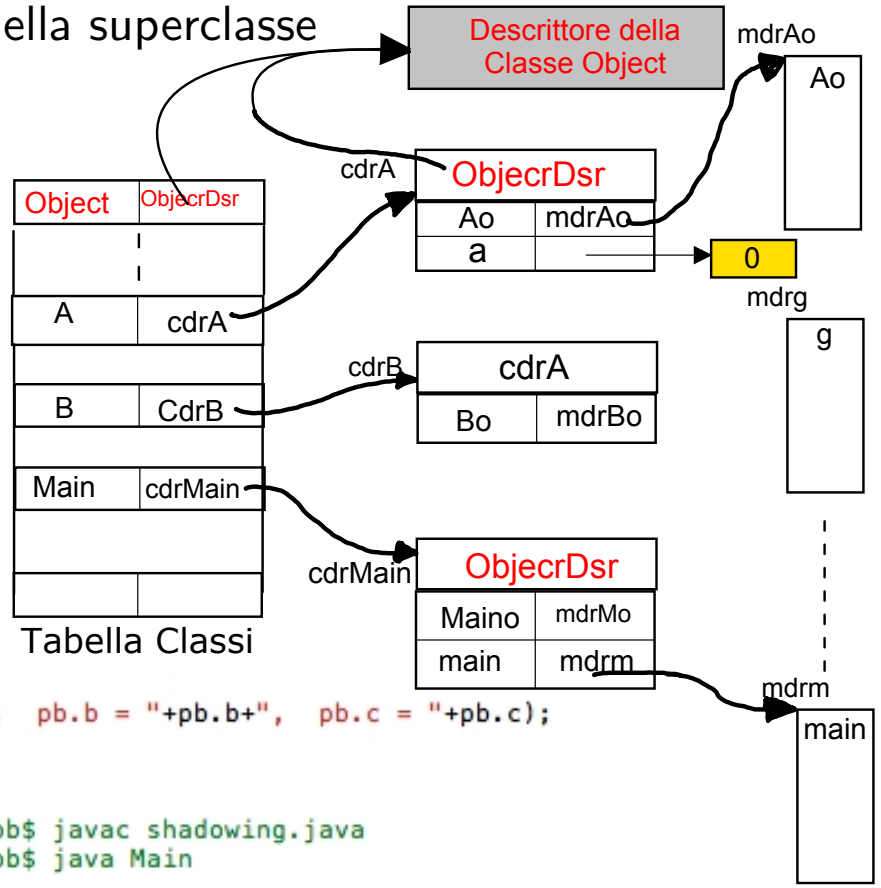
- **Sottoclasse** Estende la gerarchia delle classi del programma:
  - Estende campi e metodi della (super)classe
  - Eredita campi e metodi della superclasse

```
class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    int c;
    void h(){b=(a=5)+(c=2);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("pb.a = "+pb.a+",  pb.b = "+pb.b+",  pb.c = "+pb.c);
    }
}
/*
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ javac shadowing.java
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ java Main
 pb.a = 5,  pb.b = 7,  pb.c = 2
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$
*/
```

Classi e Sottoclassi: Oggetti e Costruttori

- **Sottoclasse** Estende la gerarchia delle classi del programma:
  - Estende campi e metodi della (super)classe
  - Eredita campi e metodi della superclasse

(a destra: vista delle classi)

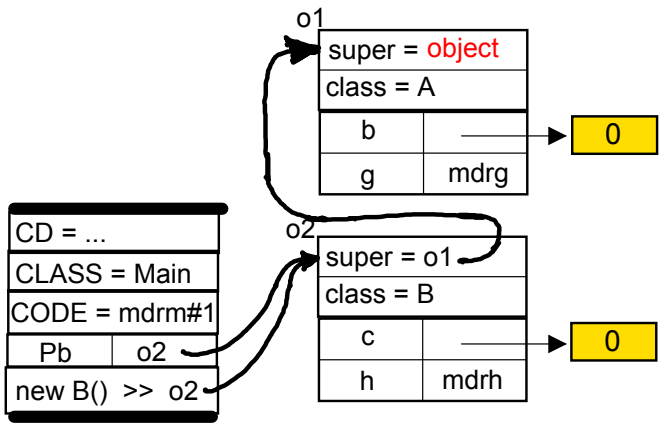



```
class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    int c;
    void h(){b=(a=5)+(c=2);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("pb.a = "+pb.a+",  pb.b = "+pb.b+",  pb.c = "+pb.c);
    }
}
/*
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ javac shadowing.java
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ java Main
 pb.a = 5,  pb.b = 7,  pb.c = 2
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$
*/
```

Classi e Sottoclassi: Oggetti, Costruttori e il costrutto new

- **Sottoclasse** Estende la gerarchia delle classi del programma:
  - Gli oggetti estendono campi e metodi degli oggetti della super
  - Gli oggetti ereditano campi e metodi degli oggetti della super

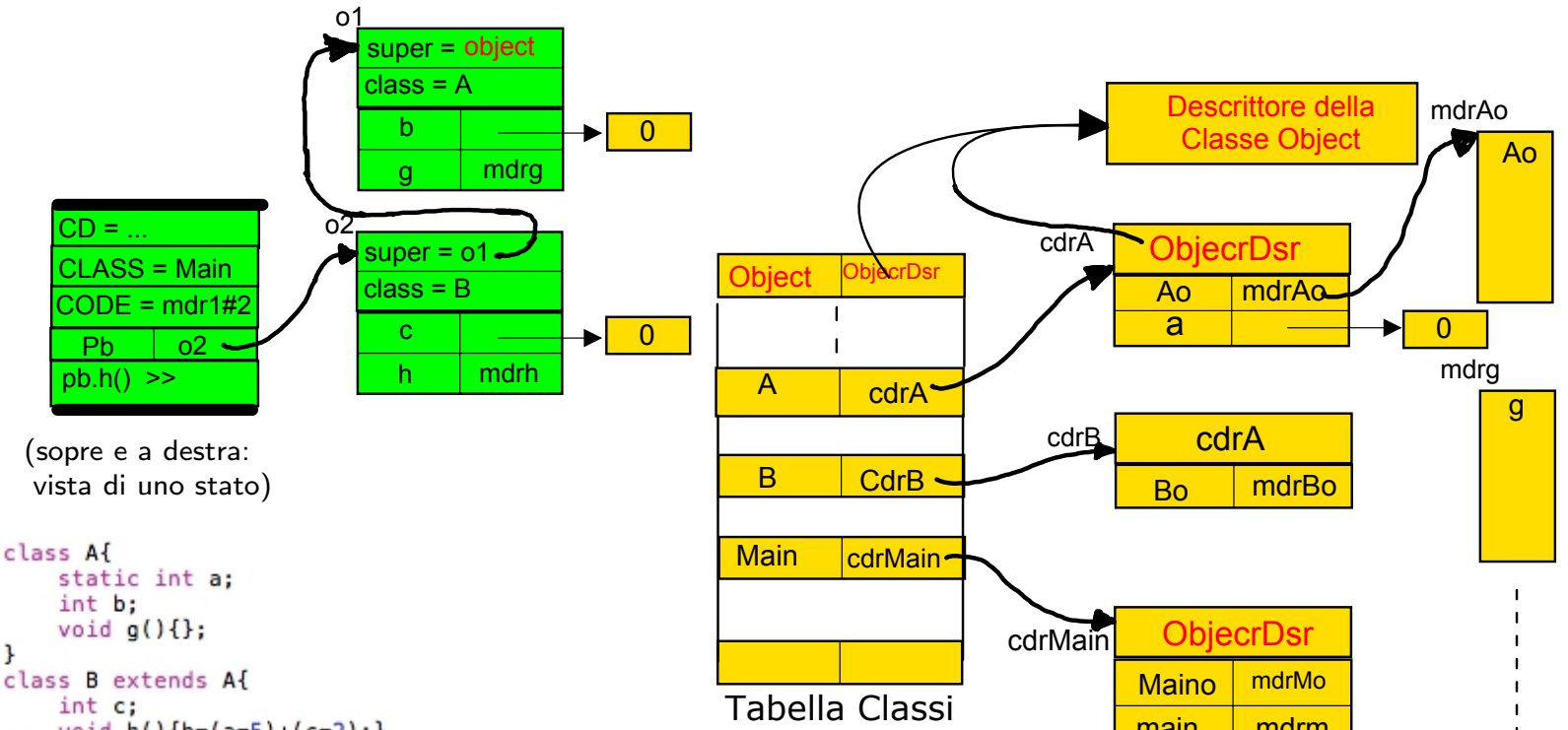
(a destra: creazione e vista di un oggetto di tipo B)




```
class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    int c;
    void h(){b=(a=5)+(c=2);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("pb.a = "+pb.a+",  pb.b = "+pb.b+",  pb.c = "+pb.c);
    }
}
/*
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ javac shadowing.java
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ java Main
 pb.a = 5,  pb.b = 7,  pb.c = 2
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$
*/
```

Classi e Sottoclassi: Oggetti e Costruttori

- **Sottoclasse** Estende la gerarchia delle classi del programma:



(sopre e a destra:
vista di uno stato)

```
class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    int c;
    void h(){b=(a=5)+(c=2);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("pb.a = "+pb.a+",  pb.b = "+pb.b+",  pb.c = "+pb.c);
    }
}
/*
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$ javac shadowing.java
```

Ereditarietà: Shadowing

- **shadowing** Una sottoclasse ridefinisce un field, e accede di default ...

```
import java.io.*;
import java.util.*;

class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    static int a;
    int c;
    void h(){b=(a=5)+(c=2);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("A.a = "+A.a+"   , B.a = "+B.a);
    }
}
/*
 bellia:shadowing marcobellia$ cd code
 bellia:code marcobellia$ java Main
 A.a = 0   , B.a = 5
 Marco-Bellias-MacBook-Pro:sottoclasse marcob$
*/
```

Ereditarietà: Shadowing/2

- **shadowing** Una sottoclasse ridefinisce un field, accede di default ... Ma può accedere a entrambi. In questo caso, lo può fare in due modi diversi:

```
import java.io.*;
import java.util.*;

class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    static int a;
    int c;
    void h(){b=(a=5)+(c=2)+(A.a=15);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("A.a = "+A.a+"   , B.a = "+B.a);
    }
}
```

- Applicabile perchè il field è di classe.
- Il secondo modo, accede al campo attraverso l'oggetto

Ereditarietà: Shadowing/3

- **shadowing** Una sottoclasse ridefinisce un field, accede di default ... Ma può accedere a entrambi, in due modi diversi:

```
import java.io.*;
import java.util.*;

class A{
    static int a;
    int b;
    void g(){};
}
class B extends A{
    static int a;
    int c;
    void h(){b=(a=5)+(c=2)+(super.a=15);}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        pb.h();
        System.out.println("A.a = "+A.a+"   , B.a = "+B.a);
    }
}
```

- Applicabile perchè il field è di classe.
- Il secondo modo, accede al campo attraverso l'oggetto

Ereditarietà: Overriding

- E l'analogo dello shadowing su metodi anzichè campi.
- Una sottoclasse ridefinisce un metodo della super, rispettando le seguenti 3 condizioni:
  - metodo di istanza (non si applica a costruttori)
  - stesso nome, stessi tipi degli argomenti,
  - eventuale tipo del valore calcolato ed eventuali tipi delle eccezioni sollevabili devono essere sotto-tipi del metodo della super.

```
class Point {
    double x;
    Point(double n1){
        x = n1;
    }
    public double distance (Point p){
        return Math.abs(x-p.x);
    }
}
class D2Point extends Point{
    double y;
    D2Point(double n1, double n2){
        super(n1);
        y = n2;
    }
    public double distance (Point p){
        double u = y-((D2Point)p).y;
        double d = super.distance(p);
        return Math.sqrt(d*d+u*u);
    }
}
```

- Riuso di codice.

Ereditarietà: Overriding e Late Binding

- Quale binding deve assegnare il compilatore all'identificatore `"f"` che compare nell'invocazione `"aa.f()"`?

```
import java.io.*;
import java.util.*;

class A{
    static int a;
    char c;
    void g(){};
    void f(){System.out.println("sono oggetto di classe effettiva A");}
}
class B extends A{
    static int a;
    int b;
    void h(){B.a=5+(a=3);};
    void f(){System.out.println("sono oggetto di classe effettiva B");}
}

class Main {
    public static void main(String[] args){
        B pb = new B();
        A pa = new A();
        A aa = pb;//ad aa posso assegnare sia pb sia pa
        aa.f();
        ((B)aa).h();
        System.out.println("A.a = "+A.a+",  B.a = "+B.a);
    }
}
```

- **Late Binding** Il binding è stabilito dinamicamente, guardando:
  - ◇ il tipo effettivo dell'oggetto calcolato dall'espressione di invocazione `"aa"` nel nostro caso.

Ereditarietà: Overriding e (Down) Cast

- La variabile "aa" ha comunque, tipo A e l'espressione "aa.h()" non è (sempre) definita.

```
import java.io.*;
import java.util.*;

class A{
    static int a;
    char c;
    void g(){};
    void f(){System.out.println("sono il metodo f di A");}
}
class B extends A{
    static int a;
    int b;
    void h(){B.a=5+(a=3);};
    void f(){System.out.println("sono il metodo f di B");}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        A pa = new A();
        A aa = pb;//ad aa posso assegnare sia pb sia pa
        aa.f();
        //aa.h(); -- error: cannot find symbol h
        ((B)aa).h(); //il cast rimanda il controllo al tempo di esecuzione
        System.out.println("A.a = "+A.a+",  B.a = "+B.a);
    }
}
/*
 Marco-Bellias-MacBook-Pro:overridingLateBinding marcob$ java Main
 sono il metodo f di B
 A.a = 0,  B.a = 8
 Marco-Bellias-MacBook-Pro:overridingLateBinding marcob$
*/
```

Ereditarietà: Overriding e (Down) Cast/2

- **(Down) Cast** deve essere utilizzato: (T)E "dichiara" che il tipo effettivo del valore calcolato da E sia T.

```
import java.io.*;
import java.util.*;

class A{
    static int a;
    char c;
    void g(){};
    void f(){System.out.println("sono il metodo f di A");}
}
class B extends A{
    static int a;
    int b;
    void h(){B.a=5+(a=3);};
    void f(){System.out.println("sono il metodo f di B");}
}
class Main {
    public static void main(String[] args){
        B pb = new B();
        A pa = new A();
        A aa = pa;//ad aa posso assegnare sia pb sia pa
        aa.f();
        //aa.h(); -- error: cannot find symbol h
        ((B)aa).h(); //il cast rimanda il controllo al tempo di esecuzione
        System.out.println("A.a = "+A.a+",  B.a = "+B.a);
    }
}
/*
 Marco-Bellias-MacBook-Pro:overridingLateBinding marcob$ java Main
 sono il metodo f di A
 Exception in thread "main" java.lang.ClassCastException: A cannot be cast to B
 at Main.main(staticE.java:25)
 Marco-Bellias-MacBook-Pro:overridingLateBinding marcob$
*/
```

Overloading vs. Overriding

- **Overloading.** Metodi statici e non, ereditati e non, che hanno stesso nome ma a coppie, segnature differenti:
  - per numero di argomenti, o per tipo di un argomento, oppure
  - se uno è ereditato, ha tipo del valore calcolato che **non è ~~su-~~ ~~pertipo~~** ~~o argomenti di sottotipo, dell'altro~~.

- Metodi overloaded sono tutti visibili ed applicabili;

- Nell'invocazione di metodo overloaded, a compile time, è scelto quello tra gli applicabili più prossimo al tipo atteso;

- Cautela. Errori nella definizione di un metodo overridden, rendono il metodo della superclasse un m. overloaded ed applicabile quando invece ci si attendeva che fosse "scavalcato".

Overloading: Metodo Invocato

- **Overloading.** Metodi statici e non, ereditati e non, che hanno stesso nome ma a coppie, segnature differenti:
  - per numero di argomenti, o per tipo di un argomento, oppure
  - se uno è ereditato, ha tipo del valore calcolato che **non è supertipo** o argomenti di sottotipo, dell'altro.

- Nell'invocazione di metodo overloaded, a compile time, è scelto quello tra gli applicabili più prossimo al tipo atteso;

```
class A{}
class B extends A{}
class C extends B{}

class E{
    void over(A x, A y){//overloaded
        System.out.println("sono overAA di E");
    }
    void over(A x, B y){//overloaded
        System.out.println("sono overAB di E");
    }
    void over(B x, C y){//overloaded
            System.out.println("sono overBC di E");
    }
}
class Main{
    public static void main(String[] argv){
        A x = new A();
        B y1 = new B();
        B y2 = new B();
        new E().over(y1,y2);
    }
```

Overloading: Rischi

- Cautela. Errori nella definizione di un metodo overridden, rendono il metodo della superclasse un metodo overloaded ed applicabile ...

```
package D2Point;

import java.io.*;
import java.util.*;

class Point {
    double x;
    Point(double n1){
        x = n1;
    }
    public double distance (Point p){
        return Math.abs(x-p.x);
    }
}
class D2Point extends Point{
    double y;
    D2Point(double n1, double n2){
        super(n1);
        y = n2;
    }
    public double distance (Point p){
        double u = y-((D2Point)p).y;
        double d = super.distance(p);
        return Math.sqrt(d*d+u*u);
    }
}
class Main {
    public static void main(String[] args){
        Point p = new Point(3);
        D2Point q = new D2Point(0,3);
        System.out.println("come si comporta la valutazione di
            q.distance(p)? " + q.distance(p));
    }
```