

# Programmazione ad Oggetti

Polimorfismo  
Classi Astratte

---

A.A. 2022/2023

---

*Docente: Prof. Salvatore D'Angelo*  
*Email: [salvatore.dangelo@unicampania.it](mailto:salvatore.dangelo@unicampania.it)*



Università  
degli Studi  
della Campania  
*Luigi Vanvitelli*

*Dipartimento di Ingegneria*

# Polimorfismo

È il terzo meccanismo fondamentale della programmazione ad oggetti dopo:

- *Data Abstraction*
- *Information Hiding*
- *Ereditarietà* □ *Riuso*

IL *Polimorfismo* permette di *estendere* un progetto già sviluppato aggiungendo nuove funzionalità

# Polimorfismo statico e dinamico

Letteralmente, per polimorfismo si intende la proprietà di una entità di assumere forme diverse nel tempo.

Riferendoci ad un sistema software ad oggetti, il polimorfismo è la capacità che hanno oggetti di classi derivate da una classe base comune di rispondere in maniera diversa ad uno stesso messaggio

Una entità è polimorfa se può fare riferimento, nel tempo, a classi diverse.

# Polimorfismo

**Polimorfismo statico** □ polimorfismo a tempo di compilazione o binding static

**Polimorfismo dinamico** □ polimorfismo a tempo di esecuzione o di binding dinamico

# Early e late binding

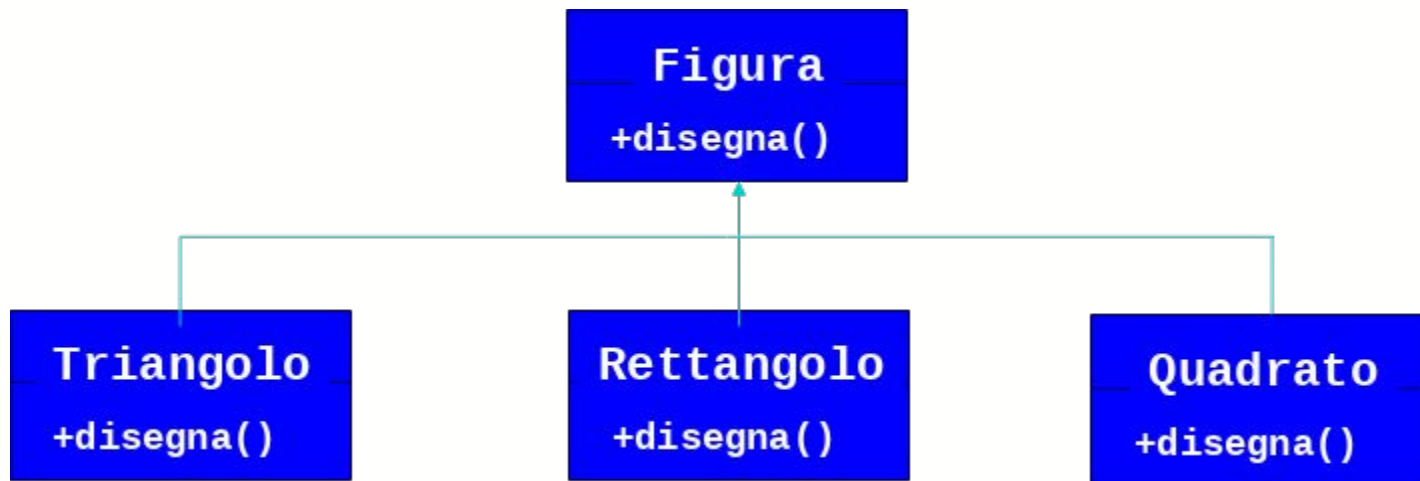
L'azione di collegare la chiamata di un metodo al codice corrispondente può avvenire a tempo di compilazione o a tempo di esecuzione

- Nel primo caso: ***early binding***
- Nel secondo caso: ***late binding***

# Early e late binding

Esempio:

Si consideri una gerarchia di classi di figure geometriche:



# Binding Statico

Sia ad es. A un vettore di N oggetti della classe Figura, composto di oggetti delle sottoclassi Triangolo, Rettangolo, Quadrato:  
Figura A[N] (ad es.: A[0] è un quadrato, A[1] un triangolo, A[2] un rettangolo, etc.)

Si consideri una funzione che disegna le figure contenute nel vettore A[i]:

```
for i = 1 to N do
  switch (A[i].tipo) {
    case 'Triangolo'
      A[i].DisegnaTriangolo()
    case 'Rettangolo'
      A[i].DisegnaRettangolo()
    case 'Quadrato'
      A[i].DisegnaQuadrato()
  }
```

# Esempio

Si consideri una funzione che contiene il seguente ciclo di istruzioni:

```
for i = 1 to N do  
    A[i].disegna()
```

L'esecuzione del ciclo richiede che sia possibile determinare dinamicamente (cioè a tempo d'esecuzione) l'implementazione della operazione disegna() da eseguire, in funzione del tipo corrente dell'oggetto A[i].

Un meccanismo con cui viene realizzato il polimorfismo è quello del *binding dinamico*.



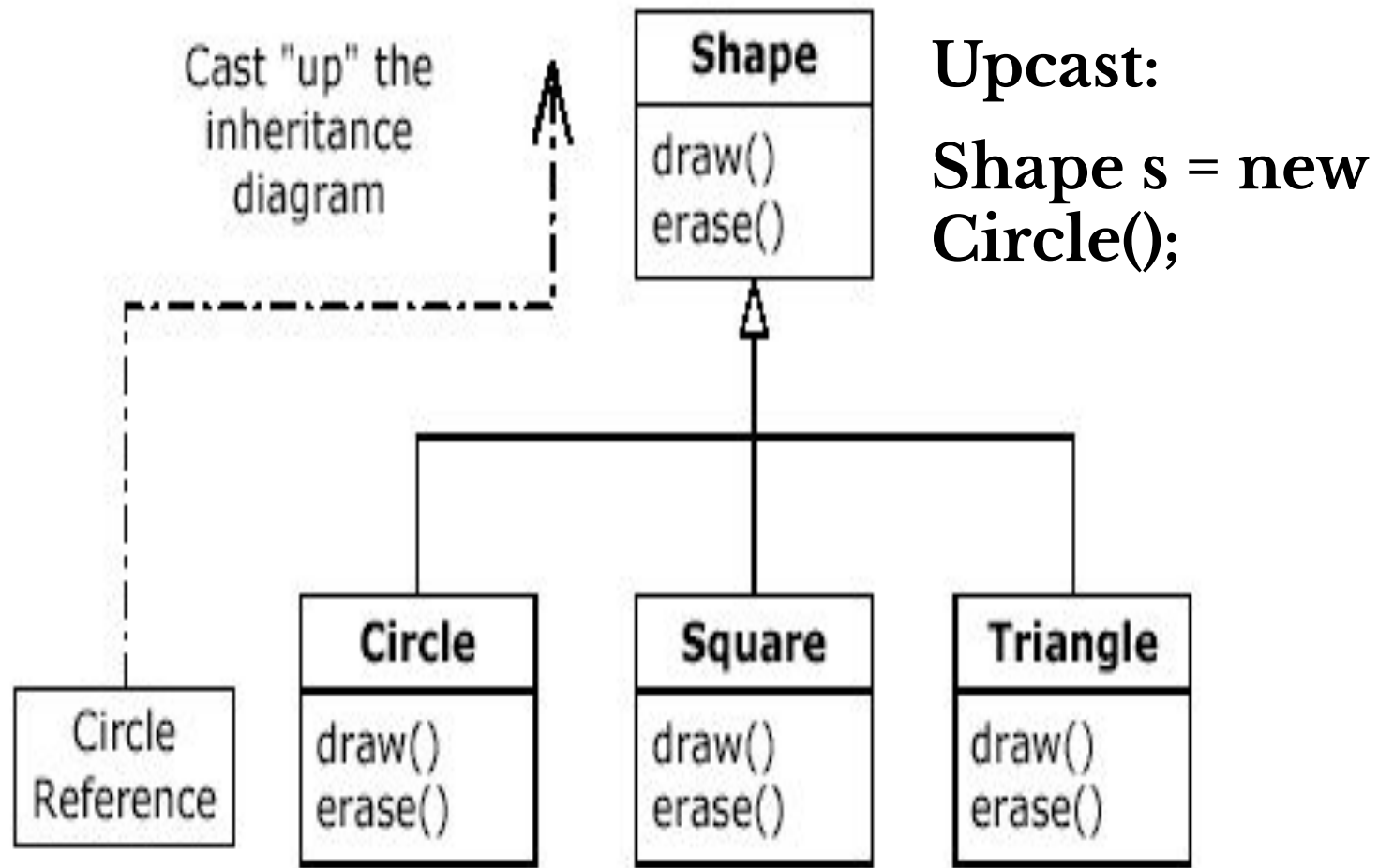
# Polimorfismo □ Estensibilità

Il polimorfismo supporta dunque la proprietà di **estensibilità** di un sistema, nel senso che minimizza la quantità di codice che occorre modificare quando si estende il sistema, cioè si introducono nuove classi e nuove funzionalità.

- L'istruzione `A[i].disegna()` non ha bisogno di essere modificato in conseguenza dell'aggiunta di una nuova sottoclasse di `Figura` (ad es.: `Cerchio`), anche se tale sottoclasse non era stata neppure prevista all'atto della stesura della funzione `disegna()`

```
for i = 1 to N do  
    A[i].disegna()
```

# Esempio



# Realizzazione delle classi

```
class Shape {  
    void draw() {}  
    void erase() {}}
```

```
class Circle extends Shape {  
    void draw() {System.out.println("Circle.draw()");}  
    void erase() {System.out.println("Circle.erase()");}}
```

```
class Square extends Shape {  
    void draw() {System.out.println("Square.draw()");}  
    void erase() {System.out.println("Square.erase()");}}
```

```
class Triangle extends Shape {  
    void draw() {System.out.println("Triangle.draw()");}  
    void erase() {System.out.println("Triangle.erase()");}}
```

# Late binding

```
class RandomShapeGenerator {
    private Random rand = new Random();
    public Shape next() {
        switch(rand.nextInt(3)) {
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();}
    }}

public class Shapes {
    private static RandomShapeGenerator gen =new
    RandomShapeGenerator();
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        for(int i = 0; i < s.length; i++)        s[i] = gen.next();    //
Classi note a tempo di esecuzione
        for(int i = 0; i < s.length; i++)        s[i].draw();
// Chiamata a metodo polimorfo
    }}
```

# Ereditarietà-Polimorfismo- Upcasting

```
class Instrument {
public void play() {System.out.println("Play Instrument"); }}

public class Wind extends Instrument {
    // Redefine interface method:
    public void play(Note n) {System.out.println("Wind.play() " + n);} }

public class Stringed extends Instrument {
    public void play(Note n) {System.out.println("Stringed.play() " + n);}}

class Brass extends Instrument {
    public void play(Note n) {System.out.println("Brass.play() " + n);}}

public class Music {
    public static void tune(Instrument i) {i.play(Note.MIDDLE_C);} //Polimorf.
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass bas= new Brass ();
        tune(flute);    // Upcasting
        tune(violin); // Upcasting
        tune(bas);      // Upcasting
    }}
}}
```

# Quale sarebbe l'alternativa?

```
class Instrument {
    public void play() {System.out.println("Play Instrument");} }

public class Wind extends Instrument {
    public void play(Note n) {System.out.println("Wind.play() " + n);}

class Stringed extends Instrument {
    public void play(Note n) {System.out.println("Stringed.play() " + n);}}

class Brass extends Instrument {
    public void play(Note n) {System.out.println("Brass.play() " + n);}}

public class Music2 {
    public static void tune(Wind i) { i.play(Note.MIDDLE_C);}
    public static void tune(Stringed i) {i.play(Note.MIDDLE_C);}
    public static void tune(Brass i) {i.play(Note.MIDDLE_C);}

    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // No upcasting
        tune(violin);
        tune(frenchHorn); }
```

# Quindi:

Il polimorfismo consente di separare il come dal cosa:

- Il come è l'interfaccia dell'oggetto
- Il cosa è il tipo o la classe di un oggetto

Nel precedente esempio riusciamo ad eseguire il codice specifico di una classe derivata senza conoscere il tipo, ovvero la classe a cui appartiene l'oggetto

# LA CLASSE OBJECT



# La classe Object

- La classe Object è la superclasse, diretta o indiretta, di ogni classe
- La classe Object definisce lo stato ed il comportamento base che ciascun oggetto deve avere e cioè l'abilità di
  - confrontarsi con un altro oggetto
  - convertirsi in una stringa
  - ritornare la classe dell'oggetto
  - attendere su una variabile condition
  - notificare che una variabile condition è cambiata

# I metodi di Object

- Che possono essere sovrascritti
  - clone
  - equals/hashCode
  - finalize
  - toString
- Che non possono essere sovrascritti
  - getClass
  - notify
  - notifyAll
  - wait

# Il metodo toString()

```
public String toString()
```

Crea una rappresentazione dell'oggetto sotto forma di stringa. La definizione originale di questo metodo è poco significativa: scrive il nome della classe e un indirizzo: Counter@712cla3c

# System.out.println()

- Vediamo in dettaglio perché le cose funzionano in questo modo
- System.out è un attributo della classe System
- E' di tipo PrintWriter (lo vedrete nelle lezioni sull'I/O), una classe che serve per scrivere a video e che ha una definizione di questo tipo:

```
public class PrintWriter extends Writer {  
    public void println(Object x) {  
        String s = x.toString();  
        ... }  
    ... }
```

- In virtù del subtyping questo implica che possiamo passare come parametro qualunque oggetto, dal momento che tutte le classi discendono da Object
- In virtù del polimorfismo questo implica che se la classe dell'oggetto passato come parametro ridefinisce il metodo toString() verrà invocato il metodo ridefinito

# Esempio: Deposito (1/4)

Scriviamo una semplice classe:

```
public class Deposito {  
    private float soldi;  
    public Deposito() { soldi=0; }  
    public Deposito(float s) { soldi=s; }  
}
```

- Dal momento che non abbiamo specificato nessun `extends` la classe discende direttamente da `Object`
- In quanto tale eredita il metodo `toString()`
- Dal momento che non lo ridefinisce invocandolo viene eseguita la versione originale definita in `Object`

# Esempio: Deposito (2/4)

Scriviamo poi una semplice applicazione che la usa:

```
public class EsempioDeposito {  
    public static void main(String args[]) {  
        Deposito d1 = new Deposito(312);  
        System.out.println(d1);  
    }  
}
```

A video otterremo:

**Deposito@712cla3c**

# Esempio: Deposito (3/4)

Aggiungiamo a Deposito il metodo toString, ridefinendo così quello ereditato da Object (overriding)

```
public class Deposito {  
    float soldi;  
    public Deposito() { soldi=0; }  
    public Deposito(float s) { soldi=s; }  
    public String toString() { return "Soldi:  
"+soldi; }  
}
```

# Esempio: Deposito (4/4)

Se usiamo la classe nell'applicazione precedente:

```
public class EsempioDeposito {  
    public static void main(String args[]) {  
        Deposito d1 = new Deposito(312);  
        System.out.println(d1);  
    }  
}
```

A video otterremo:

**Soldi: 312**



# IL metodo CLONE

In Java quando si vuole duplicare un oggetto si utilizza il metodo clone

- Crea un oggetto indipendente dall'originale
  - Modifiche sull'oggetto clonato non avranno ripercussioni sull'oggetto clonato
- `protected Object clone() throws  
CloneNotSupportedException`

Quindi:

- ritorna un Object
- è protected

# Utilizzo del metodo CLONE

Permette di effettuare una copia superficiale (per valore):

- viene allocato spazio in memoria heap e copiati tutti i valori delle variabili membro della classe originaria
- per le variabili membro di tipo complesso viene copiato solo il riferimento (modifiche all'oggetto originario vengono viste anche nel oggetto clonato)

# Overriding del metodo clone

```
Class Studente implements Cloneable {  
    // interfaccia fittizia  
    String nome;  
    int anno;  
  
    public Object clone() {  
        // Bisogna ridefinirlo public altrimenti non è visibile dalle altre classi  
        try {  
            return super.clone();  
        } catch (CloneNotSupportedException e) {  
            System.err.println("Implementation error");  
            System.exit(1);  
        }  
        return null; // qui non arriva mai  
    }  
}
```

# Utilizzo del metodo clone

```
public static void main(String args[]) {  
    Studente p, q;  
    p=new Studente();  
    p.nome = "Massimo";  
  
    q=(Studente) p.clone(); // Bisogna  
    effettuare un cast  
}
```

# Esempio

```
class P implements Cloneable {  
    int x; int y;  
    public String toString() {  
        return ("x="+x+" ; y="+y);  
    }  
    public Object clone(){  
        try {  
            return super.clone();  
        } catch (CloneNotSupportedException e) {  
            System.err.println("Implementation error");  
            System.exit(1);  
        }  
        return null; //qui non arriva mai }  
    }  
}
```

# Esempio

```
public class Test {  
    public static void main(String []a){new Test();}  
    Test() {  
        P p1=new P(); p1.x=5; p1.y=6;  
        P p2=p1;  
        P p3=p1.clone();  
        System.out.println(p1);  
        System.out.println(p2);  
        System.out.println(p3);  
        p1.x=7  
  
        System.out.println(p1);  
        System.out.println(p2);  
        System.out.println(p3);  
    }  
}
```

x=5 ; y=6

x=5 ; y=6

x=5 ; y=6

x=7 ; y=6

x=7 ; y=6

x=5 ; y=6

# Esempio - Cloning superficiale

```
class V implements Cloneable {  
    int x[];  
    V(int s) {  
        x=new int[s];  
        for (int k=0;k<x.length;k++) x[k]=k;  
    }  
  
    public String toString() {  
        String s="";  
        for (int k:i;) s=s+x[k]+" ";  
        return s;  
    }  
    ... // clone definito come prima  
}
```

# Esempio - Cloning superficiale

```
public class Test {  
    public static void main(String []a){new Test();}  
  
    Test() {  
        V p1=new V(5);  
        V p2=p1.clone();  
        System.out.println(p1);  
        System.out.println(p2);  
  
        p1.x[0]=9;  
        System.out.println(p1);  
        System.out.println(p2);  
    }  
}
```

0	1	2	3	4
0	1	2	3	4
9	1	2	3	4
9	1	2	3	4



# Esempio - Cloning profondo

```
class V implements Cloneable {
    int x[]; V(int s){...}; public String toString(){...}

    public Object clone(){
        Object tmp=null;
        try {
            tmp=super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace(); return null;
        }

        ((V)tmp).x=new int[x.length];
        for (int k:x)((V)tmp).x[k]=x[k];
        return tmp;
    }
}
```

# Esempio - Cloning profondo

```
public class Test {  
    public static void main(String []a){new Test();}  
  
    Test() {  
        V p1=new V(5);  
        V p2=p1.clone();  
        System.out.println(p1);  
        System.out.println(p2);  
  
        p1.x[0]=9;  
        System.out.println(p1);  
        System.out.println(p2);  
    }  
}
```

0	1	2	3	4
0	1	2	3	4
9	1	2	3	4
0	1	2	3	4

# Il metodo CLONE

Non tutti gli oggetti sono clonabili:

```
String a, b;  
a=new String("abcd");  
b=(String) a.clone(); // errore
```

Inoltre, non va tanto bene se **Studente** ha sottoclassi oppure è una classe composta

# Copia superficiale e profonda

La scelta di utilizzo tra copia superficiale e profonda dipende dai singoli campi, se sono mutabili o meno, e dal contesto applicativo.

- **Copia superficiale** - duplica solo i riferimenti
- **Copia profonda** - più lenta e costosa ma più sicura perché duplica gli oggetti.

# Esempio

```
public class Employee{  
    private int salary;  
    private Date hireDate;  
    private String name;  
    private Employee boss;  
}
```

- 1) **salary**, essendo un semplice intero effettuiamo una copia superficiale;
- 2) **name**, useremo una copia superficiale perché è di tipo String, anche se venisse modificato il nome, l'oggetto clonato non avrebbe ripercussioni;
- 3) **boss**, dipende dal contesto, in generale scegliamo una copia superficiale;
- 4) **hireDate**, dato che un'eventuale modifica all'oggetto originale influenzerebbe anche l'oggetto clonato effettueremo una copia profonda;

# Esempio

```
public class Employee implements Cloneable {  
    ...  
    public Employee clone() throws  
CloneNotSupportedException {  
        Employee e = (Employee) super.clone();  
        e.hireDate = (Date) hireDate.clone();  
        return e;  
    }  
}
```

# CLASSI ASTRATTE

# Problemi

Le cose viste fino ad ora possono causare dei problemi:

- Il programmatore potrebbe sbagliare il prototipo di un metodo e crearne uno nuovo invece di ridefinirlo
- Il programmatore potrebbe dimenticare di effettuare l'overriding
- Il metodo della classe base richiamata potrebbe essere solo fittizio, quindi dovremmo stampare un messaggio di errore visualizzato quando è troppo tardi
- La classe base fittizia potrebbe essere creata ed utilizzata in maniera scorretta



# Soluzione

Occorre un meccanismo per prevenire questi problemi a tempo di compilazione

Java propone l'utilizzo delle:  
**“classi astratte”**

# Classi astratte

Una classe si definisce astratta quando uno o tutti i suoi metodi non sono implementati

Essa definisce solo l'interfaccia delle classi derivate (ed eventualmente una parziale implementazione)

Essendo una classe astratta non definita completamente non è possibile istanziare un oggetto di tal tipo

Al prototipo di un metodo non corrisponde un codice che lo implementa

# Chiariamo con un esempio

```
abstract class Instrument {  
    private int i; // Storage allocated for each  
    public String what() {return "Instrument";}   
    public abstract void play(Note n);  
    public abstract void adjust();  
}
```

N.B.:

- Due metodi astratti (la loro presenza obbliga a dichiarare la classe abstract)
- what può non essere ridefinito
- La non ridefinizione degli altri genera errore di compilazione
- Si può dichiarare abstract una classe senza metodi astratti se si vuole impedire la creazione di una sua istanza
- Non cambia niente nelle classi derivate
- Un metodo astratto è per definizione polimorfo

# Considerazioni

# Un'azione pericolosa !!

La chiamata di un metodo polimorfo all'interno di un costruttore può essere pericoloso perché la costruzione dell'oggetto non è terminata

Soluzione ?

- Semplicemente evitare
- Chiamare solo i metodi finali o privati ...

# Esempio

```
abstract class Glyph {
    abstract void draw();
    Glyph() {
        System.out.println("Glyph() before draw()");
        draw();
        System.out.println("Glyph() after draw()"); }
}

class RoundGlyph extends Glyph {
    private String radius = new String();
    RoundGlyph(String r) {
        radius = r;
        System.out.println("RoundGlyph.RoundGlyph(), radius = " +
radius);
    }
    void draw() {System.out.println("RoundGlyph.draw(), radius = " +
radius);}
}

public class PolyConstructors {
    public static void main(String[] args) {new RoundGlyph(5);}
}
```

# Esempio

"Glyph() before draw()"

"RoundGlyph.draw(), radius = 0"

"Glyph() after draw()"

"RoundGlyph.RoundGlyph(), radius = 5"

# Cosa succede

1. La memoria allocata per l'oggetto è inizializzata a zero prima che succeda qualunque cosa.
2. Il costruttore della classe ereditate sono chiamati come descritto precedentemente.
  - a. A questo punto il metodo ridefinito draw( ) viene chiamato (prima che venga chiamato il costruttore di RoundGlyphv) visualizzando radius=0 per lo step 1.
3. Vengono inizializzati i membri in ordine di dichiarazione.
4. Viene chiamato il corpo del costruttore della classe derivata.



# Early e late binding

1. Linguaggio c □ usa solo early binding
2. Linguaggio c++ □ per default usa early binding, usa late binding per i metodi definiti virtual (metodi polimorfi, cioè modificati nelle classi derivate)
3. Linguaggio java □ usa solo late binding (tutti i metodi sono virtuali)
4. Quando Java usa l'early binding?
5. Es. Quando il metodo o l'attributo è dichiarato final, in caso di overload, ...
6. Il binding dinamico (o late binding) consiste nel determinare a tempo d'esecuzione, anziché a tempo di compilazione, il corpo del metodo da invocare su un dato oggetto

# Esercizio

Creare un programma, utilizzando la tecnica ad oggetti, che consente la gestione delle camere di un albergo.

- Presupporre tre diverse tipologie di camere, normale, lusso, extralusso
- Tutte le camere devono essere caratterizzate da un prezzo, uno stato (occupata/libera/luce accesa), e funzionalità base tra le quali pulisci, informazioni...
- Le camere più grandi supportano le stesse funzioni di quelle più piccole, ma hanno maggior numero di optional e operazioni (es: televisione, condizionatore...).

Presupporre che ogni stanza possa essere occupata da una lista di clienti.

- Prevedere tre tipi di clienti: adulto, bambino, ragazzo
- Per ogni oggetto di tal tipo deve essere possibile richiedere le informazioni riguardanti il cliente
- Per tutti le informazioni base sono nome e cognome
- In più per adulto e ragazzo è necessaria la carta d'identità, per il bambino età ed i nomi dei genitori.