Laboratorio di Programmazione II

Corso di Laurea in Bioinformatica

Dipartimento di Informatica - Università di Verona

Sommario

- Definizione di Classi
- Esercizi
- Ereditarieta'
- Esercizi

Definizione di Classi

Astrazione sugli oggetti

Il concetto di classe

- **Classe**: realizza l'astrazione sugli oggetti (istanze)
  - raggruppiamo oggetti simili in classi (stesse proprieta')
  - stabiliamo le operazioni che gli oggetti supportano
- Definizione di una classe in java
  - **nome**: identifica il **tipo** degli oggetti
  - **variabili di istanza**: memorizzano i dati degli oggetti
  - **metodi**: operazioni che possono essere realizzate sugli oggetti
- **modificatori**: definiscono quali campi sono visibili (possono essere letti/scritti) da oggetti esterni alla classe
- **cliente** di una classe: un utilizzatore della classe (e.g., un programma che dichiara ed usa un oggetto di una certa classe)

Classi e moduli

Classe

Una classe rappresenta un modulo (componente software con funzionalita' specifiche)

- *servizi esportati*: metodi pubblici (visibili all'esterno)
- *interfaccia*: intestazione dei metodi pubblici
- *servizi importati*: altri metodi o classi utilizzate per rappresentare gli oggetti della classe
- *struttura interna*: rappresentazione degli oggetti, implementazione dei metodi

Esempio classe in Java

Definizione di una classe in java

- Vedere file PersonaEsempio.java
- descrizione delle informazioni relative agli oggetti (e.g., nome e codiceFiscale)
- definisce un'interfaccia pubblica (e.g., i metodi get e set come getNome() e setNome(String n))
- un'implementazione nascosta: codice e dati necessari per fare funzionare i metodi (e.g., i campi nome e codiceFiscale non sono accessibili dall'esterno)
- questa realizzazione effettua side-effect (e.g., il metodo setNome modifica l'oggetto)

Uso di una classe definita

Cliente per una classe definita

- Vedere file ClientePersonaEsempio.java
- definisce una variabile di tipo PersonaEsempio (p1) inizializzata ad un nuovo oggetto (creato con new e costruttore) ed una variabile PersonaEsempio p2 inizializzata con p1.
- utilizziamo i metodi della classe per manipolare le variabili (e.g., modificare nome per la variabile p2)
- la modifica genera side-effect (agisco su p2 e modifico anche le informazioni di p1)
- Il file ClientePersonaEsempio.java deve essere salvato nella stessa directory del file PersonaEsempio.java (possiamo usare i package per modificare questo)

Accesso ai campi della classe

modificatori *public* e *private*

- *public* campo (metodo o variabile) accessibile dall'esterno della classe
- *private* campo non accessibile all'esterno
- i campi *private* sono accessibili solo da metodi della classe
- vedere il file ClientePersonaErrato.java
- Regole generali per definire l'accesso
  - metodi che definiscono funzionalita' della classe utili per i clienti sono public
  - variabili di istanza e metodi di ausilio sono private
- l'insieme dei campi *public* e' detta **interfaccia pubblica** della classe
- Una volta definita l'interfaccia pubblica posso scrivere un cliente della classe (anche se i metodi non sono implementati).

Variabili di Istanza

Variabili di Istanza in Java

- Variabili di Istanza: variabili definite all'interno della classe ma fuori da ogni altro metodo
- La dichiarazione e' simile a quella di qualsiasi altra variabile con alcune differenze:
  - puo' essere preceduta da un modifictaore
  - inizializzata esplicitamente (valori di default) o dal costruttore quando viene creato l'oggetto.
- NOTA: le variabili di istanza sono associate ad un **singolo oggetto**: due istanze di oggetti della stessa classe hanno variabili di istanza diverse.

Campo di azione e tempo di vita delle Variabili di Istanza

Campo di azione

- Tutte le variabili di istanza (a prescindere dal modificatore di accesso) sono utilizzabili da tutti i metodi della classe.
- Le variabili di istanza pubbliche sono visibili anche all'esterno, utilizzando il riferimento all'oggetto della classe e l'operatore "." (operatore di selezione).

Tempo di vita

- coincide con il tempo di vita dell'oggetto di cui fanno parte

Definizione di metodi

Metodi **non** statici

Simile a quella per metodi statici ma non compare la parola chiave *static*. Questo implica che il metodo richiede un **oggetto di invocazione**

- Intestazione:

  ```
  public tipoRisultato nomeMetodo(param.)
  ```

  - public: modificatore di accesso
  - tipoRisultato: tipo di risultato del metodo (anche void)
  - nomeMetodo: nome del metodo
  - param.: lista di parametri formali

- Blocco:

  ```
  {
     istruzioni
  }
  ```

Il parametro *this*

Parametro formale implicito

- *this* rappresenta un parametro formale implcito:
  - tutti i metodi non statici possono accedere a questo parametro (come se fosse un prametro formale qualsiasi)
  - *this* denota l'oggetto di invocazione: this e' legato al riferimento dell'oggetto di invocazione
  - in molti casi *this* puo' essere omesso (java lo inserisce automaticamente)
- *this* deve essere usato se c'e' ambiguita' sul nome delle variabili
- vedere Classe UsoThis.java

Costruttori

Uso e Definizione dei costruttori

- i costruttori permettono di inizializzare le variabili della classe (anche se le variabili sono private)
- sono metodi (non static) che:
  - hanno sempre lo stesso nome della classe
  - non hanno un tipo di ritorno (diverso da void)
- NOTA: se dichiariamo i costruttori private non possiamo costruire oggetti della classe
- vedere Classi Persona.java e ClientePersona.java
- l'istruzione *new Persona(...)* crea un oggetto di tipo Persona e mette a disposizione del chiamante un riferimento a tale oggetto. Questo riferimento puo' essere salvato in una variabile o passato ad un metodo (che ha come parametro un oggetto di tipo Persona).

Overloading di Costruttori e Costruttore Standard

Overloading di Costruttori

- sfruttando il concetto di *overloading* possiamo creare piu' costruttori per una classe (se hanno parametri differenti).
- e.g., costruttore senza argomenti e costruttore con due argomenti per Persona.java

Costruttore standard

- Se non viene definito nessun costruttore il compilatore Java crea automaticamente un *costruttore standard* che non ha parametri e lascia le variabili di istanza **non inizializzate** Vedi PersonaEsempio.java
- Se si dichiara almeno un costruttore il costruttore standard viene inibito (i.e., non generato automaticamente)
- vedi esempio Persona.java e ClientePersona.java

Esercizi Definizione Classi

Esercizi Classi

Esercizi I

- Realizzare una classe Insegnamento che abbia:
  - tre campi: nome (Stringa) aula (Stringa) numeroIscritti (intero)
  - un costruttore con un argomento che inizializza il nome dell'insegnamento

    ```
    Insegnamento(String nome)
    ```
  - tre metodi get

    ```
    String getNome()
    String getAula()
    int getNumeroIscritti()
    ```
  - due metodi set void

    ```
    setAula(String aula)
    void set NumeroIscritti(int numeroIscritti)
    ```

    Soluzione:Insegnamento.java

Esercizi Classi

Esercizi II

- Realizzare un metodo della classe Insegnamento.java che aggiunga un iscritto e restituisca il nuovo numero degli iscritti

  ```
  int aggiungiIscritto()
  ```

  Soluzione: Insegnamento.java
- verificare il corretto funzionamento della classe Insegnamento eseguendo il metodo main della classe ClienteInsegnamento.java

Esercizi Classi

Esercizi III

- Realizzare una classe Java per rappresentare informazioni relative ad un Taxi con nome TaxiInfo. Dei taxi interessano, il nome, se e' disponibile oppure no, il numero di posti e la persona che guida il taxi. Le ultime due proprietà non possono essere modificate, mentre la prima e la seconda si. I taxi inizialmente devono aver specificate tutte le informazioni. *Soluzione*: TaxiInfo.java
- Realizzare un cliente, ClienteTaxiInfo, della classe TaxiInfo. La classe ClienteTaxiInfo deve contenere un metodo statico che stampa tutte le info di un oggetto TaxiInfo (della persona stampa solo il nome). Nel metodo main vengono creati due oggetti di tipo TaxiInfo e vengono stampate le info. Poi si debbono modificare nome e disponibilita' di uno dei due e stampare di nuovo le info modificate. *Soluzione*: ClienteTaxiInfo.java

Ereditarieta'

Ereditarieta'

Definire Classi Specializzate

- **Ereditarieta'**: definire una classe che **specializza** una classe esistente
- **Specializzare una classe**: definire una classe che ha le stesse caratteristiche di una classe esistente alla quale si aggiungono altre informazioni o funzionalita'
- Non dobbiamo modificare la classe esistente ma creare una classe derivata

Example (Classe autista)

```
public class Autista extends Persona { ... }
```

- Autista e' una sottoclasse di Persona, Persona e' una superclasse di Autista

Esempio Ereditarieta'

Example (Esempio di classe derivata)

```
public class Autista extends Persona {
    private String licenza;
    public Autista(){ ... } //costruttore
    public String getLicenza(){
        return licenza;
    }
    public void setLicenza(String l){
       licenza = l;
    }
}
```

Caratteristiche delle classi derivate

Info e metodi delle classi derivate

- Una classe derivata ha accesso a tutti i metodi e variabili di istanza della classe base
- Ogni oggetto della classe derivata **e' anche un oggetto** della classe base
  - posso usare un oggetto della classe derivata ovunque posso usare un oggetto della classe base

Example (Utilizzo Classe Autista)

```
Autista a = new Autista();
a.setNome("Mario");
a.setLicenza("RM323456");
TaxiInfo t1 = new TaxiInfo("Alpha",true,3,a);
```

Costruttore per le classi derivate

Come definire un costruttore per una classe derivata

- Il costruttore della classe derivata deve poter inizializzare anche i campi della classe base (perche' quelle informazioni potrebbero essere utilizzate)
- Una classe derivata puo' utilizzare un costrutto specifico per questo: ***super*(...)**
- *super(...)* va usato come prima istruzione del costruttore della classe base

Esempio Costruttore Classe Autista

Example (Costruttore Classe Autista)

```
public Autista(String nome, String cf,
 String licenza){
    super(nome,cf);
    this.licenza = licenza;
}
```

Uso di super

Note su uso di super

- Se non inseriamo super() nel costruttore della classe derivata viene invocato di default il costruttore senza argomenti
- Se il costruttore senza argomenti della classe base non e' definito e non usiamo super abbiamo un errore a tempo di compilazione
- Se non definiamo il costruttore senza argomenti della classe derivata viene costruito un costruttore di default che chiama il costruttore senza argomenti

Compatibilita'

Compatibilita' per classi derivate

- Un oggetto della classe derivata e' anche un oggetto della classe base (ha tutte le info ed i metodi della classe base)
- Non e' vero il viceversa (potrei aver bisogno delle info e metodi aggiuntivi della classe derivata)
- la compatibilita' vale sia per assegnazioni che per passaggio di parametri

Esempio compatibilita'

Example (Esempio compatibilita')

```
Persona p1 = new Persona("Claudio",
  "VRDCLD80K16G501W");
Autista aa = new Persona("Claudio",
  "VRDCLD80K16G501W","RM123KK");
Persona pp = aa; //ok Autista
 //sottoclasse di Persona
aa = p1; //Errore
pp.getLicenza();//Errore anche se pp di
 //fatto contiene questa
 //informazione (a e' un Autista)
```

Visibilita' ai campi privati

Visibilita' campi provati per classi derivate

- La classe derivata **NON** puo' accedere ai campi/metodi privati della classe base
- Esiste un modificatore diverso *protected* che non permette l'accesso al di fuori della classe ma premette l'accesso a tutte le classi derivate.
- Nota: super(...) permette di accedere (indirettamente) a campi privati della classe base
- Nota: se non specifichiamo nessun modificatore di accesso e' come se definissimo il metodo public all'interno del package (noi non lo vederemo)

Overriding

Overriding di metodi

- **Overriding**: definire un metodo in una classe derivata che ha la stessa *signature* di un metodo della classe base.
- Quando si fa **Overriding** il metodo della classe derivata deve avere anche lo stesso tipo di ritorno.
- Il metodo della classe derivata **maschera** il metodo della classe base: se il metodo viene invocato su un oggetto della classe derivata viene eseguito il metodo che ha fatto overiding
- **Overriding** e' diverso da Overloading: in overloading la signature non e' la stessa solo il nome e' lo stesso.

Esempio Overriding I

Example (Esempio di overriding per la classe autista)

```
public class Persona {
   ...
   public void stampa(){
    System.out.println(nome + "   " + codiceFiscale);
   }
   ...
}
public class Autista extends Persona{
   ...
   public void stampa(){
    System.out.println(this.getNome() + "   " +
    this.getCF()+ " " + licenza);
   }
   ...
}
```

Esempio Overriding II

Example (Esempio di overriding per la classe autista)

```
public class TestOverriding {
   public static void main(String[] args) {
    Autista a = new Autista("Autista",
     "GNNRSS76K18G501K","RM223452");
    Persona p = new Persona("Persona",
     "RNOGTN60K18G501K");
    //utilizza stampa della classe derivata
    a.stampa();
    //utilizza stampa della classe base
    p.stampa();
...
}
```

Polimorfismo

Il concetto di polimorfismo

- L'overriding di metodi porta ad avere **polimorfismo**
- **polimorfismo**: metodi diversi con la stessa segnatura all'interno di una gerarchia di classi
- java sceglie il metodo da eseguire in base all'oggetto e non al tipo della variabile (late binding)

Esempio Polimorfismo

Example (Late Binding)

```
public class TestOverriding {
   public static void main(String[] args) {
    Autista a = new Autista("Autista",
     "GNNRSS76K18G501K","RM223452");
    Persona p = new Persona("Persona",
     "RNOGTN60K18G501K");
...
    Persona pp = a;
    pp.stampa(); //quale metodo usa ?
    //stampa di Persona o stampa di Autista ?
    //java usa stampa di Autista (late binding)
   }
}
```

Gerarchie di Classi

Gerarchie di Classi

- Una classe puo' avere molte sottoclassi, e.g., potremmo avere Studente che estende la classe Persona.
- Inoltre, una classe derivata puo' a sua volta avere sottoclassi, e.g., potremmo avere AutistaMezziPesanti che specifica la classe Autista per la guida di mezzi pesanti
- Una classe non puo' estendere piu' di una classe base

Example (Errore derivazione)

```
public class C extends A, B {...}
```

Questo restituisce errore.

La classe Object

Gerarchie di Classi

- In java tutte le classi sono sottoclassi di una classe predefinita: **Object**
- Quindi tutte le classi ereditano una serie di metodi quali: *equals, clone e toString*
- Questi metodi possono essere chiamati su qualsiasi classe perche' Object fornisce una implementazione di default
- Qualsiasi classe puo' re-implementare questi metodi per avere un comportamento piu' specifico.

Il metdo toString()

Trasformare un oggetto in una stringa

- *public String toString()*
- trasforma l'oggetto in una stringa
- se non ridefinito restituisce un codice di sistema che rappresenta l'oggetto.

Example (Uso di toString())

```
public class TestToString {
  public static void main(String[] args) {
    Persona p = new Persona("Persona",
     "RNOGTN60K18G501K");
    System.out.println(p.toString());
  }
}
```

Ridefinire il metdo toString()

Come si ridefinisce il metodo toString()

- per ridefinire il metodo toString() basta fare overising come con qualsiasi altro metodo

Example (Ridefinire toString())

```
public class Persona {
  //metodo per la stampa
  public String toString(){
   return nome + "  " + codiceFiscale;
  }
}
```

L'esecuzione di TestToString stampa: Persona RNOGTN60K18G501K

Uso di toString() in Print e Println

Uso di toString() in Print e Println

- esistono delle varianti per print e println (di PrintStream) che accettano un object (e non una String).
- queste varianti invocano il metodo toString() dell'oggetto
- L'esempio stampa due volte la stessa stringa, questo e' possibile grazie al *late binding*

```
public class TestToString {
  public static void main(String[] args) {
    Persona p = new Persona("Persona",
     "RNOGTN60K18G501K");
    System.out.println(p.toString());
    System.out.println(p);
  }
}
```

Esercizi Ereditarieta'

Esercizi Ereditarieta' I

Creare la seguente gerarchia di classi:

- C1 Insegnamento universitario (InsegnamentoUniv) deriva da Insegnamento. Specifichare il nome del dipartimento cui afferisce l'insegnamento univ. ed il numero di CFU. *Sol.*: InsegnamentoUniv.java.
- C2 Insegnamento con laboratorio (InsegnamentoLab) deriva da InsegnamentoUniv. Specificare il numero di CFU associati al laboratorio. *Sol.*: InsegnamentoLab.java
- C3 Insegnamento seminariale (InsegnamentoSem) deriva da Insegnamento. Specificare il giorno in cui si tiene il seminario (come stringa). *Sol.*: InsegnamentoSem.java

Ogni classe deve ridefinire il metodo toString() per stampare tutte le informazioni relative agli oggetti. Scrivere un metodo main per ogni classe per provare le funzionalita' di base (creazione e stampa di oggetti).

Esercizi Ereditarieta' II

Parte 1/2

Definire una classe InterfacciaInsegnamento con le seguenti 4 funzionalita' (*Sol.*:InterfacciaInsegnamento.java):

- f1 il programma chiede le informazioni comuni a tutte le tipologie di insegnamento (nome, aula e numeroIscritti). Poi legge da tastiera una stringa (di un carattere) che descrive il tipo di insegnamento: "U" per insegnamento universitario ed "S" per seminario
- f2 se l'utente immette "U" il programma chiede di specificare le informazioni relative ad un insegnamento universitario. Poi chiede se si tratta di un insegnamento con laboratorio, chiedendo all'utente di scrivere una nuova stringa, se la stringa e' "Y" chiede di specificare il numero di CFU per il laboratorio (leggendo un intero dallo scanner). Infine stampa l'oggetto creato.

Esercizi Ereditarieta' II

Parte 2/2

f3 se l'utente immette "S" il programma richiede tutte le informazioni necessarie per creare l'oggetto di interesse e poi stampa l'oggetto creato. Se l'utente immette qualsiasi altra stringa il programma chiede di nuovo l'inserimento delle informazioni richieste.

f4 al termine dell'inserimento il programma chiede se si vuole inserire un nuovo oggetto e legge una stringa, se viene inserito "Y" ripete l'inserimento di un nuovo dato, altrimenti termina con un messaggio.

NOTA: leggere tutti i valori interi con il comando Integer.parseInt(sc.nextLine()) e non sc.nextInt(). Questo perche' nextInt() non consuma il carattere di *carriage return* e potrebbe creare problemi nelle successive letture. (*sc* denota l'oggetto Scanner).