Programmazione Orientata agli Oggetti in Linguaggio Java

Ereditarietà e Polimorfismo: Conclusioni

versione 2.2

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

Sommario

- Ricapitolazione
- Gestione della Persistenza
- Eccezioni
- API di Java
  - ⇨Tokenizzazione
  - ⇨Properties
- Test e Logging

Ricapitolazione

- Ereditarietà e Polimorfismo
  - ⇨ricapitoliamo le idee fondamentali
- Perchè esistono
  - ⇨per consentire di rappresentare più fedelmente gli oggetti del mondo reale
  - ⇨relazioni di “specializzazione” nelle tassonomie e relazioni “is-a”
  - ⇨risparmiando per quanto possibile nella scrittura del codice

Ricapitolazione

- Tipi di ereditarietà
  - ⇨ereditarietà dell’implementazione >> riutilizzo del codice della superclasse
  - ⇨ereditarietà dell’interfaccia, ovvero del tipo >> sostituibilità tra oggetti
- Il secondo tipo
  - ⇨è molto importante perchè, unito al binding dinamico, consente la programmazione basata sul polimorfismo

Ricapitolazione

- In concreto
  - ⇨entrambi i meccanismi possono essere spiegati sulla base di una semantica definita
  - ⇨quella della associazione gerarchia di oggetti
- Associazione gerarchica tra oggetti
  - ⇨la creazione di un oggetto scatena sempre la creazione di altri oggetti nella gerarchia
  - ⇨gli oggetti sono in associazione (riferimenti super) e collaborano tra di loro

Ricapitolazione

- Ereditarietà dell'implementazione
  - ⇨riguarda qualsiasi classe Java (e C#)
  - ⇨ogni classe ha una superclasse, eventualmente Object
  - ⇨collaborando, l'associazione gerarchica di oggetti è in grado di eseguire tutti i metodi implementati dagli oggetti della gerarchia

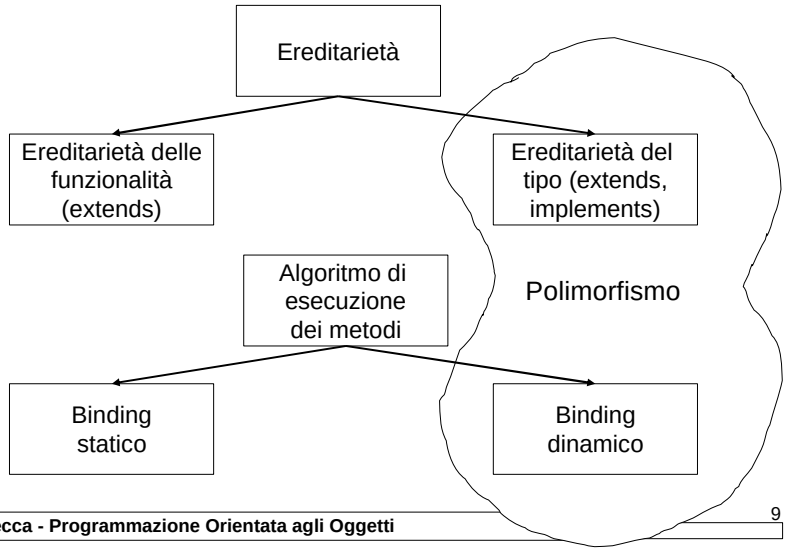
Ricapitolazione

- Ereditarietà dell'interfaccia (di tipo)
  - ⇨anche questa riguarda qualsiasi classe Java (e C#)
  - ⇨tutti gli oggetti sono anche di tipo Object
  - ⇨l'associazione gerarchica di oggetti può presentarsi ai riferimenti con "facce" diverse, ovvero come un oggetto con molti tipi
  - ⇨cambiando l'oggetto di turno
  - ⇨senza cambiare l'oggetto puntato

Ricapitolazione

- Algoritmo di esecuzione dei metodi
  - ⇨dipende dal tipo di binding
- Binding statico
  - ⇨le chiamate sono eseguite dall'oggetto di turno o dai suoi superoggetti
- Binding dinamico
  - ⇨le chiamate sono eseguite dall'oggetto puntato o dai suoi superoggetti

Ricapitolazione



Ricapitolazione

- Linee guida
  - ⇨preferire l'associazione all'estensione
  - ⇨usare l'estensione solo se le classi rappresentano oggetti in relazione is-a
  - ⇨usare l'estensione solo se è necessario l'upcast dalla sottoclasse alla superclasse
  - ⇨usare l'estensione solo quando non è possibile sostituirla con l'implementazione di interfacce

Gestione della Persistenza

- In entrambe le applicazioni
  - ⇨vengono utilizzati i file
- Gestione Apppuntamenti
  - ⇨per salvare e caricare l'agenda
- Volpi e Conigli
  - ⇨per caricare la configurazione del gioco
- Nel seguito
  - ⇨approfondiamo questi argomenti

Gestione della Persistenza

- In entrambi i casi
  - ⇨viene utilizzato un DAO per la gestione della persistenza
- DAO
  - ⇨classe dello strato di persistenza utilizzata per operazioni di caricamento e salvataggio dei dati del modello
  - ⇨tipicamente componente statico

Gestione della Persistenza

- Inoltre
  - ⇨in entrambi i casi viene utilizzato il meccanismo di trasformazione delle eccezioni
- DAOException
  - ⇨eccezione che incapsula varie eccezioni diverse che possono verificarsi nello strato di persistenza

Gestione della Persistenza

- In appuntamenti
  - ⇨DAOAgenda
  - ⇨public static Agenda carica(String nomeFile)
  - ⇨public static salva(Agenda agenda, String nomeFile)
  - ⇨l'approccio è quello usuale: caricamento, lavoro in memoria centrale, salvataggio

Gestione della Persistenza

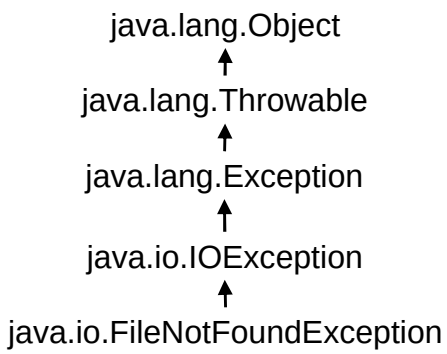
- Il metodo carica(nomeFile)
  - ⇨consiste nel creare un nuovo oggetto di tipo Agenda e riempirlo con i valori del file
  - ⇨prima del caricamento l'oggetto Agenda non esiste (e non si sa se esisterà – eccezioni)
  - ⇨viene creato e restituito dal metodo
- Il metodo salva()
  - ⇨viceversa in questo caso è necessario salvare un'agenda esistente

Eccezioni

- Nei metodi per la persistenza
  - ⇨è necessario lavorare con varie eccezioni
  - ⇨controllate e non controllate
- Eccezioni controllate
  - ⇨estendono java.lang.Exception
- Eccezioni non controllate
  - ⇨estendono java.lang.RuntimeException
- Esistono ulteriori ramificazioni

Eccezioni

- Esempio: java.io.FileNotFoundException



Eccezioni

- Attenzione all'uso della clausola catch
  - ⇨il polimorfismo interferisce con la cattura delle eccezioni
- La regola
  - ⇨un'eccezione viene catturata dal primo blocco catch il cui riferimento è compatibile con il tipo dell'eccezione
  - ⇨l'eccezione potrebbe essere compatibile con più clausole catch

Eccezioni

- Esempio: caricamento dell'agenda
- Una prima versione standard
  - ⇨crea un BufferedReader da un FileReader
  - ⇨cattura le eccezioni e le stampa sullo schermo
- Due condizioni possibili di eccezione
  - ⇨nome di file scorretto
  - ⇨malfunzionamento del disco

```
public static Agenda carica(String nomeFile) throws DAOException {
      Agenda agenda = new Agenda();
      java.io.BufferedReader flusso = null;
      try {
         java.io.FileReader fileReader = new java.io.FileReader(nomeFile);
         flusso = new java.io.BufferedReader(fileReader);
         caricaDati(agenda, flusso);
      } catch (java.io.FileNotFoundException fnfe) {
         throw new DAOException("ERRORE: file inesistente" + fnfe);
      } catch (java.io.IOException ioe) {
        throw new DAOException("ERRORE: problemi sul disco" + ioe);
      } finally {
         try {
            if (flusso != null) {
               flusso.close();
            }
         } catch (java.io.IOException ioe) {}
      }
      return agenda;
   }
```

Eccezioni

- In questo caso
  - ⇨i due blocchi catch devono essere esattamente nell'ordine specificato
  - ⇨altrimenti il secondo diventa inutile
  - ⇨l'oggetto di tipo FileNotFoundException viene intercettato dal primo blocco, perché è polimorficamente compatibile con IOException

API di Java

- Nel seguito
  - ⇨alcuni aspetti interessanti delle due applicazioni in termini di uso delle API
- In particolare
  - ⇨tokenizzazione
  - ⇨uso dei file di properties
- Nota
  - ⇨altri aspetti saranno discussi nel prossimo modulo

Tokenizzazione

- Un aspetto interessante di Agenda
  - ⇨il formato del file
  - ⇨formato libero scelto dal programmatore

```
Utente: Mario
Giorno: 25/05/04
riunione | 13:30 | riunione di prova | Università | tizio e caio | *
lezione | 15:00 | Prog. Oggetti | Aula 1 | *
-----
Giorno: 27/05/04
riunione |  8:30 | riunione n. 2 | Università | tizio | *
lezione |  9:00 | Prog. Oggetti II | Aula 1 | *
-----
Giorno: 29/05/04
riunione | 13:30 | riunione di prova | Università | tizio e caio | *
-----
```


Tokenizzazione

- Per la separazione dei giorni
  - ⇨vengono usate linee di separazione
- Per la rappresentazione degli impegni
  - ⇨viene utilizzata una riga per impegno
  - ⇨le informazioni sono separate da un carattere speciale (|)
  - ⇨questo richiede, in fase di estrazione, di effettuare una operazione di analisi delle righe

Tokenizzazione

- Analisi delle righe
  - ⇨un esempio di analisi sintattica
  - ⇨è necessario che le righe rispettino delle regole di formato
  - ⇨e quindi siano conformi ad una sintassi molto semplice
  - ⇨es: <tipo> | <ore>:<minuti> | <descrizione> | <luogo> | <partecipanti> | <note>

Tokenizzazione

- In precedenza
  - ⇨avevamo visto un altro esempio del genere
  - ⇨la lista di Record (Record: <valore>)
- In quel caso
  - ⇨l'analisi sintattica era molto semplice
- In questo caso
  - ⇨le cose sono più complesse perchè è necessaria un'operazione di "tokenizzazione"

Tokenizzazione

- Tokenizzazione
  - ⇨operazione secondo la quale, a partire da una stringa che contiene varie informazioni (token) separate da separatori uguali, vengono estratti i singoli token
- In Java
  - ⇨java.util.StringTokenizer
  - ⇨metodo String nextToken() per prelevare i token

```
private static Impegno estraiImpegno(String lineaImpegno) {
    java.util.StringTokenizer tokenizer
            = new java.util.StringTokenizer(lineaImpegno, "|");
    String tipo = tokenizer.nextToken().trim();
    Impegno impegno;
    if (tipo.equals("riunione")) {
       impegno = estraiRiunione(tokenizer);
    } else {
       impegno = estraiLezione(tokenizer);
    }
    return impegno;
}
```

```
riunione | 13:30 | riunione di prova | Università | tizio e caio | *
```

```
private static Impegno estraiRiunione(java.util.StringTokenizer tokenizer) {
    String stringaOrario = tokenizer.nextToken();
    Orario orario = estraiOrario(stringaOrario);
    String descrizione = tokenizer.nextToken().trim();
    String luogo = tokenizer.nextToken().trim();
    String partecipanti = tokenizer.nextToken().trim();
    String note = tokenizer.nextToken().trim();
    Riunione riunione = new Riunione(descrizione, partecipanti, luogo, note, orario);
    return riunione;
}
```

Tokenizzazione

- Una possibile alternativa
  - ⇨piuttosto che programmare l'analisi sintattica del file, sarebbe stato possibile usare XML
- In questo caso
  - ⇨il formato sarebbe stato diverso
  - ⇨l'analisi sintattica del documento XML sarebbe stata fatta dal parser utilizzato
  - ⇨ma il programmatore avrebbe dovuto gestire la trasformazione dal DOM al modello

Tokenizzazione

- Nella versione attuale
  - ⇨il cambiamento della tecnologia di persistenza è abbastanza semplice
  - ⇨in quanto ha effetto esclusivamente sullo strato di persistenza
  - ⇨basta cambiare l'implementazione dei metodi del DAO, senza cambiarne il prototipo
  - ⇨il DAO e DAOException isolano il controllo dal cambiamento nella persistenza

Properties

- Volpi e Conigli
  - ⇨un aspetto interessante
  - ⇨l'utilizzo di un file di configurazione esterno
- File di properties
  - ⇨formato di Java per specificare liste di coppie nome=valore da caricare rapidamente nei programmi
  - ⇨attraverso la classe java.util.Properties

Properties

```
# Volpi e Conigli – file di properties

percentualeVolpi=0.05
percentualeConigli=0.1
numeroRighe=10
numeroColonne=15
etaLimiteConigli=10
fameLimiteVolpi=5
corsaLimiteConiglio=4
```

Properties

- Per il caricamento
  - è necessario creare un oggetto di tipo java.util.Properties
  - chiedere di caricare il contenuto del file attraverso il metodo load()
  - questo riempie una mappa, da cui è possibile prelevare valori utilizzando il metodo String getProperty(String nome)

Properties

- DAOConfigurazione
  - questo compito viene svolto da un DAO opportuno
- Nota
  - Configurazione è un componente statico
  - di conseguenza DAOConfigurazione è leggermente diverso da DAOAgenda
  - in particolare, il metodo carica() è una procedura che cambia lo stato del componente statico e non costruisce nessun oggetto
  - si tratta di un esempio di modifica di componente globale che si ripercuote sugli altri componenti

```
package it.unibas.volpieconigli.persistenza;
import it.unibas.volpieconigli.modello.Configurazione;

public class DAOConfigurazione {
   private DAOConfigurazione() {}

   public static void caricaConfigurazione(String nomeFile) throws DAOException {
      try {
         java.util.Properties proprieta = caricaProperties(nomeFile);
         Configurazione.setPercentualeVolpi(
                  Double.parseDouble(proprieta.getProperty("percentualeVolpi")));
         Configurazione.setPercentualeConigli(
                  Double.parseDouble(proprieta.getProperty("percentualeConigli")));
         Configurazione.setNumeroRighe(
                  Integer.parseInt(proprieta.getProperty("numeroRighe")));
         ...
      } catch (java.io.IOException e) {
         throw new DAOException("Errore nel caricamento \n" + e);
      } catch (NumberFormatException e) {
         throw new DAOException("Errore nel nel formato del file \n" + e);
      }
   }
```


```
   private static java.util.Properties caricaProperties(String nomeFile)
                                                   throws java.io.IOException {
      java.util.Properties proprieta = new java.util.Properties();
      java.io.File file = new java.io.File(nomeFile);
      java.io.FileInputStream stream = null;
      try {
         stream = new java.io.FileInputStream(file);
         proprieta.load(stream);
      } finally {
         if (stream != null) {
            stream.close();
         }
      }
      return proprieta;
   }

}
```

Properties

- Scelta del file di properties
  - ⇨nell'applicazione il nome del file di properties viene passato come parametro al main
  - ⇨il main verifica se viene passato un parametro, e in quel caso usa il file di configurazione specificato
  - ⇨altrimenti utilizza un file di configurazione standard

```
public static void main(String[] args) {
        String nomeFile = null;
        if (args.length > 1) {
            System.out.println("Utilizzo: java Principale");
            System.out.println("  oppure: java Principale nomeFile.properties");
            return;
        } else if (args.length == 1) {
            nomeFile = args[0];
        } else if (args.length == 0) {
            nomeFile = "e:\\codice\\it\\unibas\\volpiEConigli\\config.properties";
        }
        Principale p = new Principale();
        p.esegui(nomeFile);
}
public void esegui(String nomeFile) {
        try {
            DAOConfigurazione.caricaConfigurazione(nomeFile);
            Gioco gioco = new Gioco();
            gioca(gioco);
        } catch (DAOException daoe) {
            System.out.println("Errore negli argomenti " + daoe.getMessage());
        }
}
```

Test e Logging

- Per finire
  - ⇨alcuni aspetti interessanti dei test delle due applicazioni
- Gestione appuntamenti
  - ⇨test sui metodi che gestiscono la persistenza
- Volpi e Conigli
  - ⇨test di unità e test funzionali
  - ⇨l'utilizzo di test manuali

Test e Logging

- Test sulla persistenza
  - ⇨test sui metodi di caricamento dell'agenda
  - ⇨test sui metodi di salvataggio dell'agenda
- Per verificare il caricamento
  - ⇨è necessario predisporre file di dati di test
  - ⇨caricarli
  - ⇨ed effettuare asserzioni sui dati caricati

Test e Logging >> it.unibas.appuntamenti.test.TestAgenda

- Per verificare il salvataggio
  - ⇨è necessario creare un oggetto di tipo agenda e riempirlo con impegni vari
  - ⇨salvarlo sul disco in un file
  - ⇨ricaricarlo ed effettuare asserzioni sui dati caricati

Test e Logging

- I test di Volpi e Conigli
  - ⇨due caratteristiche interessanti
- La prima caratteristica
  - ⇨effettuare test manuali sull'applicazione è praticamente impossibile
  - ⇨i test di regressione sono assolutamente indispensabili per garantire la correttezza dell'applicazione

Test e Logging

>> it.unibas.volpieconigli.modello.Gioco

- Attenzione
  - ⇨l'applicazione ha una logica applicativa molto complessa
  - ⇨per effettuare il debugging è stato necessario utilizzare estensivamente il logging
  - ⇨solo attraverso l'analisi dei messaggi di log si riesce a ricostruire il movimento degli animali sulla scacchiera

Test e Logging

>> it.unibas.volpieconigli.test.TestGioco

- La seconda caratteristica
  - ⇨i test della classe Gioco sono in realtà test funzionali dell'applicazione
  - ⇨verificano che le configurazioni successive del gioco siano corrette
  - ⇨sono un po' intricati da scrivere perchè richiedono confronti tra la configurazione precedente e quella successiva

Riassumendo

- Ricapitolazione
- Gestione della Persistenza
- Eccezioni
- API di Java
  - ⇨Tokenizzazione
  - ⇨Properties
- Test e Logging

Termini della Licenza