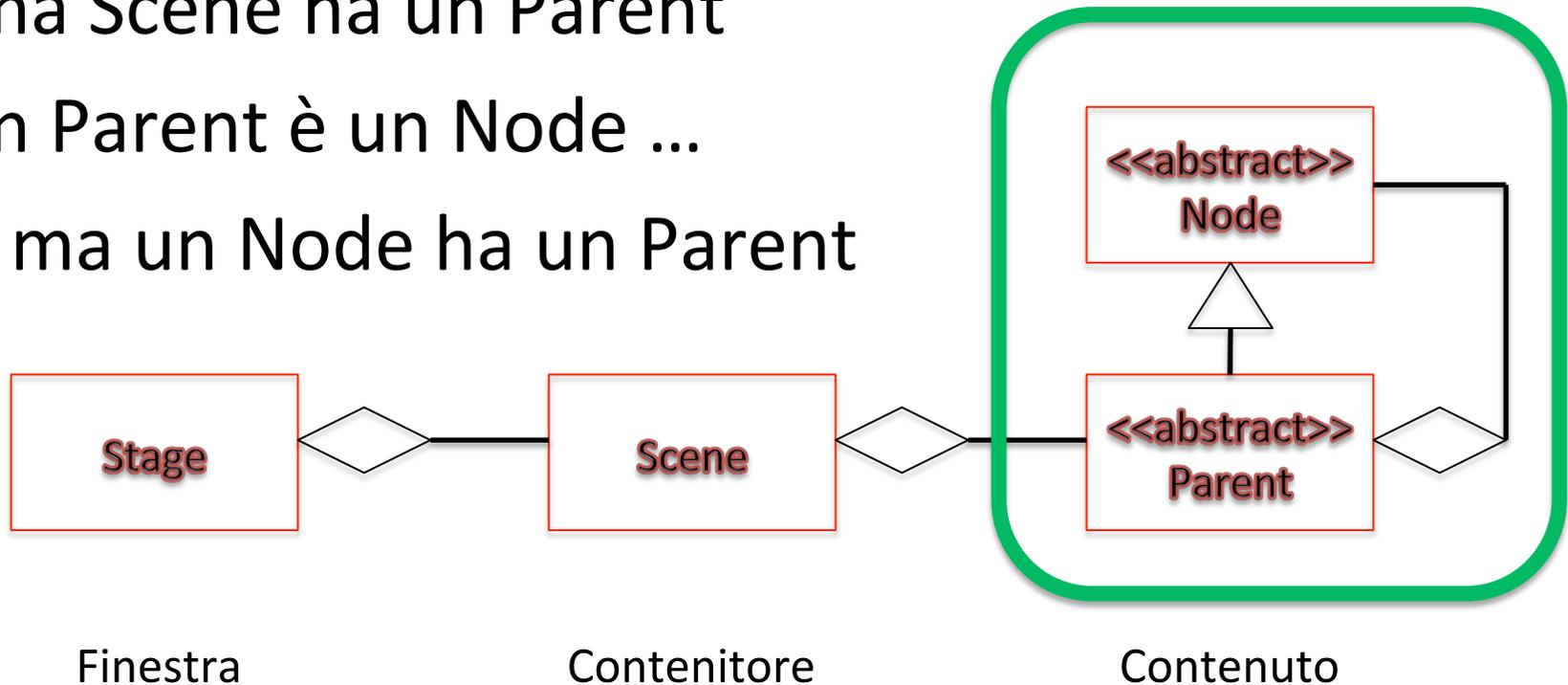


# Stage/Scene/Parent/Node

Stage = “finestra”

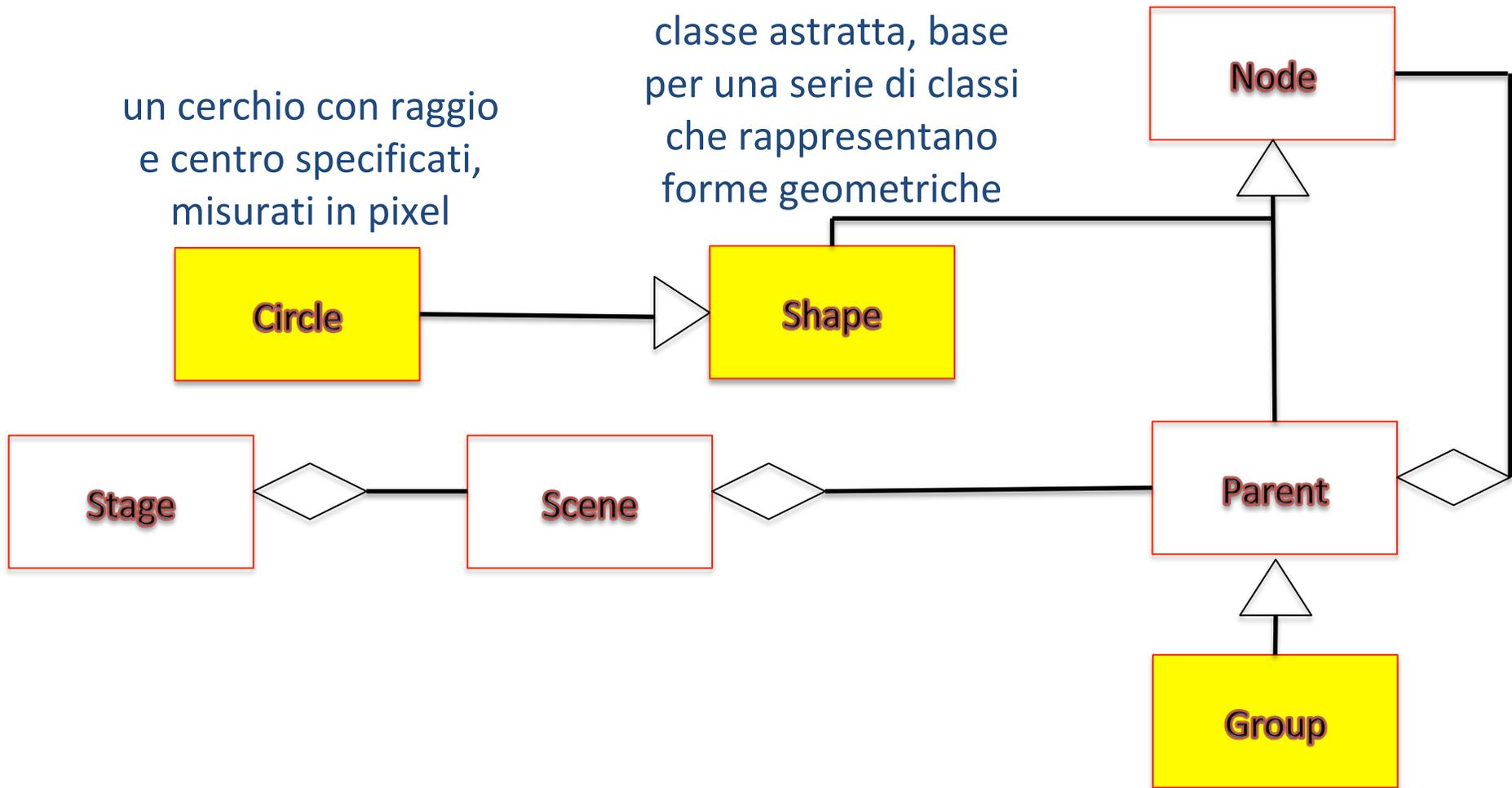
- Uno Stage contiene una Scene
- Una Scene ha un Parent
- Un Parent è un Node ...
- ... ma un Node ha un Parent



# Group – Shape - Circle

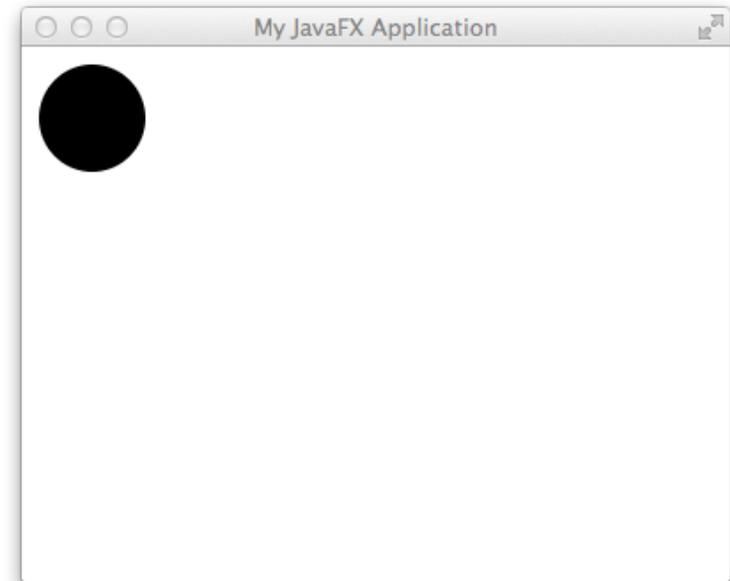
un cerchio con raggio  
e centro specificati,  
misurati in pixel

classe astratta, base  
per una serie di classi  
che rappresentano  
forme geometriche



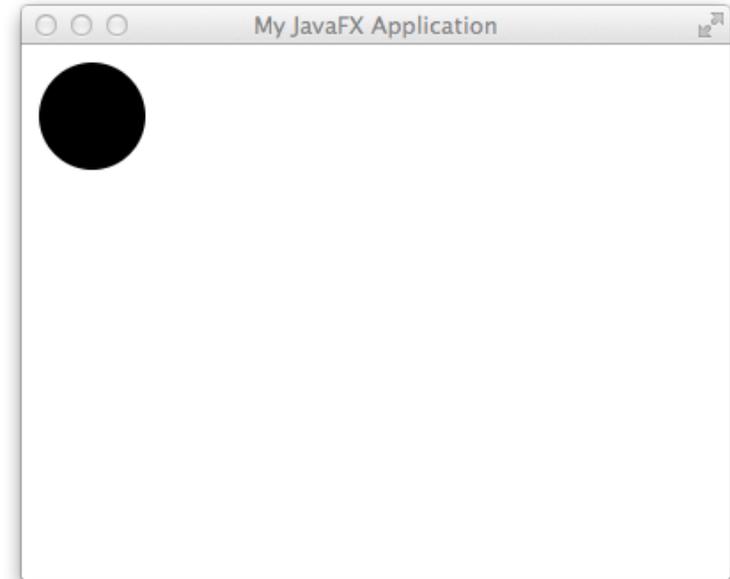
# Applicazione minima

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
public class MinimalApp extends Application {
    public void start(Stage stage) {
        Circle circ = new Circle(40, 40, 30);
        Group root = new Group(circ);
        Scene scene = new Scene(root, 400, 300);
        stage.setTitle("My JavaFX Application");
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```



# Applicazione minima

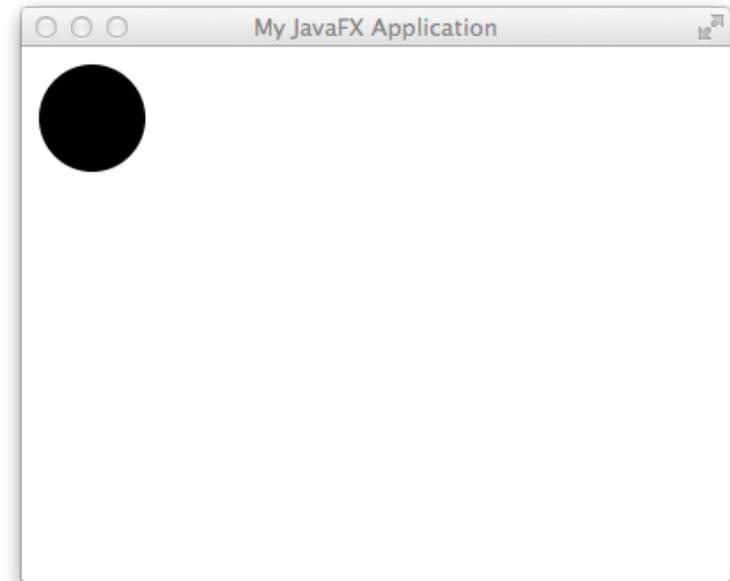
```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
public class MinimalApp extends Application {
    public void start(Stage stage) {
        Node circ = new Circle(40, 40, 30);
        Parent root = new Group(circ);
        Scene scene = new Scene(root, 400, 300);
        stage.setTitle("My JavaFX Application");
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```



Superclasse a  
Sx dell'uguale!

# Applicazione minima

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
public class MinimalApp extends Application {
    public void start(Stage stage) {
        Circle circ = new Circle(40, 40, 30);
        Group root = new Group();
        root.getChildren().addAll(circ);
        Scene scene = new Scene(root, 400, 300);
        stage.setTitle("My JavaFX Application");
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```



Modo alternativo  
di aggiungere  
componenti

# Static and dynamic binding

# Polimorfismo

- Polimorfismo (in generale): la capacità di assumere forme diverse
- Polimorfismo (nei linguaggi di programmazione): la capacità di un elemento sintattico di riferirsi a elementi di diverso tipo
- Es.  $f(\text{Persona } p)$ ,  $f(\text{Gatto } g)$ ;

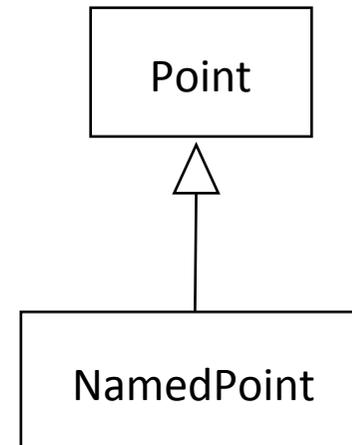
## Principio di sostituzione di Liskov:

Se  $S$  è un sottotipo di  $T$ , allora variabili di tipo  $T$  in un programma possono essere sostituite da variabili di tipo  $S$  senza alterare alcuna proprietà desiderabile del programma

```
Point p=new Point(1,2);  
p.move(3,4);
```

Ovunque ci sia un Point posso  
mettere un NamedPoint

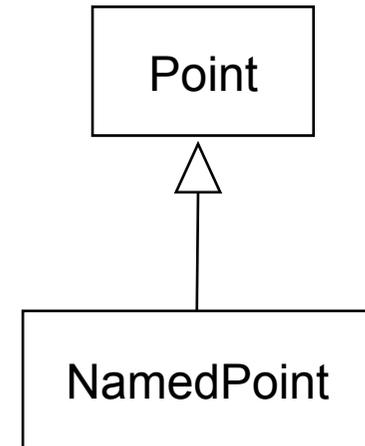
```
Point p=new NamedPoint(1,2,"A");  
p.move(3,4);
```



# Polimorfismo in Java: esempio 2

```
public class Line {  
    Point p1, p2;  
    Line(Point p1, Point p2)  
    {...}  
}
```

```
Point p1 = new Point(1,2);  
NamedPoint p2 = new  
NamedPoint(5,7,"A");  
Line l = new Line(p1, p2);
```



- Ovunque c'è un **Point** posso mettere un **NamedPoint** ...

# Polimorfismo in Java

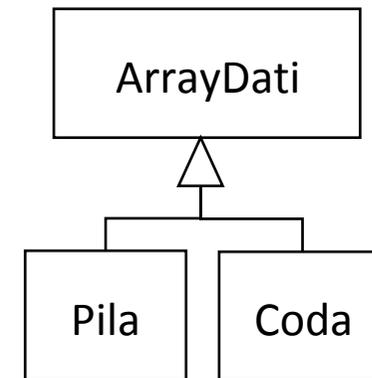
- Una variabile di tipo riferimento **T** può riferirsi ad un qualsiasi oggetto il cui tipo sia **T** **o un suo sottotipo**
- Analogamente, un parametro formale di tipo riferimento **T** può riferirsi a parametri attuali il cui tipo sia **T** **o un suo sottotipo**

# Altro esempio

```
class Automobile {...}
class AutomobileElettrica extends Automobile {...}
class Parcheggio {
    private Automobile buf[];
    private int nAuto;
    public Parcheggio(int dim) {buf=new Automobile[dim];}
    public void aggiungi(Automobile a) {buf[nAuto++]=a;}
}
...
AutomobileElettrica b = new AutomobileElettrica();
Automobile a = new Automobile();
Parcheggio p = new Parcheggio(100);
p.aggiungi(a);
p.aggiungi(b);
```

# Decisioni al volo...

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



Il tipo di **p** viene deciso a **runtime!**

Il legame tra un oggetto e il suo tipo è **dinamico**  
(*dynamic binding, late binding, o lazy evaluation*)

# Binding dinamico: esempio

```
class Persona {
    private String nome;
    public Persona(String nome) { this.nome = nome;}
    public void chiSei() {
        System.out.println("Ciao, io sono " + nome);
    }
}
class Studente extends Persona {
    public Studente(String nome) { super(nome); }
    public void chiSei() {
        super.chiSei();
        System.out.println(" e sono uno studente");}
}
...
Persona p = new Studente("Giovanni");
p.chiSei();
```

Output:

**Ciao, io sono Giovanni e sono uno  
studente**

# Tipo statico e tipo dinamico

- In presenza di polimorfismo si distingue tra il **tipo statico** (dichiarato a compilation time) ed il **tipo dinamico** (assunto a runtime) di una variabile o parametro formale
- In Java, in un invocazione  $\mathbf{x} . \mathbf{f} (\mathbf{x1} , . . . , \mathbf{xn})$ , l'implementazione scelta per il metodo  $\mathbf{f}$  dipende dal tipo dinamico di  $\mathbf{x}$  e non dal suo tipo statico

# Tipo statico e tipo dinamico

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

A x = new A() // tipo statico A, tipo dinamico A

B y = new B() // tipo statico B, tipo dinamico B

A z = new B() // tipo statico A, tipo dinamico B

B w = new A() // illegale

# Assegnazione illegale

B w = new A() // illegale

```
class A {  
    int x;  
}  
class B extends A {  
    int y;  
}
```

Infatti w.y non avrebbe senso:

non c'è il campo y nella variabile istanziata!

# Regola 1

Il compilatore determina la firma del metodo da eseguire basandosi sempre sul **tipo statico**.

# Esercizio: determinare l'output

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

```
public void f(A x) {System.out.println((x instanceof B)+" A");}  
public void f(B x) {System.out.println((x instanceof B)+" B");}
```

```
public static void main (String a[]) {  
    A a=new A();  
    A ab=new B();  
    B b=new B();  
  
    this.f(a);  
    this.f(ab);  
    this.f(b);  
    this.f((A)b);  
}
```

# Esercizio: soluzione

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

```
public void f(A x) {System.out.println((x instanceof B)+" A");}  
public void f(B x) {System.out.println((x instanceof B)+" B");}
```

```
public static void main (String a[]) {  
    A a=new A();  
    A ab=new B();  
    B b=new B();
```

```
false A  
true A  
true B  
true A
```

```
    this.f(a);    //declared type of arg. is A, runtime type is A, calls f(A)  
    this.f(ab);  //declared type of arg. is A, runtime type is B, calls f(A)  
    this.f(b);   //declared type of arg. is B, runtime type is B, calls f(B)  
    this.f((A)b); //declared type of arg. is A, runtime type is B, calls f(A)
```

```
}
```

# Regola 2

In caso di **overriding** (e solo in questo caso), la specifica implementazione del metodo la cui firma è stata decisa dal compilatore viene determinata a runtime basandosi sul **tipo dinamico**.

# Esempio: determinare l'output

```
A a=new A();  
A ab=new B();  
B b=new B();
```

```
a.g(a);  
a.g(ab);  
a.g(b);
```

```
ab.g(a);  
ab.g(ab);  
ab.g(b);
```

```
b.g(a);  
b.g(ab);  
b.g(b);
```

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

# Esempio: determinare l'output - I

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

a.g(a);

a.g(ab);

a.g(b);

# Soluzione, parte 1.1: determinazione (statica) della firma

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

a.g(a); => A.g(A)

a.g(ab); => A.g(A)

a.g(b); => A.g(B) non c'è! Ma grazie a Liskov  
può essere sostituito con A.g(A)

# Soluzione, parte 1.2: binding (dinamico) del metodo

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

a.g(a); => A.g(A) => A.g(A)

a.g(ab); => A.g(A) => A.g(A)

a.g(b); => A.g(A) => A.g(A)

Vado a vedere chi c'è nella variabile a,  
trovo un oggetto di tipo A,  
Tutto è coerente non serve fare altro.

# Esempio: determinare l'output - II

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

ab.g(a);

ab.g(ab);

ab.g(b);

# Soluzione, parte 2.1: determinazione (statica) della firma

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

ab.g(a); => A.g(A)

ab.g(ab); => A.g(A)

ab.g(b); => A.g(B) non c'è! Ma grazie a Liskov  
può essere sostituito con A.g(A)

ab è formalmente di tipo A,  
quindi le risoluzioni (statiche) delle firme  
sono le stesse del caso precedente.

# Soluzione, parte 2.2: binding (dinamico) del metodo

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

ab.g(a); => A.g(A) => B.g(A)

ab.g(ab); => A.g(A) => B.g(A)

ab.g(b); => A.g(A) => B.g(A)

Vado a vedere chi c'è nella variabile a,  
trovo un oggetto di tipo B,  
E' un caso di overriding,  
quindi devo risolvere sul tipo dinamico.

# Esempio: determinare l'output - III

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

b.g(a);

b.g(ab);

b.g(b);

# Soluzione, parte 3.1: determinazione (statica) della firma

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

b.g(a); => B.g(A)

b.g(ab); => B.g(A)

b.g(b); => B.g(B)

# Soluzione, parte 3.2: binding (dinamico) del metodo

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

A a=new A();

A ab=new B();

B b=new B();

b.g(a); => B.g(A) => B.g(A)

b.g(ab); => B.g(A) => B.g(A)

b.g(b); => B.g(B) => B.g(B)

Vado a vedere chi c'è nella variabile b,  
trovo un oggetto di tipo B,  
Devo risolvere sul tipo dinamico,  
ma questo non implica nessuna variazione.

# Riassunto soluzione, parte 1: determinazione (statica) della firma

```
A a=new A();  
A ab=new B();  
B b=new B();
```

```
a.g(a); => A.g(A)  
a.g(ab); => A.g(A)  
a.g(b); => A.g(A)
```

```
ab.g(a); => A.g(A)  
ab.g(ab); => A.g(A)  
ab.g(b); => A.g(A)
```

```
b.g(a); => B.g(A)  
b.g(ab); => B.g(A)  
b.g(b); => B.g(B)
```

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

# Riassunto soluzione, parte 2: binding (dinamico) del metodo

```
A a=new A();  
A ab=new B();  
B b=new B();
```

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

a.g(a); => A.g(A) => A.g(A)

a.g(ab); => A.g(A) => A.g(A)

a.g(b); => A.g(A) => A.g(A)

ab.g(a); => A.g(A) => B.g(A)

ab.g(ab); => A.g(A) => B.g(A)

ab.g(b); => A.g(A) => B.g(A)

b.g(a); => B.g(A) => B.g(A)

b.g(ab); => B.g(A) => B.g(A)

b.g(b); => B.g(B) => B.g(B)

# Riassunto soluzione, parte 3: output

```
A a=new A();  
A ab=new B();  
B b=new B();
```

```
class A {  
    public void g(A x) {System.out.println("called on instance of A");}  
}  
class B extends A {  
    public void g(A x) {System.out.println("called 1 on instance of B");}  
    public void g(B x) {System.out.println("called 2 on instance of B");}  
}
```

```
a.g(a); => A.g(A) => A.g(A)   called on instance of A  
a.g(ab); => A.g(A) => A.g(A)   called on instance of A  
a.g(b); => A.g(A) => A.g(A)   called on instance of A
```

```
ab.g(a); => A.g(A) => B.g(A)   called 1 on instance of B  
ab.g(ab); => A.g(A) => B.g(A)   called 1 on instance of B  
ab.g(b); => A.g(A) => B.g(A)   called 1 on instance of B
```

```
b.g(a); => B.g(A) => B.g(A)   called 1 on instance of B  
b.g(ab); => B.g(A) => B.g(A)   called 1 on instance of B  
b.g(b); => B.g(B) => B.g(B)   called 2 on instance of B
```

# Riassunto: regola per il binding

- Si assuma  
 $C \ o = \dots;$   
 $o.m(\dots);$   
il metodo scelto dipende dal tipo dinamico di  $o$ ,  
e viene deciso (a runtime) con questa logica:
  1. Si cerca all'interno della classe  $C$  (tipo statico di  $o$ )  
il metodo con la firma «più vicina»  
all'invocazione
  2. Si guarda al tipo dinamico  $D$  di  $o$ ; se è un  
sottotipo di  $C$ , si deve verificare se ridefinisce  
(override)  $m$ . Se sì, si usa l'implementazione di  $D$ ,  
altrimenti quella di  $C$

# Static e dynamic binding

- Il C++ offre al programmatore complessi meccanismi per decidere se usare **dynamic binding** (tipo deciso a runtime) o **static binding** (tipo deciso a compile time)
- In C++ la keyword “virtual” abilita il dynamic binding, che altrimenti è sempre static
- In Java le decisioni sono sempre a runtime  
... salvo quando sia possibile decidere automaticamente a compile time, e cioè per:  
metodi **private**, **static** e **final**  
costruttori