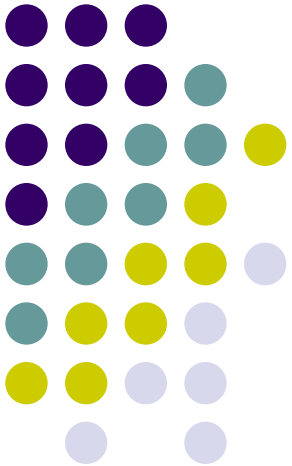
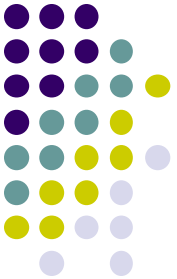
Corso di Programmazione

Polimorfismo

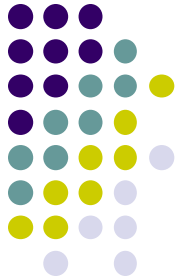


Il concetto di *Polimorfismo*



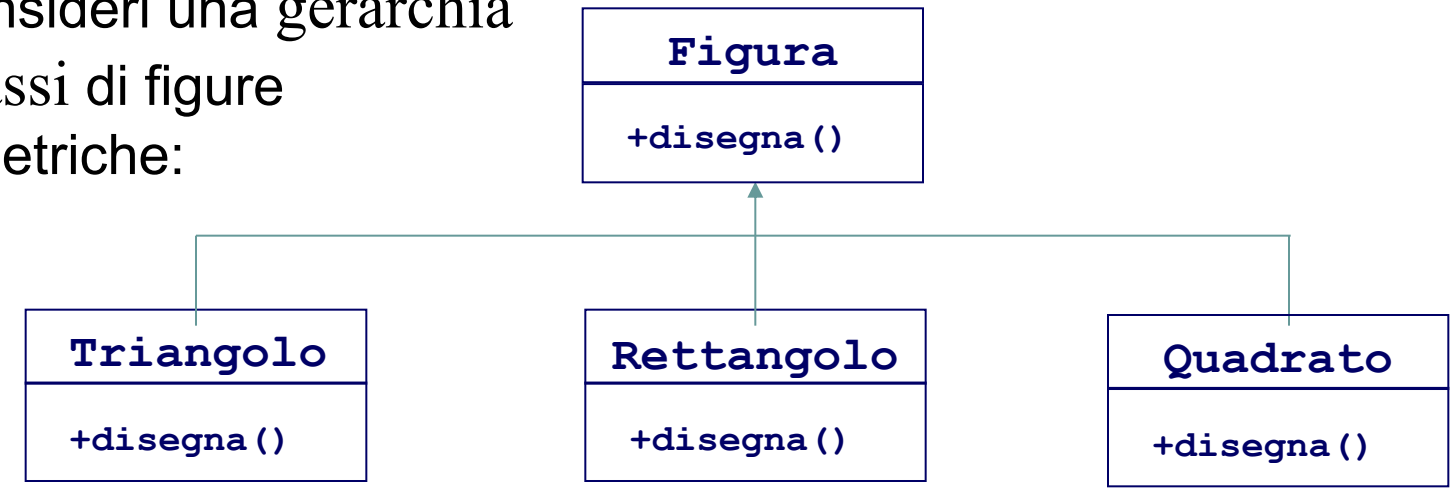
- Il **polimorfismo** consente agli oggetti di classi diverse all'interno di una gerarchia di esibire ***un comportamento diverso a tempo di esecuzione*** in risposta ad uno stesso messaggio (invocazione di una funzione) che assume forma diversa a seconda del tipo di oggetto
- Il tipo dell'oggetto su cui la funzione viene applicata cioè NON E' NOTO A TEMPO DI COMPILAZIONE
- Pertanto il legame tra la *signature* della funzione e il codice corrispondente deve essere ritardato al tempo di esecuzione (late binding) in modo da poter invocare la funzione in base al tipo dell'oggetto

Polimorfismo: esempio (1/4)



- Esempio:

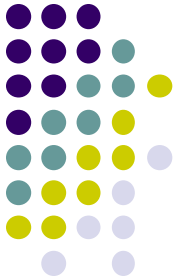
  Si consideri una gerarchia di classi di figure geometriche:



- Supponiamo di voler elaborare una sequenza A di figure, che può includere quadrati, triangoli e rettangoli.

  (ad es.: A_0 è un quadrato, A_1 un triangolo, A_2 un rettangolo, etc.)

Polimorfismo: esempio (2/4)



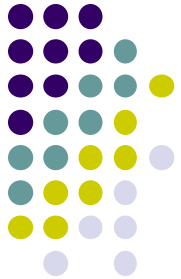
- Si consideri una funzione Disegna_Figure(), che attraverso un costrutto iterativo vuole effettuare su ciascun elemento della sequenza la stessa operazione *disegna()* (in altre parole vuole inviare a tutti gli oggetti lo stesso messaggio)
- In pseudocodice:

  for i = 1 to N do

  A_i .disegna()

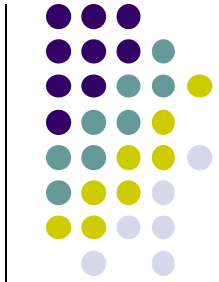
- L'esecuzione del ciclo richiede che sia possibile determinare dinamicamente (cioè a tempo d'esecuzione) l'implementazione della operazione *disegna()* da eseguire, in funzione del tipo corrente dell'oggetto A_i.

Polimorfismo: esempio (3/4)



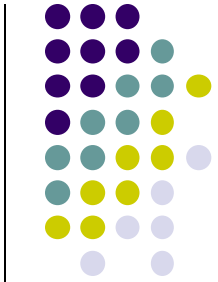
- L'istruzione A_i .disegna() non ha bisogno di essere modificata in conseguenza dell'aggiunta (o della eliminazione) di una sottoclasse di Figura (ad es.: Cerchio)
  - anche se tale sottoclasse non era stata neppure prevista all'atto della stesura della funzione Disegna_Figure()
  - Si confronti questo con l'uso di una istruzione *case* nella programmazione tradizionale

Polimorfismo: osservazioni



- Quindi il polimorfismo supporta la proprietà di **estensibilità** di un sistema software
  - minimizza la quantità di codice che l' UTENTE della gerarchia deve modificare quando si introducono nuove classi e nuove funzionalità
  - aggiungendo una classe alla gerarchia il codice UTENTE che utilizza funzionalità «generali» della gerarchia funziona….anche per la nuova classe
- Si realizza sfruttando la particolare **relazione** che sussiste **tra classe base (superclasse) e classe derivata (sottoclasse)**.

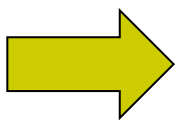
Polimorfismo in JAVA



- L'ereditarietà è una relazione «is-a»
  - Un oggetto della sottoclasse è anche un oggetto della sua superclasse
- Questa relazione consente ad un oggetto di una sottoclasse di poter prendere il posto di un oggetto della sua superclasse
- In particolare in JAVA questo significa che ***un riferimento a oggetto derivato può prendere il posto di un riferimento a un oggetto della superclasse***

Polimorfismo in JAVA: *upcasting*

```
Figura F;  //riferimento alla superclasse figura
Triangolo T = new Triangolo(…);
Rettangolo R = new Rettangolo(…);
Quadrato Q = new Quadrato(…);
```

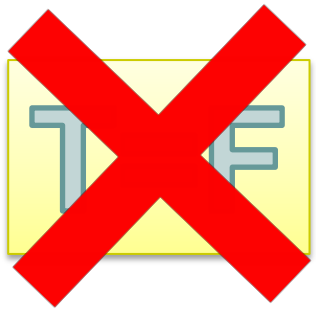


F=T

- F può puntare a qualunque degli oggetti derivati
- F = T; // F si riferisce un oggetto di tipo Triangolo
- Questo è un caso di CONVERSIONE IMPLICITA (il tipo del left value e del right value non è lo stesso)

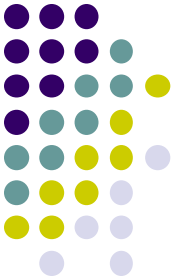
Conversione implicita
VERSO L'ALTO

Polimorfismo in JAVA: *downcasting*



- Un riferimento di tipo sottoclasse riferisce un oggetto base: downcasting
- E’ una operazione generalmente considerata **non lecita**, deve essere eventualmente forzata con una conversione esplicita

Polimorfismo in JAVA

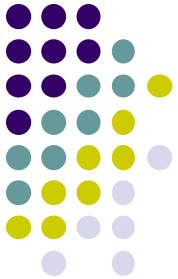


```
Figura F;  //riferimento alla superclasse figura
Triangolo T = new Triangolo(…);
Quadrato Q = new Quadrato(…);

F=T;
F.disegna();

Q.disegna();
F=Q;
F.disegna();

// sequenza di «figure»
Figura A = new Figura[N];
A[0]=T; A[1]=Q; A[2]=Q; ….

          for(int i=0; i<N; i++)
              A[i].disegna();
```

Viene chiamato il metodo disegna della classe Triangolo

Viene chiamato il metodo disegna della classe Quadrato

Esempio (1/3)



```
package Persona;

public class persona {
    private String nome;
    private String cognome;
    private int eta;

    public persona() {nome="***"; cognome="***"; eta=0;}
    public persona(String n, String c, int e) {nome=n; cognome=c; eta=e;}
    public void set_nome(String n) {nome=n;}
    public void set_cognome(String c) {cognome=c;}
    public void set_eta(int e) {eta=e;}
    public String get_nome() {return nome;}
    public String get_cognome() {return cognome;}
    public int get_eta() {return eta;}
    public String toString(){return nome + ' ' + cognome + ' ' + eta;}
}
```

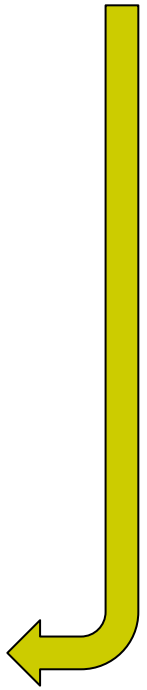
Esempio (2/3)

```
package Impiegato;
import Persona.persona;

public class impiegato extends persona {
    private String azienda;
    private int anzianita;

    public impiegato() { super(); azienda="***"; anzianita=0;}
    public impiegato(String n, String c, int e, String a, int an)
    { super(n,c,e); azienda=a; anzianita=an;}
    public void set_anzianita(int a) {anzianita=a;}
    public void set_azienda(String a) {azienda=a;}
    public String get_azienda() {return azienda;}
    public int get_anzianita() {return anzianita;}
    public String toString() {return super.toString() + ' ' + azienda + ' ' + anzianita;}
}
```

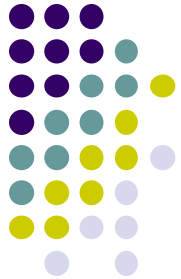
Esempio (3/3)

- Viene invocato il metodo toString() della classe base oppure quello della classe derivata in base NON al tipo del riferimento (che provocherebbe sempre la chiamata al metodo della classe persona) ma in base al tipo dell'oggetto puntato a tempo di esecuzione

```
package classe;
import java.util.Scanner;
import Persona.persona;
import Impiegato.impiegato;

public class Classe {

    public static void main(String[] args) {
        persona p1 =new persona("Mario", "Rossi", 35);
        impiegato i1 =new impiegato("luca", "Bianchi", 30, "IBM", 3);
        persona p2 =new persona("Anna", "Esposito", 15);
        impiegato i2 =new impiegato("John", "Smith", 15, "Harrods", 10);
        persona Elenco[] = new persona[4];
        Elenco[0]=p1; Elenco[1]=i1; Elenco[2]=p2; Elenco[3]=i2;
        stampa_oggetti(Elenco,4);
}
public static void stampa_oggetti(persona [] elenco,int n){
        for(int number=0; number<n; number++)
            System.out.println(elenco[number].toString() + '\n');
}
} //fine classe
```



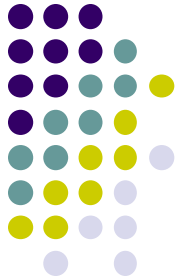
Esempio: output



Output - Classe (run)

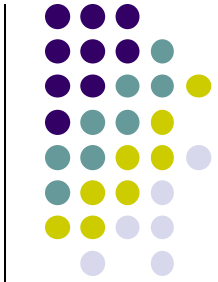
```
run:
Mario Rossi 35

luca Bianchi 30 IBM 3

Anna Esposito 15

John Smith 15 Harrods 10

BUILD SUCCESSFUL (total time: 0 seconds)
```

Early (static) e late (dynamic) binding



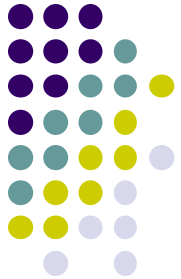
- Il concetto di binding si applica in generale a diverse entità in un linguaggio di programmazione
- Si riferisce sempre al momento in cui ad una entità viene associata una sua proprietà nello sviluppo dii un programma
  - Ad esempio il momento a cui una variabile viene associato il suo tipo, o un'area di memoria
- Early binding si riferisce in genere al ***tempo di compilazione***
- Late binding si riferisce a un tempo successivo alla compilazione, in genere al *tempo di esecuzione*

Metodi Java e binding



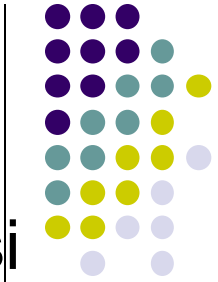
- Si parla di early o late binding anche per riferirsi al tempo in cui alla chiamata di una funzione/metodo viene ‘associato’ il relativo codice
- **Il compilatore** esamina il tipo dell'oggetto e il nome del metodo. Ottiene una lista di metodi corrispondenti a quel nome dall'oggetto.
- Successivamente, il compilatore esamina i tipi di parametro della chiamata del metodo e i metodi disponibili, e trova la corrispondenza migliore (basandosi sulla firma del metodo - overloading). Se nessuna corrispondenza può essere trovata, viene generato un errore.
- Se il metodo è **privato, statico, finale o un costruttore**, il compilatore utilizza il **binding statico** per indicare esattamente quale metodo chiamare. Nel caso degli altri metodi di istanza, è necessario utilizzare il **binding dinamico** durante l'esecuzione per determinare quale metodo chiamare.

JVM e binding dinamico



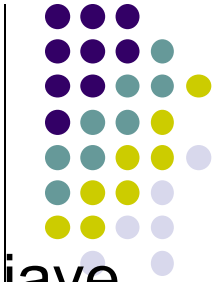
- il *binding* di tutti i metodi in Java che non siano static, final, privati o costruttori, avviene tramite *late binding*
- Il late binding consente la risoluzione dell'overriding ed è a carico della **JVM**
- A fronte della una chiamata p.m() la JVM cerca un metodo con la stessa interfaccia di m da eseguire, a partire dalla classe dell'oggetto a cui *effettivamente* punta il riferimento p a tempo di esecuzione
- se non lo si trova, si passa alla superclasse e così via, fino ad arrivare ad Object

Classe Astratta



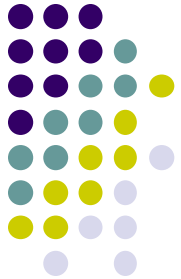
- Si possono dare dei casi in cui è utile avere classi da cui però non ha senso istanziare oggetti
  - La classe «Figura» può essere esempio di questa situazione, in effetti introduce *un «concetto»* troppo *generale* per poter essere davvero utile
- D'altra parte la classe può fornire variabili di istanza e metodi che ha senso ereditare, e ancor di più, specificare uno o più metodi che **si vuole** che le sue sottoclassi implementino
  - La classe «Figura» ad esempio potrebbe specificare un colore per il contorno e per il riempimento della forma, le setter e le getter per queste variabili, ma anche il metodo disegna, che non ha senso implementare nella classe ma si potrebbe voler rendere «obbligatorio» alle sottoclassi!

Classe Astratta in JAVA



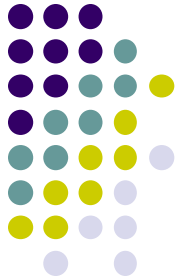
- Una classe si rende astratta in Java con la parola chiave *abstract*
- *Una classe astratta **non è** completamente implementata, cioè **contiene almeno un metodo di cui non si fornisce l'implementazione ma solo l'interfaccia***
- *I metodi non implementati devono a loro volta essere dichiarati abstract*
- *Le sottoclassi **devono** ridefinire (override) e implementare i metodi astratti a meno di non essere classi astratte a loro volta*
- *Una classe astratta quindi è una classe dalla quale si può ereditare ma non si possono istanziare oggetti, solo riferimenti…utili per il polimorfimo*

Esempio



```
package polimorfismo;

public abstract class Figura {

    private String colore_forma;
    private String colore_contorno;

    public void Figura() {
        colore_forma = "nessuno";
        colore_contorno = "nessuno";}
    public void set_coloref(String cf){colore_forma=cf; }
    public void set_colorec(String cc){colore_contorno=cc; }
    public String get_coloref(){return colore_forma;}
    public String get_colorec(){return colore_contorno;}
    public abstract void disegna();
}
```

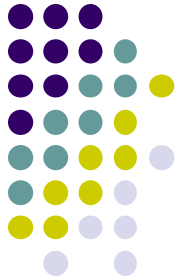
Esempio



```
public abstract class Figura2D extends Figura {
    public Figura2D() {super();}
    public abstract double area();
    public abstract double perimetro();
    public abstract void disegna();
}
```

```
public abstract class Figura3D extends Figura {
    public Figura3D() {super();}
    public abstract double superficie();
    public abstract double volume();
    public abstract void disegna();
}
```

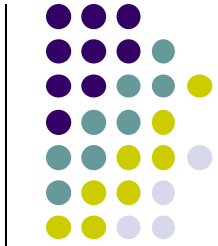
Esempio



```
public class Quadrato extends Figura2D {
    private double lato;
    public Quadrato(double l) {super(); lato=l;}
    @Override
    public double area() {return lato*lato;}
    @Override
    public double perimetro() {return lato*4;}
    @Override
    public void disegna() {System.out.println("disegno un quadrato");}
}
```

```
import static java.lang.Math.PI;

public class Sfera extends Figura3D {
    private double raggio;
    public Sfera(double r) {super(); raggio=r;}
    @Override
    public double superficie() {return 4*PI*raggio*raggio;}
    @Override
    public double volume() {return 4/3*PI*raggio*raggio*raggio;}
    @Override
    public void disegna() {System.out.println("disegno una sfera");}
}
```

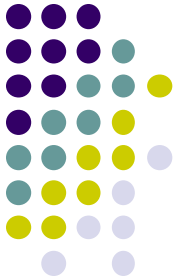
Esempio



```
public class Polimorfismo {

public static void main(String[] args) {
    int n1=4, n2=2;
    //Figura F = new Figura(); ERRORE!!!
    //Figura2D D1 = new Figura2D(); ERRORE!!!
    //Figura3D D2 = new Figura3D(); ERRORE!!!
    Figura figure[] = new Figura[n1]; //OK
    Figura2D figure2d[] = new Figura2D[n2];
    Figura3D figure3d[] = new Figura3D[n2];

    Quadrato Q1 = new Quadrato(4.0);
    Quadrato Q2 = new Quadrato(3.5);
    Sfera S1 = new Sfera(2.5);
    Sfera S2 = new Sfera(5.7);
```

```
        figure[0] = Q1;
        figure[1] = S1;
        figure[2] = Q2;
        figure[3] = S2;

        figure2d[0] = Q1;
        figure2d[1] = Q2;

        figure3d[0] = S1;
        figure3d[1] = S2;

        for(int i=0; i<n1; i++)
          figure[i].disegna();
        for(int i=0; i<n2; i++)
          System.out.println(figure2d[i].perimetro());
        for(int i=0; i<n2; i++)
          System.out.println(figure3d[i].volume());
    }
}
```

Riferimenti



- Programmare in Java:
  Capitolo 10 fino al §10.8