Programmazione Orientata agli Oggetti

Approfondimenti Interface

Estensione (prima parte)

La classe `Object`

Sommario

- Estensione di Interfacce
- Estensione di Classi (prima parte)
- La classe `Object`

Sommario

- Estensione di Interfacce
- Estensione di Classi (prima parte)
- La classe `Object`

Estensione di Interface (1)

- Talvolta può essere utile definire una nuova interface a partire da una interface esistente
- Questo significa definire una nuova interface che offre qualche servizio (metodo) **aggiuntivo** rispetto ad una interface nota

Estensione di Interface (2)

- Esempio: data l'interface **A**

```
public interface A {
   public void a1(int i);
   public String a2();
}
```

- Supponiamo di dover definire l'interface **B**, che debba offrire gli stessi metodi di **A**, ed in più il metodo **b1()**

Estensione di Interface (3)

- Potremmo definire la nuova interface in questo modo

```
public interface B {
    public void a1(int i);
    public String a2();
    public int b1();
}
```

- Ma le due interface non avrebbero alcuna relazione esplicita (soprattutto, i *tipi* che definiscono non la possiedono affatto)
- Di conseguenza non potremmo referenziare con un oggetto **B** un riferimento ad **A**, anche se concettualmente sembrerebbe sensato

```
A a = new ClasseCheImplementa_A();
B b = new ClasseCheImplementa_B();
a=b; // ERRORE
```

Estensione di Interface (4)

- Sarebbe utile e sensato riuscire a specificare che **`B`** è un sottotipo di **`A`**
- In Java (e analogamente in altri linguaggi) questo è possibile definendo una interface come una *estensione* di un'altra interface
- Relativamente al nostro esempio possiamo scrivere

```
public interface B extends A {
   public int b1();
}
```

Estensione di Interface (5)

- In questo modo stiamo definendo una nuova interface (**`B`**) a partire da una già esistente (**`A`**)
- In particolare stiamo dicendo che:
  - **`B`** è un sottotipo di **`A`**
  - **`B`** offre tutti i metodi di **`A`** più il metodo **`b1()`**
- Quindi *vale* il principio di sostituzione
- In questo caso le istruzioni:

```
A a = new ClasseCheImplementa_A();
B b = new ClasseCheImplementa_B();
a = b; // OK B è un sottotipo di A
```

sono corrette

Estensione di Interface: Corrette Motivazioni

- L'estensione di interfacce *non* è una scorciatoria per non ripetere la scrittura di metodi
- E' invece un sofisticato meccanismo per definire sottotipi con effetti importanti sulla modellazione del dominio (vedi corso APS >>)
- Usare correttamente l'estensione delle interface richiede esperienza
  - Il legame tra le due interface (tipo/sottotipo) è forte e deve essere ben giustificato dal dominio

Utilizzo dell’Estensione di Interface

- In questo corso non arriveremo praticamente mai a far utilizzo dell'estensione delle interface già in fase di progettazione

- Però dobbiamo essere in grado di capirne la semantica, perché *useremo* spesso e volentieri molte interface definite per estensione di altre
  - Sono infatti frequentemente utilizzate in alcune delle API di Java approfondite in seguito, come ad es. nelle collezioni>>

Meccanismi per la Creazione di Tipi

- Attraverso l'estensione delle interfacce è possibile definire nuovi tipi a partire da tipi già esistenti
- Riassumiamo tutti i meccanismi visti sinora per introdurre nuovi tipi in Java annunciando le linee guida per il loro utilizzo
  - Esistono altri meccanismi (classi astratte, tipi enumerativi, classi nidificate) che per ragioni di natura prettamente didattica conviene rimandare

✔ Sfruttiamo invece l'occasione per introdurre la classe **`Object`**, che al contrario conviene comprendere il prima possibile

Creazione di Tipi Ex-Novo

- Le *Interfacce*
  - permettono di definire nuovi tipi senza definire l'implementazione dei metodi che formano la specifica di tipo
- Le *Classi*
  - permettono di definire nuovi tipi ma richiedono l'implementazione di tutti i metodi che formano la specifica di tipo
- Le *Classi Astratte (>>)*
  - strumento "intermedio": permette di lasciare qualche metodo astratto, ovvero senza implementazione, pur consentendone la definizione

Definizione di Nuovi Tipi per Estensione

- Estensione di interfacce
  - nuove interfacce definite come estensione di altre
  - nessun metodo nelle due interfacce possiede implementazione
- Estensione di classi
  - nuove classi definite come estensione di altre già esistenti
  - i metodi ereditano anche l'implementazione, che se necessario può essere sovrascritta (*override*)
- Per entrambe:
  - l'insieme dei metodi del tipo esteso comprende quelli pubblici del tipo base, più altri di nuova definizione

Sommario

- Estensione di Interfacce
- Estensione di Classi (prima parte)
- La classe `Object`

Estensione di Classi: Introduzione

- Uno dei meccanismi più caratteristici (e più difficili da usare *correttamente*) dei linguaggi OO è l'estensione (o ereditarietà)
- Con l'estensione possiamo definire una nuova classe a partire da una classe esistente
  - **aggiungendo** campi (variabili e/o metodi) a quelli della classe originale
  - **sovrascrivendo** metodi della classe originale

Estensione di Classi: Terminologia

- La classe di partenza viene chiamata *superclasse*, o *classe base*, o *classe genitore*
- La classe definita per estensione a partire da una classe base viene chiamata *classe estesa*, o *classe derivata*, o *sottoclasse*, o *classe figlia*
  - ✓ nell'ambito del corso preferiamo usare i termini *classe base* e *classe estesa/derivata*
- Siccome la classe derivata può a sua volta essere utilizzata come classe base di una nuova classe, si dice anche che le classi sono organizzate in una *gerarchia*

Estensione di Classi: Caratteristiche Generali

- La classe estesa conserva ("eredita") tutti i campi della classe base
- Rispetto alla classe base, la classe estesa di solito:
  - può avere qualche membro (campo e/o metodo) in aggiunta
  - può ridefinire il comportamento di qualche metodo
- La classe base viene considerata un supertipo della classe estesa
  - Quindi vale il *principio di sostituzione:* un'istanza della classe estesa può essere considerata anche come un'istanza della superclasse

Estensione di Classi: Esempio (1)

```
public class Persona {
   private String nome;

   public Persona(String nome) {
       this.nome = nome;
   }

   public void setNome(String nome) {
       this.nome = nome;
   }

   public String getNome() {
       return this.nome;
   }

   public String toString() {
       return this.getNome();
   }
}
```

Definizione di una Classe estesa

- In Java per indicare la definizione di una nuova classe per estensione di una già esistente si usa la parola chiave **`extends`**

```
class Studente extends Persona {

    // metodi e campi

}
```

Ereditarietà: Caratteristiche Generali (1)

- Tutte le variabili e tutte le operazioni definite nella classe base sono «ereditate» nella classe estesa
- Rispetto alla classe base, la classe estesa
  - può avere qualche membro (campo e/o metodo) *in più*
  - può sovrascrivere il comportamento di qualche metodo
- La classe base viene considerata un supertipo della classe estesa
  - vale il principio di sostituzione: un'istanza della classe estesa può essere considerata anche come un'istanza della superclasse

Estensione di Classi: Esempio (2)

```
public class Studente extends Persona {
   private String matricola;

   public Studente(String nome, String matricola) {
       // vediamo dopo
   }

   public void setMatricola (String matricola) {
       this.matricola = matricola;
   }

   public String getMatricola() {
       return this.matricola;
   }
  @Override
   public String toString() {
       // vediamo dopo
   }
}
```

Definizione di una Classe Estesa (1)

- La classe **`Studente`** rispetto alla classe **`Persona`** possiede un nuova variabile di istanza:
  - **`matricola`**

  ... e i corrispondenti due nuovi metodi accessori
  - **`void setMatricola(String)`**
  - **`String getMatricola()`**
- Le variabili di istanza e i metodi vengono *ereditati:*
  - le istanze della classe estesa hanno le stesse variabili della classe base *più* quelle eventualmente aggiunte
  - tutti i metodi pubblici della classe base sono disponibili nella nuova classe, senza necessità di ridefinirli
  - ✔ N.B.: esattamente come per tutte le altre classi *esterne* alla classe base, anche la classe estesa può accedere ai membri (variabili di istanza o metodi) pubblici della classe base ma *non* a quelli privati

Definizione di una Classe Estesa (2)

- I nuovi membri della classe estesa non hanno nulla di particolare
- Se si dispone di un oggetto **`Studente`** è possibile invocare i metodi della classe base **`Persona`**
  - esempio: se si dispone di un riferimento ad un oggetto **`Studente`** è possibile invocare i metodi della classe base

```
Studente anonimo = new Studente("","");
anonimo.setNome("Paolo");
```

- In generale, possiamo usare tutti i metodi pubblici della classe base (ereditati) oltre quelli della classe estesa

Estensione e Polimorfismo (1)

- Una classe estesa è un sottotipo della classe base (la classe base è un supertipo della classe estesa)
- Infatti la classe estesa offre l'interfaccia (e l'implementazione di alcuni) dei metodi della classe base
- Quindi, in base al principio di sostituzione, la classe estesa può essere usata sempre laddove è richiesto un oggetto della classe base

✔N.B.: esattamente come nel caso di una classe che implementa un’interfaccia fornendone un sottotipo *concreto*

Estensione e Polimorfismo (2)

- **`Studente`** automaticamente possiede tutti i metodi di **`Persona`**, senza bisogno di definirli
- La classe estesa ha quindi l'interfaccia e l'*implementazione* dei metodi della classe base

- **`Studente`** è un sottotipo di **`Persona`**: può essere usata al posto di **`Persona`**

Estensione di Classi (1)

- Vale il principio di sostituzione: il sottotipo può certamente essere usato al posto di un supertipo

```
public class ProvaPersona {
    public static void main(String[] args) {
        Persona p = new Studente("Paolo", "123456");
        p.setNome("Anna");
        System.out.println(p.getNome());
        Studente s = new Studente("Luigi","654321");
        s.setNome("Antonio");
    }
}
```

Estensione di Classi (2)

- Attenzione: possono essere invocati solo i metodi (pubblici) del tipo statico

```
public class RiProvaPersona {
    public static void main(String[] args) {
        Persona p = new Studente("Paolo", "123456");
        p.setNome("Anna");
        p.setMatricola("33333"); // NON COMPILA!
    }
}
```

Ereditarietà: Caratteristiche Generali (2)

- Tutte le variabili e tutte le operazioni definite nella classe base sono "*ereditate*" nella classe estesa
- Rispetto alla classe base, la classe estesa
  - ha qualche membro (campi e/o metodi) *in più*
  - può sovrascrivere il comportamento di qualche metodo

- La classe base viene considerata un supertipo della classe estesa
  - vale il principio di sostituzione: un'istanza della classe estesa può essere considerata anche come un'istanza della superclasse ed usata al suo posto

Overriding (“Riscrittura”)

- Alcune implementazioni dei metodi offerti nella classe base possono essere *non adatte* alla classe estesa
- Tipico esempio il metodo

  `String toString()`

  la stampa dovrebbe permettere di distinguere le istanze della classe base da quelle della classe estesa
  - ✓ Ad es. nella stringa restituita per gli studenti vogliamo che compaia anche la matricola (che non ha senso per tutte le persone)
- Questo comportamento si ottiene facendo l'*overriding* (*sovrascrittura*) del metodo
  - ✓ Attenzione: non confondere *overriding* ed *overloading*

Estensione di Classi: Esempio (3)

```
public class Studente extends Persona {
   private String matricola;

   public Studente(String nome, String matricola) {
      // vediamo dopo
   }

   public v                                        {
      this.
   }

   public S
      return this.matricola;
   }
   @Override
   public String toString() {
      return this.nome + " " + this.matricola;
   }
}
```

NON COMPILA: si prova ad accedere

a campi privati (la variabile di istanza **nome**)

Estensione: Overriding

- La soluzione precedente non è praticabile perché i metodi della classe estesa (**`Studente`**) non possono accedere ai campi privati della classe base (**`Persona`**)
  - anche se ogni oggetto **`Studente`** ha ereditato una variabile di istanza in cui viene memorizzata la stringa che rappresenta il nome, non è accessibile!
- Se i metodi della classe estesa vogliono accedere ai campi della classe base *devono* usare l'interfaccia pubblica della classe base come tutte le altri classe *esterne* alla stessa

Estensione di Classi: Esempio (4)

```
public class Studente extends Persona {
   private String matricola;

   public Studente(String nome, String matricola) {
      // vediamo dopo
   }

   public void setMatricola (String matricola) {
      this.matricola = matricola;
   }

   public String getMatricola() {
      return this.matricola;
   }
   @Override
   public String toString() {
      return this.getNome() + " " + this.getMatricola();
   }
}
```

Preferire sempre e comunque l'utilizzo dei metodi accessori rispetto all'uso diretto delle variabili di istanza

Accesso al metodo pubblico

Overriding e Polimorfismo (1)

- Un'istanza della classe estesa può essere usata al posto di una istanza della classe base
- Anche qui, si manifesta il polimorfismo (già visto per le interfacce) e il legame al codice avviene a tempo di esecuzione (late binding). Rispetto alle interfacce:
  - se il metodo non è ridefinito, si utilizza (si eredita) l'implementazione della superclasse
  - se il metodo è ridefinito (overriding) si utilizza l'implementazione della classe estesa
- In definitiva si sceglie *sempre* l'implementazione del tipo dinamico

Overriding e Polimorfismo (2)

Il tipo *dinamico* della variabile locale **`studente`** risulta essere **`Studente`**, all'invocazione di **`toString()`** viene eseguito il codice del corpo del metodo presente nella classe **`Studente`**

```
public class AltraProvaStudente{
    public static void main(String[] args) {

        Studente studente = new Studente("Paolo", "123456");
        Persona persona = new Studente("Anna", "654321");

        System.out.println(studente.toString());
        System.out.println(persona.toString());
    }
}
```

Il tipo *dinamico* della variabile locale **`persona`** è **`Studente`**, all'invocazione di **`toString()`** viene eseguito il codice del corpo del metodo presente nella classe **`Studente`**

Estensione: Creazione di Istanze

- Nella creazione di oggetti istanze di una classe estesa bisogna tener presente la relazione esistente con le istanze della classe padre
  - ✓ ogni istanza della classe estesa *è* anche una istanza della superclasse
- Alcuni meccanismi offerti dal linguaggio Java nella gestione dei costruttori per classi estese si comprendono meglio tenendo a mente che
  - ✓ ciascuna classe deve essere l'unica responsabile dell'inizializzazione delle proprie istanze
  - ✓ per creare una istanza di una classe estesa bisogna *prima* creare l'istanza della classe base «che è in lei»
  - ✓ bisogna concludere la creazione e l'inizializzazione di una istanza prima di fare qualsiasi altra cosa con la stessa
- ✓ A ben vedere, il servizio di creazione di `Object` (>> e derivati) può essere offerto *solo* dalla JVM

Estensione: Costruttori (1)

- Il costruttore di una classe estesa deve inizializzare:
  - *direttamente* le proprie variabili di istanza
  - *indirettamente* quelle ereditate dalla classe base
- Per il principio dell'information hiding, la classe estesa non può avere la responsabilità di inizializzare direttamente le variabili di istanza della classe base
- Per non violarlo, il costruttore della classe estesa *deve* poter delegare l'inizializzazione delle variabili di istanza della classe base ad un costruttore della stessa
  - questa operazione in Java si effettua chiamando dal corpo del costruttore della classe estesa il costruttore della classe base mediante la parola chiave **`super()`** e specificando i parametri attuali del costruttore della classe base tipicamente sulla base dei parametri formali ricevuti (>>)

Estensione: Costruttori (2)

```
public class Studente extends Persona {
   private String matricola;

   public Studente(String nome, String matricola) {
      super(nome);
      this.matricola = matricola;
   }

   public void setMatricola (String matricola) {
      this.matricola = matricola;
   }

   public String getMatricola() {
      return this.matricola;
   }

   @Override
   public String toString() {
      return this.getNome() + " " + this.getMatricola();
   }
}
```

Estensione: Costruttori (3)

- Il corpo dei costruttori di una classe estesa *deve sempre* avere una chiamata al costruttore della classe base mediante l'uso della parola chiave **`super`** come *prima* istruzione
- In assenza di una chiamata esplicita, il compilatore ne inserisce automaticamente una al costruttore *no-arg* della superclasse
  - Attenzione: *solo* in assenza di tale costruttore nella superclasse si verifica un errore a tempo di compilazione
- La chiamata al costruttore della classe base *deve* essere la *prima istruzione* nel corpo del costruttore della classe estesa

✓ Come già per i metodi, ricordiamo che anche per i costruttori è possibile definire diverse versioni sovraccariche; valgono le regole già viste per l'overloading di metodi

Estensione (Prima Parte): Ricapitoliamo

- La classe estesa:
  - ha tutte le proprietà della classe base
  - è in grado di eseguire tutti i metodi (pubblici) della classe base
  - non ha accesso ai membri privati della classe base (nessuna eccezione al principio dell'information hiding)
- Inoltre:
  - può possedere variabili di istanza proprie, oltre a quelle ereditate dalla classe base
  - può avere metodi propri
  - può specializzare il comportamento di alcuni metodi della classe base
  - può avere versioni sovraccariche del costruttore

Sommario

- Estensione di Interfacce
- Estensione di Classi (prima parte)
- La classe `Object`

La Classe `Object`

- Tutte le classi estendono (direttamente od indirettamente) la classe **`Object`**
  - si usa dire che **`Object`** è la radice della gerarchia dei tipi Java
- **`Object`** è una classe predefinita, che viene automaticamente estesa da ogni nuova classe (direttamente o indirettamente)
- La classe **`Object`** ha un insieme di metodi molto generici, ereditati e (volendo sovrascritti) da ogni nuova classe
  - tra questi metodi ce ne sono alcuni già noti (ed altri che lo saranno presto)

... extends Object

- Tutte le classi estendono automaticamente la classe **`Object`**
- E' una classe predefinita, che viene automaticamente ed implicitamente estesa da ogni nuova classe (direttamente o indirettamente)

✔ Scrivere:

```
public class MiaClasse {
}
```

è del tutto equivalente a scrivere:

```
public class MiaClasse extends Object {
}
```

Classe `Object`: Alcuni Metodi

- Vedere documentazione javadoc
  - **`String toString()`**
  - **`boolean equals(Object o)`**
  - …
- Di tutti questi metodi le nostre classi ereditano l'implementazione, oltre che la segnatura
- Le nostre classi possono ridefinirne l'implementazione (ma a tal fine *devono* rispettarne la segnatura)

Metodo `String toString()` (1)

- Questo è il motivo per il quale non è strettamente necessario definire il metodo **`toString()`** dentro le classi di nostra definizione per poterlo usare
- Se non lo definiamo, verrà comunque ereditata la definizione del metodo **`toString()`** propria della classe **`java.lang.Object`**
- Questa si preoccupa di stampare un messaggio testuale che dipende dall'indirizzo in memoria dell'oggetto sul quale viene invocato
- Di solito risulta poco esplicativo ed utile, e perciò conviene quasi sempre ridefinirlo sulla base delle specificità della classe definita

Metodo `String toString()` (2)

```
public class Persona {
   private String nome;
   …
   public void setNome(String nome){
       this.nome = nome;
   }

   public static void main(String[] args) {
       Persona p = new Persona("Paolo");
       System.out.println(p.toString());
   }
}
```

Stampa (qualcosa simile a): **`Persona@10b62c9`**

Il metodo **`toString()`** viene ereditato da **`Object`**: vale l'implementazione di **`Object`**

Metodo `String toString()` (3)

```
public class Persona {
   private String nome;

   …
   public void setNome(String nome){
        this.nome = nome;
   }
   @Override
   public String toString() {
        return this.nome;
   }

   public static void main(String[] args) {
        Persona p = new Persona();
        p.setNome("Antonio");
        System.out.println(p.toString());
   }

}
```

Stampa: **Antonio**

Il metodo **`toString()`** è stato sovrascritto: vale l'implementazione riscritta

Metodo `boolean equals(Object o)`

- Analogamente non è strettamente necessario definire il metodo **`equals()`** dentro le classi di nostra definizione per poterlo usare
- Anche in questo caso, se non lo definiamo, verrà comunque ereditata la definizione del metodo **`equals()`** propria della classe **`java.lang.Object`**
  - questa confronta l'indirizzo in memoria dell'oggetto sul quale viene invocato con il riferimento passato come parametro (analogamente all'operatore **`==`**)
  - di solito non è questa la semantica desiderata
  - definisce il criterio di equivalenza di oggetti distinti ma istanza della stessa classe (specie se le istanze saranno utilizzate dentro *collezioni*>>)

Metodo `equals()` & Testing

```
public class TestPersona {

   @Test
   public void testEquals() {
      Persona p1 = new Persona("Paolo");
      Persona p2 = new Persona("Paolo");
      assertEquals(p1, p2);
   }
}
```

FALLISCE

Il metodo **`equals()`** viene ereditato da **`Object`**:

✔ vale l'implementazione del metodo nella classe **`Object`**

Metodo `equals()`: Esempio

```
public class Persona {
   private String nome;
   …
   public void setNome(String nome){
       this.nome = nome;
   }

   @Override // overrides toString() di java.lang.Object
   public String toString() {
       return this.nome;
   }

   @Override // overrides equals(Object o) di java.lang.Object
   public boolean equals(Object o) {
       Persona that = (Persona)o; // ← (downcast)
       return this.getNome().equals(that.getNome());
   }
}
```

Il metodo **`equals(Object o)`** è stato ridefinito:

✓ il precedente test ora va a buon fine!

Metodo `equals()`: Downcast Necessario

- Attenzione alla segnatura

```
boolean equals(Object o)
```

- L'argomento è di tipo **`Object`**!
- Per questo abbiamo dovuto fare un cast (tecnicamente un *downcast:* abbiamo forzato il tipo statico al sottotipo atteso a tempo dinamico)

```
Persona that = (Persona)o;
```

- Senza questo downcast non potremmo invocare i metodi propri della classe (**`Persona`** nell'esempio) su cui si basa il confronto (l'uguaglianza degli oggetti **`String`** ottenute invocando **`getNome()`** nell'es.)
  - ✓ N.B. A sua volta **`String`** ridefinisce **`equals()`**...

Metodo `equals()`: Un Errore Troppo Frequente (1)

- Un errore è quello di usare come parametro formale un riferimento ad un oggetto della classe sulla quale si sta ridefinendo il metodo

```
boolean equals(Persona persona) // ERRORE
```

- Questa *non* è una *sovrascrittura* del metodo

```
boolean equals(Object persona)
```

- Utilizzare sempre l'annotazione **`@Override`** per ribadire al compilatore che si sta eseguendo una sovrascrittura
  - ✓ segnalerà questo tipo di problemi già in fase di compilazione ed eviterà che si debba ricercarli sulla base degli effetti (non sempre evidenti) in fase di esecuzione

Metodo `equals()`: Un Errore Troppo Frequente (2)

```
public class Persona {
    private String nome;

    …
    public void setNome(String nome){
        this.nome = nome;
    }

    public String toString() {
        return this.nome;
    }

    public boolean equals(Persona that) {
        return this.getNome().equals(that.getNome());
    }
}

public class TestPersona {

    @Test
    public void testEquals() {
        Persona p1 = new Persona("Paolo");
        Persona p2 = new Persona("Paolo");
        assertEquals(p1, p2);
    }

}
```

FALLISCE
perché viene invocato il metodo **`equals(Object o)`**: non essendo stato sovrascritto per l'errata segnatura, viene invece usato quello ereditato da `Object` che **confronta gli indirizzi in memoria degli oggetti**

Conclusioni

- Molti altri dettagli nell'estensione di classi meritano nuovamente uno spazio dedicato più avanti nel corso
- E' tuttavia utile capire già ora come tutte le nostre classi estendono e sono sottotipi di **`java.lang.Object`**, la radice della gerarchia dei tipi in Java, dal quale ereditano alcuni metodi di pubblica e generale utilità come
  - **`String toString()`**
  - **`boolean equals(Object o)`**
- ✔ Questi metodi vengono frequentemente sovrascritti nelle nostre classi per adattarli alle loro peculiarità