UNITELMA SAPIENZA

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Metodologie di Programmazione

Lezione 12: Static, consistenza ed enumerazioni

Lezione 12: Sommario

UNITELMA SAPIENZA

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- Metodi e campi statici
- Enumerazioni

Ancora sui metodi

- Il miglior modo per sviluppare e mantenere un programma grande è di costruirlo da pezzi piccoli e semplici
- Principio del divide et impera

Vantaggi dei metodi

- I metodi permettono di modularizzare un programma separandone i compiti in unità autocontenute
- Le istruzioni di un metodo non sono visibili da un altro metodo
  - Ma possono essere riutilizzate in molti punti del programma
- Tuttavia, certi metodi non utilizzano lo stato dell'oggetto
  - Ma si applicano all'intera classe

Metodi statici

- Definizione: specificando static nell'intestazione del metodo

```
public static String getLinea(int k)
{
    String s = "";
    while(k-- > 0) s += "*";
    return s;
}
```

- Accesso:
  - Dall'interno della classe, semplicemente chiamando il metodo
  - Dall'esterno, NomeClasse.nomeMetodo()

Esempi di accesso a metodi statici


```
public class LineaDiTesto
{
    private String testo;
    private int starLength;

    public LineaDiTesto(String testo, int starLength)
    {
        this.testo = testo;
        this.starLength = starLength;
    }

    public static String getLinea(int k)
    {
        String s = "";
        while(k-- > 0) s += "*";
        return s;
    }

    public String toString()
    {
        // accesso da un metodo non statico
        String starLine = getLinea(starLength);
        return starLine+testo+starLine;
    }

    public static void main(String[] args)
    {
        LineaDiTesto s = new LineaDiTesto("titolo", 5);

        // accesso da un oggetto
        s.getLinea(5);

        // accesso da un metodo statico
        getLinea(5);

        // accesso da qualunque classe
        LineaDiTesto.getLinea(5);
    }
}
```

Da un metodo non statico: **OK**

Da un riferimento a oggetto: **sconsigliato**

Da un metodo statico: **OK**

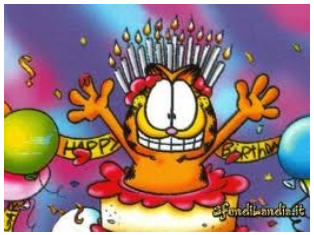
Da qualsiasi classe anteponendo il nome della classe: **OK**

Perché il metodo main() è dichiarato static?

- La Java Virtual Machine invoca il metodo main della classe specificata ancora prima di aver creato qualsiasi oggetto
- L'oggetto potrebbe non avere un costruttore senza parametri con cui creare l'oggetto

Perché non accedere direttamente ai campi?

- Perché implementiamo l'incapsulamento
- Ma anche perché garantiamo la consistenza dei dati
  - Velocità di un aeroplano
  - Date di compleanno
  - Numeri di telefono
  - Peso di un bilanciere
  - Temperatura di un forno



Esempio: l'orologio


```
public class Orologio
{
    private int ora;
    private int minuto;

    public boolean setOra(int ora)
    {
        if (ora >= 0 && ora < 24)
        {
            this.ora = ora;
            return true;
        }

        return false;
    }

    public boolean setMinuto(int minuto)
    {
        if (minuto >= 0 && minuto <= 59)
        {
            this.minuto = minuto;
            return true;
        }

        return false;
    }

    public String toString() { return ora+":"+minuto; }
}
```

Verifica se l'ora è consistente prima di modificare il campo

Idem sui minuti

Metodi get() e set()

- Tipicamente l'accesso a (alcuni) campi è garantito dai metodi getX() e setX()
  - X è tipicamente il nome del campo
- Garantiscono la consistenza dei dati
  - L'accesso pubblico al campo **NO**
- Fanno da "filtro" tra i dettagli implementativi e ciò che vede l'utente esterno
  - Ad esempio, si potrebbe utilizzare un campo che memorizza i minuti passati dall'ora 00:00
  - I metodi get() e set() nascondono questo dettaglio
- Posso sempre **CAMBIARE IDEA!**

Esempio: orologio con implementazione "criptica"

- L'utente di questa classe **non** è a conoscenza dell'implementazione "criptica" della classe:

```
public class Orologio
{
    private int oraInMinuti;
    private int minuto;

    public boolean setOra(int ora)
    {
        if (ora >= 0 && ora < 24)
        {
            oraInMinuti = ora*60;
            return true;
        }

        return false;
    }

    public boolean setMinuto(int minuto)
    {
        if (minuto >= 0 && minuto <= 59)
        {
            this.minuto = minuto;
            return true;
        }

        return false;
    }

    public int getOra() { return oraInMinuti/60; }
    public int getMinuto() { return minuto; }

    public String toString() { return getOra()+":"+getMinuto(); }
}
```

Esempio: orologio con implementazione "criptica"

- Ancora peggio!

```
public class Orologio2
{
    private int minutiDallaMezzanotte;

    public boolean setOra(int ora, int minuto)
    {
        if (ora >= 0 && ora < 24 && minuto >= 0 && minuto <= 59)
        {
            minutiDallaMezzanotte = ora*60+minuto;
            return true;
        }

        return false;
    }


    public int getOra() { return minutiDallaMezzanotte/60; }
    public int getMinuto() { return minutiDallaMezzanotte%60; }

    public String toString() { return getOra()+":"+getMinuto(); }
}
```

Campi statici

- Definizione: specificando static nell'intestazione del campo

```
public class LineaDiTesto
{
    static private char asterisco = '*';

    private String testo;
    private int starLength;

    public LineaDiTesto(String testo, int starLength)
    {
        this.testo = testo;
        this.starLength = starLength;
    }

    public static String getLinea(int k)
    {
        String s = "";
        while(k-- > 0) s += asterisco;
        return s;
    }
```

...

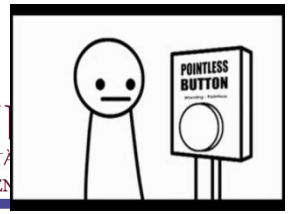
- Accesso (anal
  - Dall'interno della classe, semplicemente con l'identificatore nomeCampo
  - Dall'esterno, NomeClasse.nomeCampo

Alcuni campi statici molto noti

- Math.PI (3.141592…)
- Math.E (2.71828…)
- Dichiarati nella classe Math con modificatori public, final e static
  - public perché accessibili a tutti
  - final perché costanti
  - static perché non variano secondo lo stato dell'oggetto

- Anche le vostre costanti dovranno essere specificate in questo modo!

Importazione statica di campi



- import static permette di importare campi statici come se fossero definiti nella classe in cui si importano

```
import static java.lang.Math.E;

public class StaticImport
{
    public static void main(String[] args)
    {
        System.out.println(E);
    }
}
```

- E' possibile anche importare **TUTTI** i campi statici di una classe:

```
import static java.lang.Math.*;
public class StaticImport
{
    public static void main(String[] args)
    {
        System.out.println(E);
        System.out.println(PI);
    }
}
```

Enumerazioni

- Spesso è utile definire dei tipi (detti enumerazioni) i cui valori possono essere scelti tra un insieme predefinito di identificatori univoci
- Ogni identificatore corrisponde a una costante
- Le costanti enumerative sono implicitamente static
- Non è possibile creare un oggetto del tipo enumerato
- Un tipo enumerazione viene dichiarato mediante la sintassi:

```
public enum NomeEnumerazione
{
COSTANTE1, COSTANTE2, …, COSTANTEN
}
```

Esempio: Seme e valore di una carta


```
public enum SemeCarta
{
    CUORI,
    QUADRI,
    FIORI,
    PICCHE
}
```

```
public enum ValoreCarta
{
    ASSO,
    DUE,
    TRE,
    QUATTRO,
    CINQUE,
    SEI,
    SETTE,
    OTTO,
    NOVE,
    DIECI,
    JACK,
    DONNA,
    RE
}
```

Dichiarazione di una enumerazione

- Come tutte le classi, la dichiarazione di una enumerazione può contenere altre componenti tradizionali
  - Costruttori
  - Campi
  - Metodi

Come implementeresti una classe Mese?

```
public class Mese
{
    private int mese;

    public Mese(int mese) { this.mese = mese; }

    public int toInt() { return mese; }
    public String toString()
    {
        switch(mese)
        {
            case 1: return "GEN";
            case 2: return "FEB";
            /* ... */
            case 12: return "DIC";
            default: return null;
        }
    }
}
```

E, invece, con le enumerazioni?

UNITELMA SAPIENZA
SAPIENZA UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
public enum Mese
{
    GEN(1), FEB(2), MAR(3), APR(4), MAG(5), GIU(6), LUG(7), AGO(8), SET(9), OTT(10), NOV(11), DIC(12);

    private int mese;

    /**
     * Costruttore delle costanti enumerative
     * @param mese il mese intero
     */
    Mese(int mese) { this.mese = mese; }
    public int toInt() { return mese; }
}
```

I metodi statici values() e valueOf()

- Per ogni enumerazione, il compilatore genera il metodo statico values() che restituisce un array delle costanti enumerative
- Viene generato anche un metodo valueOf() che restituisce la costante enumerativa associata alla stringa fornita in input
- Se il valore non esiste, viene emessa un'eccezione

```
SemeCarta[] valori = SemeCarta.values();
for (int k = 0; k < valori.length; k++)
    System.out.println(valori[k]);

String v = "PICCHE";
SemeCarta picche = SemeCarta.valueOf(v);
System.out.println(picche);
```

Enumerazioni e switch

- Le enumerazioni possono essere utilizzate all'interno di un costrutto switch

```
SemeCarta seme = null;

/* ... */

switch(seme)
{
    case CUORI: System.out.println("come"); break;
    case QUADRI: System.out.println("quando"); break;
    case FIORI: System.out.println("fuori"); break;
    case PICCHE: System.out.println("piove"); break;
}
```

Esercizio: MazzoDiCarte

- Reimplementate le classi Carta e MazzoDiCarte utilizzando le enumerazioni invece degli interi per rappresentare semi e valori delle carte

Esercizio: Campo Minato

- Progettare la classe CampoMinato che realizzi il gioco del campo minato (http://it.wikipedia.org/wiki/Campo_minato_(videogioco))
- Il costruttore deve inizializzare il campo NxM (dove N e M sono interi forniti in ingresso al costruttore insieme al numero m di mine) piazzando casualmente le m mine nel campo
- Implementare un metodo scopri() che, dati x e y in ingresso, scopre la casella e restituisce un intero pari a:
  - -1 se la casella contiene una mina
  - La quantità di caselle adiacenti contenenti mine (incluse quelle in diagonale)
  - 0 se la caselle adiacenti non contengono mine. In quest'ultimo caso, vengono scoperte anche le caselle adiacenti **finché** non si incontra un numero > 0 (richiede la **ricorsione**!)
- Implementare un metodo toString() che restituisce la situazione attuale del gioco
- Implementare un metodo vinto() che restituisce lo **stato del gioco**: perso, vinto, in gioco

Esercizio: Gioco del Quindici

- Progettare la classe GiocoDelQuindici che realizzi il gioco del quindici (http://it.wikipedia.org/wiki/Gioco_del_quindici)
- Il costruttore deve inizializzare una tabellina 4x4 in cui sono posizionate casualmente 15 tessere quadrate (da 1 a 15)
- Implementare un metodo privato mischia che posiziona le caselle casualmente nella tabella (usato anche dal costruttore)
- Implementare un metodo scorri che prende in ingresso la posizione x e y della casella e la **direzione** in cui spostare la casella
- Implementare un metodo vinto che restituisce un booleano corrispondente alla vincita del giocatore (ovvero se si è riuscito a posizionare le caselle esattamente nell'ordine da 1 a 15, come riportato in alto a destra in figura)

Classi wrapper ("involucro")

- Permettono di convertire i valori di un tipo primitivo in un oggetto
- Forniscono metodi di accesso e visualizzazione dei valori

Tipo primitivo	Classe	Argomenti del costruttore
byte	Byte	byte o String
short	Short	short o String
int	Integer	int o String
long	Long	long o String
float	Float	float, double o String
double	Double	double o String
char	Character	char
boolean	Boolean	boolean o String

Confrontare oggetti interi

- Confrontavamo i valori interi primitivi mediante gli operatori di confronto ==, !=, <, <=, >, >=
- **Ma**: new Integer(5) != new Integer(5)… Perché??
- Avendo un oggetto, dobbiamo utilizzare metodi per il confronto
  - equals(): restituisce true se e solo se l'oggetto in input è un intero di valore uguale al proprio
  - compareTo(): restituisce 0 se sono uguali, < 0 se il proprio valore è < di quello in ingresso, > 0 altrimenti

Alcuni membri statici delle classi wrapper

- Integer.MIN_VALUE, Integer.MAX_VALUE
- Double.MIN_VALUE, Double.MAX_VALUE
- I metodi Integer.parseInt(), Double.parseDouble() ecc.
- Il metodo toString() fornisce una rappresentazione di tipo stringa per un tipo primitivo
- Character.isLetter(), Character.isDigit(), Character.isUpperCase(), Character.isLowerCase(), Character.toUpperCase(), ecc.

Autoboxing e auto-unboxing

- L'autoboxing converte automaticamente un tipo primitivo al suo tipo wrapper associato

```
Integer k = 3;
Integer[] array = { 5, 3, 7, 8, 9 };
```

- L'auto-unboxing converte automaticamente da un tipo wrapper all'equivalente tipo primitivo

```
int j = k;
int n = array[j];
```