

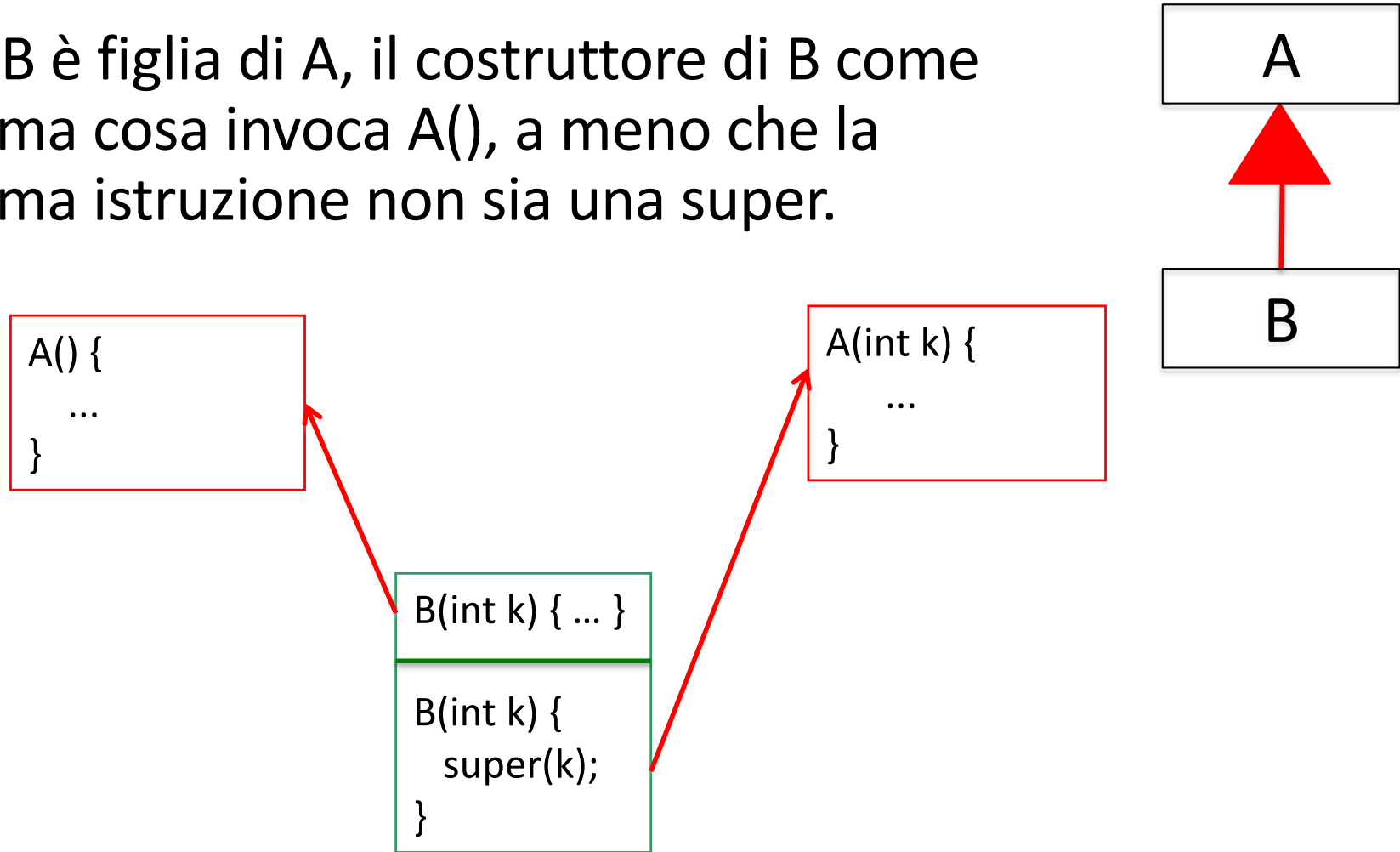
Costruttori

# Definizione dei costruttori

- Se per una classe A non scrivo nessun costruttore, **il sistema automaticamente crea il costruttore A();**
- Se invece definisco almeno un costruttore non void, ad es. A(int s), **il sistema non crea il costruttore A();**

# Definizione dei costruttori

- Se B è figlia di A, il costruttore di B come prima cosa invoca A(), a meno che la prima istruzione non sia una super.



# Invocazione dei costruttori

```
public class A {  
    public A() {  
        System.out.println("Creo A");  
    }  
}  
public class B extends A {  
    public B() {  
        System.out.println("Creo B");  
    }  
    public B(int k) {  
        System.out.println("Creo B_int");  
    }  
}
```

Caso 1.  
Qual è l'output ?

```
public static void main(String [] a) {  
    B b=new B(1);  
}
```

# Invocazione dei costruttori

```
public class A {  
    public A(int k) {  
        System.out.println("Creo A");  
    }  
}  
  
public class B extends A {  
    public B() {  
        System.out.println("Creo B");  
    }  
    public B(int k) {  
        System.out.println("Creo B_int");  
    }  
}
```

Caso 2.  
Qual è l'output ?

```
public static void main(String [] a) {  
    B b=new B(1);  
}
```

# Invocazione dei costruttori

```
public class A {  
    public A(int k) {  
        System.out.println("Creo A");  
    }  
}  
  
public class B extends A {  
    public B() {  
        System.out.println("Creo B");  
    }  
    public B(int k) {  
        super(k);  
        System.out.println("Creo B_int");  
    }  
}  
  
public static void main(String [] a) {  
    B b=new B(1);  
}
```

Caso 3.  
Qual è l'output ?

La chiamata a **super**  
**DEVE**  
essere la prima  
istruzione!

# Chiamata ad altro costruttore della stessa classe

```
public class P {  
    float[] x;  
    public P(int k) {  
        x=new float[k];  
    }  
    public P() {  
        this(100);  
    }  
}
```



La chiamata a **this**  
**DEVE**  
essere la prima  
istruzione!

# 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(Persona p), f(Gatto g);`

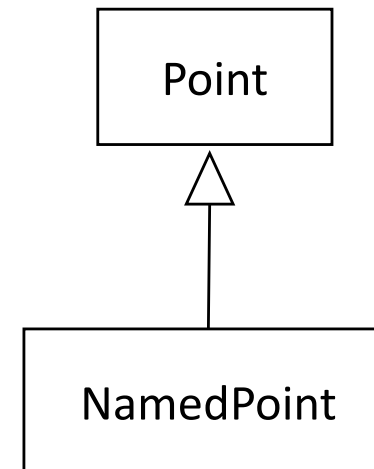
# Principio di sostituzione di Liskov:

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) ;
```



# Barbara Huberman Liskov

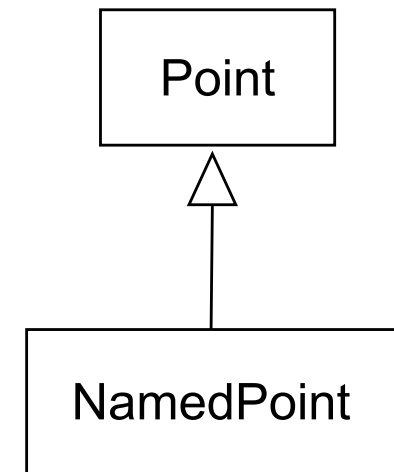
- prima donna negli Stati Uniti ad aver ricevuto un dottorato in Computer Science (1968)
- medaglia Neumann (2004)
- premio Turing (2008)



# 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

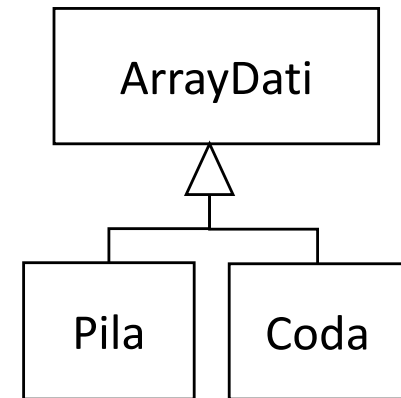
- Una variabile di tipo riferimento **T** può riferirsi ad un qualsiasi oggetto il cui tipo sia **T** **o un suo sottotipo**
- Analogamente, un parametro formale di tipo riferimento **T** può riferirsi a parametri attuali il cui tipo sia **T** **o un suo sottotipo**

# 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  $\mathbf{x.f(x_1, \dots, x_n)}$ , 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**.

## 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**.

# Come funziona il gioco?

- Compilatore:
  - "leggo il codice, e decido qual'è la firma del metodo da chiamare"
- Runtime:
  - "data la firma del metodo da chiamare (decisa dal compilatore), vedo quale scegliere tra quelle disponibili"

# 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 0 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 0 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");}  
}
```

*abbiamo un caso di overriding e uno di overloading*

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

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

```
class A {  
    public void g(A x) {System.out.println("called 0 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** è di tipo A, quindi le risoluzioni (statiche) delle firme sono ovvie (non ho opzioni).

**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) (perchè B is-a A)*

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

```
class A {  
    public void g(A x) {System.out.println("called 0 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();

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

**a.g(a);** => A.g(A) => A.g(A) => *"called 0..."*

**a.g(ab);** => A.g(A) => A.g(A) => *"called 0..."*

**a.g(b);** => A.g(A) => A.g(A) => *"called 0..."*

# Esempio: determinare l'output - II

```
class A {  
    public void g(A x) {System.out.println("called 0 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 0 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 0 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();

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

**ab.g(a);** => A.g(A) => B.g(A) => *"called 1..."*

**ab.g(ab);** => A.g(A) => B.g(A) => *"called 1..."*

**ab.g(b);** => A.g(A) => B.g(A) => *"called 1..."*

# Esempio: determinare l'output - III

```
class A {  
    public void g(A x) {System.out.println("called 0 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 0 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)

b è formalmente di tipo B,  
quindi le risoluzioni (statiche) delle firme  
sono quelle di B

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

```
class A {  
    public void g(A x) {System.out.println("called 0 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();

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.

b.g(a); => B.g(A) => B.g(A) => *"called 1..."*

b.g(ab); => B.g(A) => B.g(A) => *"called 1..."*

b.g(b); => B.g(B) => B.g(B) => *"called 2..."*



# 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 0 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 0 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 0 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 0 on instance of A  
a.g(ab); => A.g(A) => A.g(A)  called 0 on instance of A  
a.g(b);  => A.g(A) => A.g(A)  called 0 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  
 $\mathbf{C} \circ = \dots;$   
 $\circ.\mathbf{m}(\dots);$   
il metodo scelto dipende dal tipo dinamico di  $\circ$ ,  
e viene deciso (a runtime) con questa logica:
  1. Si cerca all'interno della classe  $\mathbf{C}$  (tipo statico di  $\circ$ )  
il metodo con la firma «più vicina»  
all'invocazione
  2. Si guarda al tipo dinamico  $\mathbf{D}$  di  $\circ$ ; se è un  
sottotipo di  $\mathbf{C}$ , si deve verificare se ridefinisce  
(override)  $\mathbf{m}$ . Se sì, si usa l'implementazione di  $\mathbf{D}$ ,  
altrimenti quella di  $\mathbf{C}$

# 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