Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Modulo di Laboratorio di Algoritmi e Strutture Dati

Java: Introduzione – parte II

Inizializzatori statici

- Il modificatore **`static`** può anche essere utilizzato per marcare un semplice blocco di codice, che viene a sua volta ribattezzato **inizializzatore statico.**
- In pratica un blocco statico definito all'interno di una classe avrà la caratteristica di essere chiamato al momento del caricamento in memoria della classe stessa, addirittura prima di un eventuale costruttore
- Nota che non ha senso usare un costruttore per inizializzare una variabile statica

Inizializzatori statici

```
public class EsempioStatico
{
    private static int a = 10;
    public EsempioStatico()
    {
        a += 10;
    }
    static
    {
        System.out.println("valore statico = " + a);
    }
}
```

- Istanziamo con la seguente sintassi:

  `EsempioStatico ogg = new EsempioStatico();`

- Output: `valore statico = 10`

Inizializzatori statici

- Un blocco statico, come nel caso dei metodi statici, potrà utilizzare variabili definite fuori da esso se e solo se dichiarate statiche.
- È possibile inserire in una classe anche più di un inizializzatore statico:
  - in caso, questi verranno eseguiti in maniera sequenziale “dall’alto in basso”.

Inizializzatori d'istanza

- Esiste anche un'altra tipologia di inizializzatore, ma non statico.
- Si tratta dell'**inizializzatore d'istanza (instance initializer o object initializer)** e si implementa includendo codice in un blocco di parentesi graffe all'interno di una classe.
- La sua caratteristica è l'essere eseguito quando viene istanziato un oggetto, prima di un eventuale costruttore, e, solo per la prima istanza, dopo l'eventuale inizializzatore statico.

Inizializzatori d'istanza

```
public class InstanceInitializer {
    public InstanceInitializer() {
        System.out.println("Costruttore");
    }
    {
        System.out.println("Inizializzatore");
    }
}
```

- Istanziamo con la seguente sintassi:

```
InstanceInitializer ogg = new InstanceInitializer();
```

- Output:

```
Inizializzatore
Costruttore
```

Processo di creazione di un oggetto

L'interprete Java esegue **`new T()`** attraverso i seguenti passi (rif. esercitazione: Test-Initializers):

A. se è la prima volta che viene creata un'istanza della classe **`T`**:
   1. carica la classe **`T.class`** e
   2. riserva memoria ed inizializza, una volta per tutte, le variabili statiche;
B. riserva un'area di memoria per l'oggetto e la inizializza a zero;
C. esegue le inizializzazioni esplicite;
D. se è la prima volta che viene creata un'istanza della classe **`T`**: esegue tutti gli eventuali inizializzatori statici
E. Esegue tutti gli eventuali inizializzatori d'istanza
F. esegue il costruttore

Proprietà delle classi e degli oggetti (richiami)

Le proprietà principali dei linguaggi O.O. sono:

- Incapsulamento
- Ereditarietà
- Polimorfimo
- Overriding

Incapsulamento

- L'**incapsulamento** è la proprietà di rendere invisibili i dati e di gestirli solo tramite metodi
- Si nasconde l'implementazione dell'oggetto
  - Nella vita reale usiamo oggetti e macchine senza conoscere il contenuto ed il funzionamento interno
  - Il modificatore **private** impedisce l'accesso *diretto* alle componenti degli oggetti
  - L'accesso (lettura e scrittura) agli oggetti potrà essere fornito da un'interfaccia pubblica costituita da metodi dichiarati **public**, e quindi accessibili da altre classi.

Incapsulamento

Vantaggi dell'incapsulamento:

- possiamo associare azioni alla modifica delle componenti
- possiamo impedire la modifica di componenti in certe condizioni:
  - possiamo impedire l'inserimento di alcuni valori (l'età di una persona deve essere un valore positivo)
  - Possiamo decidere che, una volta costruito, l'oggetto non si può più cambiare (assenza di metodi "set")
  - Possiamo permettere l'accesso solo in certe condizioni
  - …

Ereditarietà

- A volte si incontrano classi con funzionalità simili in quanto **sottendono concetti semanticamente "vicini"**
- È possibile creare classi disgiunte replicando le porzioni di stato/comportamento condivise
  - L'approccio "Cut&Paste" non è una strategia vincente
  - Difficoltà di manutenzione correttiva e perfettiva
- Meglio "specializzare", cioè estendere e potenziare classi già esistenti

Ereditarietà

- **Superclasse**: la classe più "generale" (quella preesistente)
- **Sottoclasse o classe derivata**: la nuova classe più "specializzata"
  - la sottoclasse **eredita** dalla superclasse i membri preesistenti;
  - la sottoclasse **estende** la superclasse con i nuovi membri.
- Questo viene realizzato tramite la parola chiave **`extends`**
- La superclasse modella un concetto generico
- La sottoclasse modella un concetto più specifico:
  - Dispone di tutte le funzionalità (attributi e metodi) di quella base
  - Può aggiungere funzionalità proprie
  - Può ridefinirne il funzionamento di metodi esistenti (polimorfismo)

Ereditarietà

Vantaggi dell'ereditarietà:

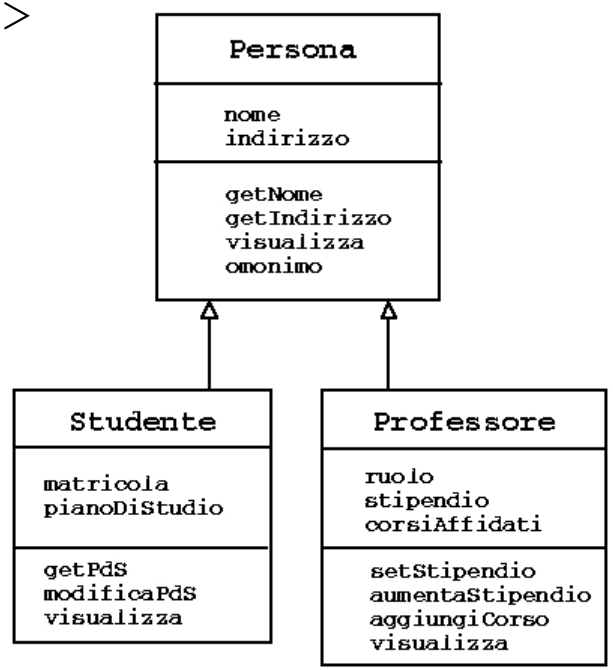
- Evitare la duplicazione di codice
- Permettere il riuso di funzionalità
- Semplificare la costruzione di nuove classi
- Facilitare la manutenzione
- Garantire la consistenza delle "interfacce"

Ereditarietà

- La sintassi per estendere una (super)classe ereditandone struttura e funzionalità è:

```
public class <sottoclasse> extends <superclasse> {
        <nuove_variabili_istanza>
        <nuovi_metodi>
}
```

- Esempio: rif. Test-Ereditarieta



Ereditarietà: un esempio

- Un'istanza di **`Studente`** avrà 4 variabili di istanza:
  - **`nome`** e **`indirizzo`**, ereditate dalla classe **`Persona`**
  - **`matricola`** e **`pianoDiStudio`**, definite nella classe **`Studente`**
- Concettualmente vuol dire che ogni istanza di **`Studente`** è anche un'istanza di **`Persona`**:
  - su di un'istanza di **`Studente`** possono essere invocati anche i metodi della superclasse **`Persona`**.

Ereditarietà: un esempio

```
Persona tizio = new Studente("Mario Rossi","L'Aquila");

   /* corretto: adesso su tizio posso invocare i
       metodi di Persona, grazie all'ereditarieta' */
    tizio.visualizza();

Studente pippo = new Studente("Pinco Pallino", "Empoli");
    ...
if ( tizio.omonimo(pippo) )
          ...
    /* corretto: posso passare pippo come parametro
       attuale, anche se era richiesta una Persona */
```

Ereditarietà

- Ogni classe definisce un tipo:
  - Un oggetto, istanza di una sottoclasse, è **formalmente compatibile** con il tipo della superclasse
  - Il contrario non è vero! (Uno studente è una persona ma una persona non è necessariamente uno studente)
  - è possibile utilizzare un'istanza di una sottoclasse (**`Studente`**) **dovunque** sia richiesto un oggetto della superclasse (**`Persona`**), come in un assegnamento o nel passaggio di parametri.
- La compatibilità diviene effettiva se i metodi ridefiniti nella sottoclasse rispettano la semantica della superclasse
- L'ereditarietà gode delle proprietà transitiva

Ereditarietà

- Il compilatore non permette di invocare un metodo di una sottoclasse su di una variabile di una superclasse

```
Persona tizio = new Studente("Mario Rossi","L'Aquila"); // Ok

tizio.modificaPdS("Algebra");  // errore a tempo di compilazione
```

- Per farlo si può usare l'operazione di **cast**

Ereditarietà

- Un **cast** consente di modificare temporaneamente la classe di riferimento di un object-id.
- Questa operazione si effettua premettendo tra parentesi tonde il nome della classe a cui si desidera promuovere l'object-id alla variabile contenente l'object-id stesso.
- Il compilatore consente di effettuare un cast solo verso classi derivate o genitrici (ma in quest'ultimo caso il cast è superfluo)

```
((Studente) tizio).modificaPdS("Algebra"); // compila senza errori
```

- Quando si valuta **`((Studente) tizio)`** se **`tizio`** non si riferisce ad un'istanza di **`Studente`**, verrà generato un **errore run-time**

Instaceof

- Per essere certi del tipo dell'istanza corrispondente all'object-id, si può usare l'operatore `instanceof` che consente di stabilire se su un'istanza è possibile effettuare un cast o meno.

```
    if (tizio instanceof Studente)
((Studente) tizio).modificaPdS("Algebra");
```

NB: in questo esempio l'espressione (`tizio instanceof Studente)` restituisce `true` (un'istanza di una classe derivata è istanza anche della classe genitrice)

Esercitazione: La classe Tempo (rif. codice)

- Esempio di esecuzione di **ProvaTempo1.java**

```
Caricamento della classe Tempo...
Separatore iniziale: -
Nuova istanza: 0-0-0
Nuova istanza: 0-0-0
22-30-5
7-50-5
Nuovo separatore: :
22:30:5
17:50:5
```

- Esempio di esecuzione di **ProvaTempo2.java**

```
0:0:0:0
0:0:0:0
22:30:5:25
17:50:5:30
```

Ereditarietà

In rif. alla classe `Tempo`:

- Abbiamo esteso la classe `Tempo` con una classe `Tempo2` che contiene anche i centesimi di secondo
- La classe `Tempo2` contiene tutti i metodi e i campi della classe `Tempo`, più i campi e i metodi definiti in essa.

```
class Tempo2 extends Tempo {
  private int centesimi;
  public void assegnaCentesimi(int cent){
      this.centesimi = cent ; }
  public int getCentesimi(){
      return centesimi ; }
}
```

Il polimorfismo

- Java mantiene traccia della classe effettiva di un dato oggetto
  - seleziona sempre il metodo più specifico…
  - …anche se la variabile che lo contiene appartiene ad una classe più generica!
- Una variabile generica può avere "molte forme"
  - contenere oggetti di **sottoclassi differenti**
  - in caso di ridefinizione, il metodo chiamato dipende dal **tipo effettivo** dell'oggetto
- Il **polimorfismo** è la proprietà di invocare metodi diversi con lo stesso nome a seconda degli oggetti coinvolti.

Il polimorfismo

- Per sfruttare questa tecnica:
  - Si definiscono, nella superclasse, metodi con implementazione generica...
  - ...sostituiti, nelle sottoclassi, da implementazioni specifiche
  - Si utilizzano variabili aventi come tipo quello della
  - superclasse
- Meccanismo estremamente potente e versatile, alla base di molti “pattern” di programmazione
- **Binding**: operazione con cui il linguaggio lega il metodo da invocare ad un’istanza. Java effettua un «binding dinamico», ovvero effettua tale operazione durante l’esecuzione del programma.

Esempio

- Nella classe `Tempo2` abbiamo definito un metodo `assegnaTempo` che assegna anche i centesimi, che ridefinisce il metodo con lo stesso nome della classe `Tempo` (overloading).

```
public int assegnaTempo(int ora,int min,int sec, int cent){
      if (ora>=0 && ora<24 && min>=0&& min<60 &&   sec>=0 &&
  sec<60 && cent>=0 && cent<60) {
            ore=ora; minuti=min; secondi=sec;
            centesimi=cent;
            return 0;
      } else return -1;
  }
            miotempo2.assegnatempo(o, m, s, c);
```

- Visualizza in `Tempo2` ridefinisce l'omonimo metodo della classe `Tempo1` (overriding)

Overriding e overloading

- Il meccanismo di overriding (sovrascrittura) è concettualmente molto diverso da quello di overloading (sovraccarico), e non deve essere confuso con esso.
- L'**overloading** consente di definire in una stessa classe più metodi aventi lo stesso nome, ma che differiscano nella firma, cioè nella sequenza dei tipi dei parametri formali. È il compilatore che determina quale dei metodi verrà invocato, in base al numero e al tipo dei parametri attuali.
- L'**overriding** consente di ridefinire un metodo in una sottoclasse: il metodo originale e quello che lo ridefinisce hanno necessariamente la stessa firma, e solo l'interprete, a tempo di esecuzione, determina quale dei due deve essere eseguito.

Super

- Se si vuole usare in una classe derivata un metodo della classe genitrice che ha lo stesso nome del metodo della classe derivata si può fare tramite la parola chiave `super`:

```
super.assegnaTempo(o, m, s);
```

- Ricorda: L'**overriding** non va confuso con il polimorfismo. Nel caso dell'overriding i metodi hanno stessa firma, ma stanno in due classi diverse!
- La variabile **super** viene usata tipicamente per accedere a metodi della superclasse che sono stati sovrascritti nella sottoclasse.

Super

- L'invocazione di un costruttore della classe genitrice può avvenire solo a patto che sia la prima istruzione di un costruttore. Il costruttore della classe `Tempo2` potrebbe essere scritto come segue:

```
Tempo2 (int ora,int minuto,int secondo,int centesimo){
        super (ora, minuto, secondo);
        this.centesimo = centesimo;
    }
```

- È buona norma, quando possibile, fornire sempre ad una classe un costruttore senza argomenti

Composizione

- La seguente classe fornisce più o meno le stesse funzionalità di `Tempo2` non sfruttando l'ereditarietà ma utilizzando una tecnica che è detta di *composizione*.

```
class Tempo3 {
   public Tempo oreMinSec = new Tempo ();
   int centesimi;

   void assegnaTempo (int ora, int minuto, int secondo,
                                                int cent) {
      oreMinSec.assegnaTempo (ora, minuto, secondo);
      centesimi = cent;
   }

   int getOra() { return oreMinSec.getOra(); }
   …
}
```

composizione

- Senza istanziare `OreMinSec` la classe `Tempo3` non funzionerebbe in quanto alla variabile `OreMinSec` non corrisponderebbe nessuna istanza (errore in fase di esecuzione non appena si invoca un metodo che gli si riferisce)
- E' quindi necessario istanziare oreMinSec

composizione

- Se due o più costruttori debbono condividere una parte consistente di codice, può tornare utile usare un **inizializzatore d'istanza**.
- L'esempio precedente potrebbe quindi essere trasformato nel modo seguente:

```
class Tempo3 {
   public Tempo oreMinSec;
   int centesimi;

   { oreMinSec = new Tempo (); } //inizializzatore d'istanza

   void assegnaTempo (int ora, int minuto, int
                              secondo, int cent) {
      oreMinSec.assegnaTempo (ora, minuto, secondo);
      centesimi = cent;
   }

  …}  …
}
```