Programmazione Orientata agli Oggetti in Linguaggio Java

Ereditarietà e Polimorfismo: Ereditarietà

versione 1.2

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

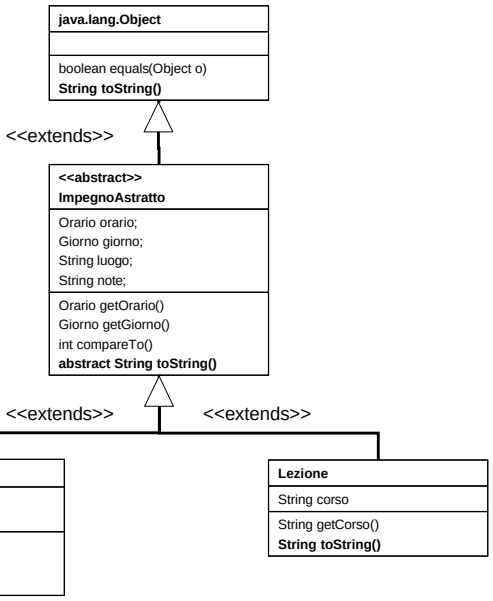
- Ereditarietà dell'Implementazione
- Sintassi
- Semantica
- Regole per la Scrittura del Codice
  - ⇨Costruzione
  - ⇨Visibilità
- Ridefinizione dei Metodi ("Overriding")

Ereditarietà dell'Implementazione

- In questa lezione
  - ⇨parliamo di uno degli aspetti dell'ereditarietà
  - ⇨l'ereditarietà delle funzionalità
  - ⇨ovvero l'ereditarietà dell'implementazione
- Ereditarietà dell'implementazione
  - ⇨fenomeno per cui gli oggetti della sottoclasse hanno, senza doverle definire, proprietà e metodi degli oggetti della superclasse

Ereditarietà dell'Implementazione

- Un esempio
  - ⇨la classe Riunione
  - ⇨estende ImpegnoAstratto
  - ⇨che estende Object
  - ⇨gli oggetti di tipo riunione ereditano tutte le proprietà e tutti i metodi di ImpegnoAstratto
  - ⇨es: è possibile chiedere ad un oggetto di tipo Riunione di eseguire il metodo getOrario()



Sintassi

- Per estendere una superclasse
  - ⇨la sottoclasse deve utilizzare la parola chiave extends
- Esempio

```
package it.unibas.appuntamenti.modello;

public class Riunione extends ImpegnoAstratto {
   private String descrizione;
   private String partecipanti;
   ...
```

Sintassi

- La parola chiave extends
  - ⇨può essere utilizzata in due modi diversi
- Ereditarietà singola
  - ⇨una classe può estenderne solo un'altra (un unico nome di classe dopo extends)
- Ereditarietà multipla
  - ⇨una classe può estenderne più di una (una lista di nomi di classe dopo extends)

Sintassi

- L'ereditarietà multipla
  - ⇨è un meccanismo potente del C++
  - ⇨ma sperimentatamente molto complesso
- Nei linguaggi moderni (Java, C#)
  - ⇨c'è il vincolo di ereditarietà singola
  - ⇨ogni classe estende un'unica classe

Sintassi

- Attenzione
  - ⇨i linguaggi moderni prevedono anche la presenza di Object
  - ⇨se una classe non specifica la clausola extends, implicitamente estende Object
- Quindi ereditarietà singola
  - ⇨vuol dire che ciascuna classe ha sempre esattamente un padre nella gerarchia
  - ⇨o definito esplicitamente oppure Object

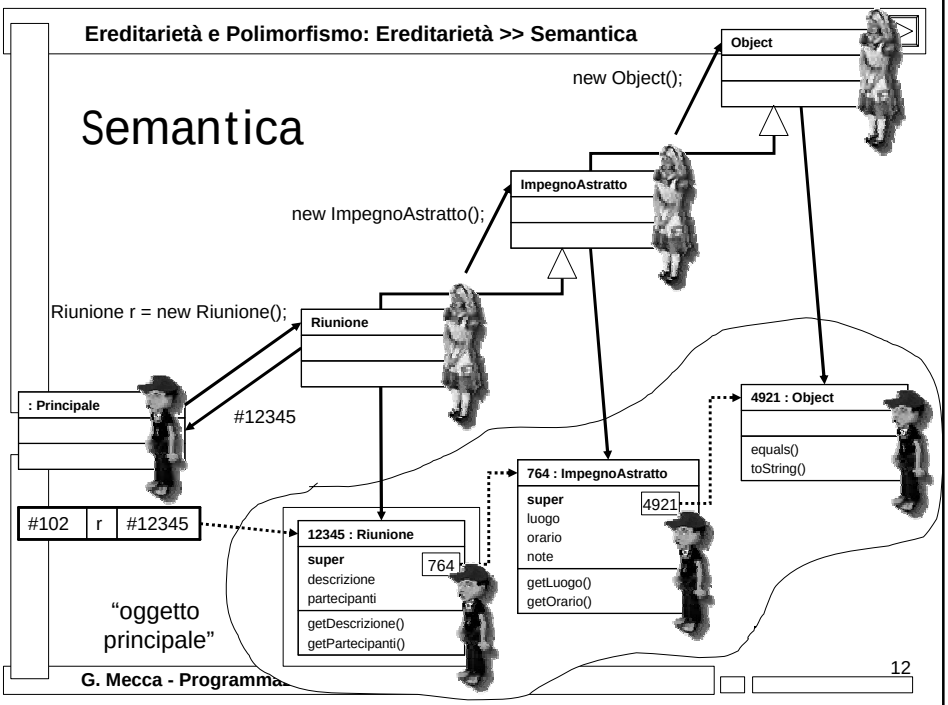
Semantica

- Il funzionamento concreto
  - ⇨è legato al meccanismo della “associazione gerarchica tra gli oggetti”
- In particolare
  - ⇨vediamo cosa succede quando viene creato un oggetto della sottoclasse
  - ⇨es: Riunione r = new Riunione();

Semantica

- Semantica
  - ⇨la creazione dell'oggetto della sottoclasse innesca automaticamente la creazione di un oggetto della sua superclasse
  - ⇨nel codice della sottoclasse è disponibile un riferimento speciale all'oggetto "padre": super
  - ⇨che può essere utilizzato per accedere alle proprietà e ai metodi dell'oggetto padre
  - ⇨il meccanismo si ripete lungo la gerarchia

Semantica



Semantica

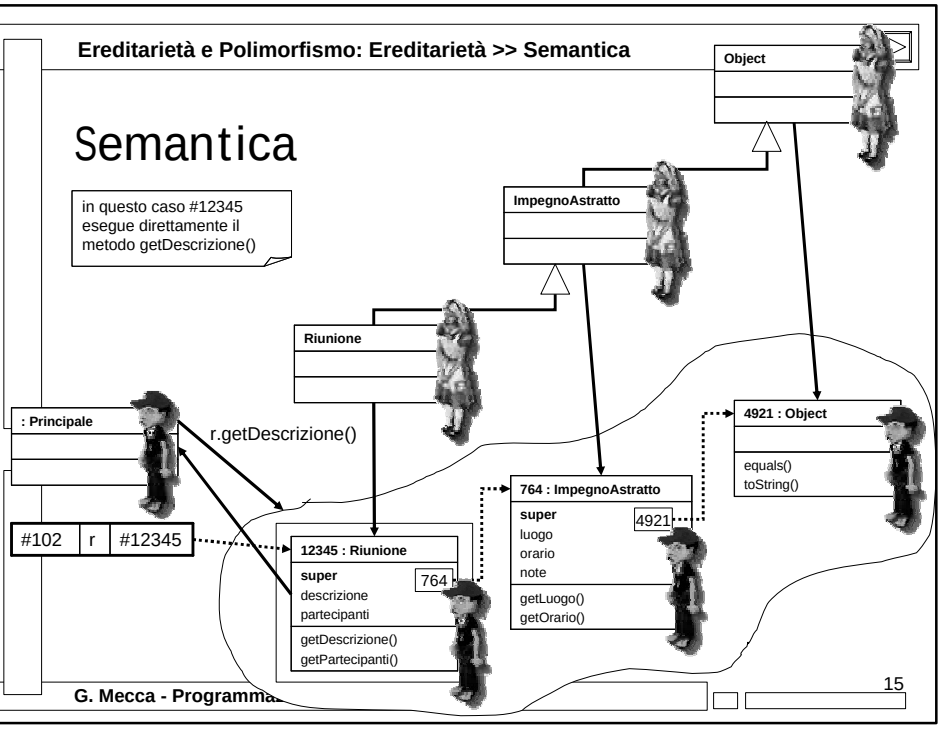
- Semantica (continua)
  - ⇨da quel momento in poi il gruppo di oggetti coopera per l'esecuzione dei metodi
  - ⇨le chiamate vengono eseguite secondo un algoritmo preciso
  - ⇨di cui per ora diamo una descrizione parziale

Semantica

- Algoritmo di esecuzione (preliminare)
  - ⇨ogni volta che viene inviato un messaggio al gruppo, l'oggetto di turno cerca il metodo tra i suoi
  - ⇨se lo trova, lo esegue
  - ⇨se non lo trova, si rivolge al suo "super" chiedendogli di eseguire il metodo a sua volta
  - ⇨il processo si ripete ricorsivamente

Semantica

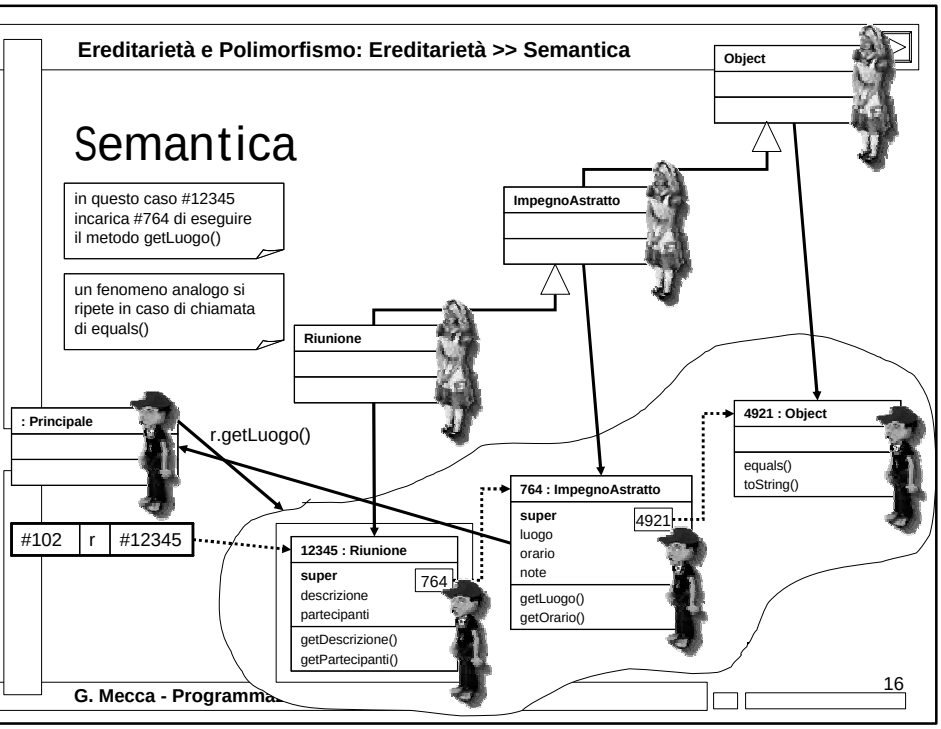
in questo caso #12345 esegue direttamente il metodo getDescrizione()



Semantica

in questo caso #12345 incarica #764 di eseguire il metodo getLuogo()

un fenomeno analogo si ripete in caso di chiamata di equals()



Semantica

- In sintesi
  - ⇨il gruppo si comporta come un "superoggetto"
  - ⇨con tutte le funzionalità degli oggetti del gruppo
  - ⇨nonostante ciascun oggetto ne implementi solo una parte

Semantica

- Vantaggi di questo approccio
  - ⇨il vantaggio principale è che ho rappresentato in modo naturale una tassonomia di concetti della realtà (es: quella degli appuntamenti)
  - ⇨il secondo vantaggio è che ho risparmiato nella scrittura del codice, evitando duplicazioni

Semantica

- Infatti
  - ⇨nel caso della classe Lezione, il procedimento si ripete esattamente allo stesso modo
  - ⇨in questo modo ho la possibilità di “sfruttare” sia in Riunione che in Lezione il codice di ImpegnoAstratto
  - ⇨primo vantaggio: velocità di scrittura
  - ⇨secondo vantaggio: facilità di manutenzione

Regole per la Scrittura del Codice

- A questo punto
  - ⇨siamo in grado di illustrare le regole imposte dal linguaggio per il funzionamento dell’ereditarietà
- Le regole
  - ⇨regole sulla costruzione degli oggetti
  - ⇨regole di visibilità

Costruzione degli Oggetti

- Regole sulla costruzione degli oggetti
  - la costruzione degli oggetti delle superclassi deve avvenire prima di quelli delle sottoclassi
  - in modo da garantire la corretta inizializzazione della proprietà super
- E' necessario distinguere vari casi
  - utilizzo di costruttori no-arg
  - utilizzo di costruttori con argomenti

Costruzione degli Oggetti

- Utilizzo di costruttori no-arg
  - prima di eseguire il costruttore no-arg della sottoclasse, la macchina virtuale esegue automaticamente una chiamata al costruttore no-arg della superclasse
  - in questo caso tutte le proprietà degli oggetti del gruppo vengono inizializzate con la regola del valore nullo
  - non è necessario l'intervento del programmatore

Costruzione degli Oggetti

- Utilizzo di costruttori con argomenti
  - ⇨in questo caso, il meccanismo è più complesso
  - ⇨tipicamente è necessario inizializzare, attraverso il costruttore della sottoclasse, anche proprietà della superclasse
  - ⇨può essere necessario invocare esplicitamente il costruttore della superclasse in quello della sottoclasse utilizzando la parola chiave super

Costruzione degli Oggetti

```
package it.unibas.appuntamenti.modello;

public class Riunione extends ImpegnoAstratto {

   private String descrizione;
   private String partecipanti;

   public Riunione() {}

   public Riunione(String descrizione, String partecipanti,
                   String luogo, String note, Orario orario) {
      super(luogo, note, orario);
      this.descrizione = descrizione;
      this.partecipanti = partecipanti;
   }
```

in questo caso il costruttore deve inizializzare anche luogo, note e orario

è necessario invocare esplicitamente il costruttore con argomenti della superclasse
NOTA: deve essere la prima istruzione

Costruzione degli Oggetti

- Riassumendo
  - è necessario che gli oggetti siano correttamente creati lungo la gerarchia e che siano inizializzati correttamente i riferimenti super
  - regola: prima di eseguire le operazioni di un costruttore della sottoclasse deve essere invocato un costruttore della superclasse
  - l'invocazione può essere fatta esplicitamente con la parola chiave super

Costruzione degli Oggetti

- Riassumendo (continua)
  - se un costruttore contiene una chiamata a super, questa deve essere la prima istruzione del corpo del costruttore
  - se il costruttore della sottoclasse non contiene una chiamata super esplicita, la macchina virtuale chiama automaticamente il costruttore no-arg della superclasse
  - se la superclasse non ha costruttore no-arg si verifica un errore a tempo di compilazione

Costruzione degli Oggetti: Esempio

```
public abstract class ImpegnoAstratto implements Impegno {
   public ImpegnoAstratto() {}
   public ImpegnoAstratto(String luogo, String note, Orario orario) { ... }


public class Riunione extends ImpegnoAstratto {
   public Riunione() {}
   public Riunione(String descrizione, String partecipanti,
                   String luogo, String note, Orario orario) {
         super(luogo, note, orario);
   ...}
```

costruttore con argomenti di Riunione

sintatticamente corretto: contiene come prima istruzione una chiamata al costruttore della superclasse

costruttore no-arg di Riunione

sintatticamente corretto: la macchina virtuale chiama automaticamente il costruttore no-arg della superclasse

Costruzione degli Oggetti: Esempio

```
public abstract class ImpegnoAstratto implements Impegno {
   public ImpegnoAstratto(String luogo, String note, Orario orario) { ... }

public class Riunione extends ImpegnoAstratto {
   public Riunione() {}
   public Riunione(String descrizione, String partecipanti,
                   String luogo, String note, Orario orario) {...}
```

costruttore con argomenti di Riunione

sintatticamente scorretto: non contiene come prima istruzione una chiamata al costruttore della superclasse; la m.v. non può chiamare il costruttore no-arg della superclasse che non esiste

costruttore no-arg di Riunione

sintatticamente scorretto: la macchina virtuale non può chiamare il costruttore no-arg della superclasse che non esiste

Regole di Visibilità

- Effetto dell'estensione (ereditarietà)
  - ⇨varie alterazioni alle regole di visibilità viste finora
- In generale
  - ⇨la macchina virtuale tende a dare l'impressione che le caratteristiche ereditate dalla superclasse siano effettivamente implementate nella sottoclasse

Regole di Visibilità

- Una prima regola
  - ⇨con la variante dell'ereditarietà attraverso un riferimento ad un oggetto della sottoclasse è possibile chiamare metodi pubblici (o "friendly") di tutta la gerarchia
- Esempio: Riunione r = new Riunione();
  - ⇨r.getDescrizione();
  - ⇨r.getLuogo();
  - ⇨r.equals(r1);

Regole di Visibilità

- Nella sottoclasse
  - ⇨è visibile tutto quello che NON è privato della superclasse
  - ⇨sono escluse le proprietà ed i metodi privati
- Un livello di visibilità intermedio
  - ⇨pensato espressamente per l'estensione
  - ⇨il livello "protected"
  - ⇨rende la proprietà o il metodo visibile nelle sottoclassi

Regole di Visibilità

```
package it.unibas.appuntamenti.modello;

public abstract class ImpegnoAstratto implements Impegno {

    protected String luogo;
    protected String note;
    protected Orario orario;
    protected Giorno giorno;

    protected ImpegnoAstratto() {}

    protected ImpegnoAstratto(String luogo, String note, Orario orario) {
        this.luogo = luogo;
        this.note = note;
        this.orario = orario;
    }
    ...
```

Regole di Visibilità

- Per inviare messaggi alla superclasse
  - ⇨è possibile utilizzare il riferimento super
- Esempio
  - ⇨il metodo toString() di Riunione

```
public String toString() {
        return (super.orario + " -- " + this.descrizione
            + " - luogo: " + super.luogo
            + " - partecipanti: " + this.partecipanti
            + " - note: " + super.note);
    }
```


Regole di Visibilità

- Abbreviazione sintattica
  - ⇨metodi e le proprietà della superclasse sono visibili nella sottoclasse in vari modi
  - ⇨attraverso il riferimento super
  - ⇨senza specificare nessun riferimento
  - ⇨attraverso il riferimento this
- Lo spirito dell'ereditarietà
  - ⇨dare l'impressione che si tratti di codice della sottoclasse

Regole di Visibilità

- Ovvero
  - ⇨entrambe le versioni seguenti del metodo toString() sono sintatticamente corrette

```
public String toString() {
   return (orario + " -- "
      + descrizione
      + " - luogo: " + luogo
      + " - partec.: " + partecipanti
      + " - note: " + note);
}
```

```
public String toString() {
   return (this.orario + " -- "
      + descrizione
      + " - luogo: " + this.luogo
      + " - partec.: " + partecipanti
      + " - note: " + this.note);
}
```


Regole di Visibilità

- Le bizzarrie di protected
  - ⇨ce ne sono varie
- I particolarità: Visibilità nella sottoclasse
  - ⇨nel codice della sottoclasse è possibile utilizzare solo proprietà e metodi protected ereditati chiamandoli con super/this
  - ⇨non è possibile chiamare metodi protected su altri riferimenti

Regole di Visibilità

- Un esempio
  - ⇨il metodo Object clone() di java.lang.Object
  - ⇨effettua una copia bit a bit nello heap dell'oggetto
  - ⇨tutte le classi lo ereditano e possono chiamare super.clone(), this.clone()
  - ⇨ma non è possibile, in una classe C scrivere

```
Integer i = new Integer();
i.clone(); // errore sintattico: clone è protected
```


Regole di Visibilità

- Riassumendo
  - ⇨cosa è visibile nella sottoclasse ?
  - ⇨proprietà e metodi public della superclasse
  - ⇨proprietà e metodi "friendly" della superclasse se le due classi sono nello stesso package
  - ⇨proprietà e metodi protected della superclasse attraverso il riferimento super o this

Regole di Visibilità

- Il particolarità: Visibilità nel package
  - ⇨il contenuto protetto è visibile in tutte le classi dello stesso package in cui è definito
  - ⇨ovvero dappertutto in java.lang posso chiamare (su qualsiasi riferimento) il metodo clone()
  - ⇨es: in java.lang.String

```
Integer i = new Integer();
i.clone(); // ok: stesso package di Object.clone()
```


Regole di Visibilità

- Di conseguenza
  - ⇨protected è stato introdotto per semplificare la scrittura delle sottoclassi
  - ⇨ma rappresenta una significativa violazione del principio di incapsulamento
  - ⇨in alcuni casi può essere opportuno evitarne l'utilizzo basandosi solo sui metodi pubblici e proprietà private

Regole di Visibilità

anche in questo caso super può essere omesso o sostituito da this

- Esempio
  - ⇨il metodo toString() di Riunione

```
public String toString() {
    return (super.getOrario() + " -- " + this.descrizione
        + " - luogo: " + super.getLuogo()
        + " - partecipanti: " + this.partecipanti
        + " - note: " + super.getNote());
}
```


Overriding

- Un fenomeno interessante
  - ⇨la ridefinizione dei metodi
- Ridefinizione dei metodi (“Overriding”)
  - ⇨i metodi delle superclassi possono essere “ridefiniti” nelle sottoclassi
  - ⇨meccanismo coerente con il fenomeno di specializzazione delle sottoclassi

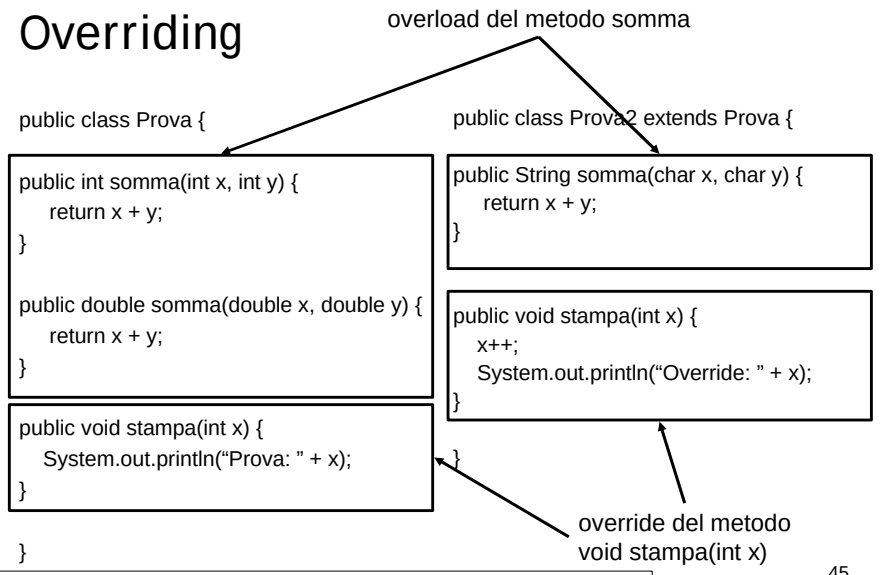
Overriding

- Esempio
  - ⇨gli oggetti della classe Riunione ereditano il metodo toString() da Object
  - ⇨ma questo viene ridefinito per renderlo più adatto alle necessità dell'applicazione
- Questo però
  - ⇨introduce delle sottigliezze nell'algoritmo di risposta alle chiamate
  - ⇨sottigliezze collegate al polimorfismo (>>)

Overriding

- Attenzione alla differenza
  - ⇨l'overriding (o riscrittura) si verifica tra due metodi con lo stesso prototipo
  - ⇨da non confondersi con l'overloading (o sovraccarico) di un metodo, che si verifica quando esistono due metodi con lo stesso nome ma parametri (e quindi prototipo diverso)

Overriding



Riassumendo

- Ereditarietà dell'Implementazione
- Sintassi
- Semantica
- Regole per la Scrittura del Codice
  - ⇨Costruzione
  - ⇨Visibilità
- Ridefinizione dei Metodi ("Overriding")

Termini della Licenza