Polimorfismo

Polimorfismo - Esempio

- Definiamo la classe Persona che gestisce i dati anagrafici di una persona (nome ed età per semplicità)
- La classe definisce un costruttore e il metodo print che stampa a video nome ed età:

```
public class Persona
{
  protected String nome;
  protected int anni;
  public Persona(String n, int a)
  { nome=n;
    anni=a;
  }
  public void print()
  {
    System.out.print(”Mi chiamo " + nome);
    System.out.println(" e ho ” + anni + "anni");
  }
}
```

Esempio: … e studenti

- Definiamo la classe Studente, sottoclasse di Persona che ridefinisce il metodo print():

```
public class Studente extends Persona
{
  protected int matr;
  public Studente(String n, int a, int m)
  {
    super(n,a);
    matr=m;
   }
  public void print()
  {
    super.print(); // stampa nome ed età
    System.out.println("Matr = " + matr);
  }
}
```

- In questo modo print() stampa nome, età e matricola

EsempioDiCitta

- Definiamo infine la classe EsempioDiCitta che implementa il metodo statico main ed è quindi la classe principale della nostra applicazione:

```
public class EsempioDiCitta
{
  public static void main(String args[])
  {
    Persona  p = new Persona("John",45);
    Studente s = new Studente("Tom",20,156453);
    p.print();  // stampa nome ed età
    s.print();  // stampa nome, età, matricola
    p=s;        // Ok, per il subtyping
    p.print();  // COSA STAMPA ???
  }
}
```

- Cosa stampa l'ultima istruzione?

Polimorfismo

- La risposta è: stampa **nome, età e matricola!**
- p è un riferimento di tipo Persona,
- Però p punta ad un'istanza di classe Studente
- Se scriviamo p.print() viene eseguito il metodo print() ridefinito dalla classe Studente e non quello originale definito nella classe Persona
- Quindi: anche se usiamo un riferimento che ha per tipo una superclasse il fatto che l'istanza a cui il riferimento punta appartenga alla sottoclasse fa sì che il metodo invocato sia quello della sottoclasse
- Questa proprietà prende il nome di **polimorfismo (verticale)**
- Ereditarietà e polimorfismo sono i due principi che differenziano la programmazione **object-oriented** dalla programmazione **object-based**

Subtyping e polimorfismo - 1

- Subtyping e polimorfismo sono strettamente correlati
- Grazie al subtyping possiamo scrivere:
- ```
  Persona p;
  p = new Studente("Pietro",22,456327);
  ```
- Abbiamo cioè assegnato un'istanza di tipo Studente a un riferimento di tipo Persona
- Di conseguenza abbiamo potuto scrivere:

  ```
  p.print();
  ```
- In virtù del **subtyping** questa espressione è valida
- In virtù del **polimorfismo** il metodo print() che viene eseguito è quello di Studente
  ```

Riprendiamo l'esempio: Counter e CentoCounter - 2

- Sostituendo un'istanza di Counter con una di CentoCounter nel nostro esempio:

```
public class Esempio
{
   public static void main(String[] args)
   {
    int n;
    Counter c1;
    c1 = new CentoCounter();
    c1.reset();
    for (int i=0;i<150;i++)
        c1.inc();
    n = c1.getValue();
    System.out.println(n);
   }
 }
```

- Cosa stampa?

Subtyping e polimorfismo - 2

- Riassumendo:
  - **Il tipo del riferimento determina quello che si può fare:** possiamo invocare solo i metodi definiti nella classe a cui il riferimento appartiene (**subtyping**)
  - **Il tipo dell'istanza determina cosa viene effettivamente fatto:** viene invocato il metodo definito nella classe a cui l'istanza appartiene (**polimorfismo**)

- Per risolvere le chiamate ai metodi Java utilizza **late binding**
- **NOTA BENE: il codice delle chiamate è individuato dinamicamente sulla base della natura dell'istanza**

Downcasting

- Abbiamo visto che se scriviamo:
- ```
  Counter c;
  c = new BiCounter(); c.dec();
  ```
- Possiamo utilizzare i metodi definiti in Counter ma non quelli definiti in BiCounter
- Quindi non è ammesso un'istruzione come: `c.dec();`
- Se vogliamo chiamare `dec()` come possiamo fare?
- Dobbiamo ricorrere ad una conversione esplicita (typecasting)
- ```
  BiCounter b = (BiCounter)c;
  b.dec();
  ```
- Questa conversione viene chiamata **downcasting**

Sostituibilità

- Il **subtyping** in qualche modo implica che le classi derivate possono essere sostituite alle classi base in modo “sicuro”
- Da un punto di vista sintattico questo è senz’altro vero: il tipo del riferimento decide cosa possiamo fare e quindi **possiamo invocare sulla classe derivata solo i metodi che erano presenti anche nella classe base**
- Non corriamo il rischio di chiamare metodi inesistenti
- Se nella classe derivata abbiamo soltanto aggiunto metodi questo è vero in assoluto: i metodi della classe base sono quelli originali e si comportano come nella classe base
- Se invece abbiamo **ridefinito qualche metodo**, in virtù del **polimorfismo** il metodo eseguito dipende dall’istanza e non dal riferimento
- **Potremmo trovarci di fronte a qualche sorpresa!**

Riprendiamo l'esempio: Counter e CentoCounter - 2

- Sostituendo un'istanza di Counter con una di CentoCounter nel nostro esempio:

```
public class Esempio
{
   public static void main(String[] args)
   {
    int n;
    Counter c1;
    c1 = new CentoCounter();
    c1.reset();
    for (int i=0;i<150;i++)
        c1.inc();
    n = c1.getValue();
    System.out.println(n);
   }
 }
```

- Cosa stampa?

Violazioni di sostituibilità

- **Cosa è successo?**
- Ridefinendo il metodo inc() in CentoCounter abbiamo fatto saltare la sostituibilità fra Counter e CentoCounter
- **La classe derivata non è più sostituibile con la classe base**
- Il pasticcio è nato dalla combinazione di due fattori:
  - Abbiamo ridefinito un metodo (overriding)
  - Nel ridefinirlo abbiamo **ristretto** il comportamento della classe derivata
- **L'errore è stato quello di usare l'ereditarietà per restringere e non per estendere**
- Non a caso la parola chiave che Java usa per indicare i legami di ereditarietà è **extends**