



Interfacce e Polimorfismo

30/11/2004

Laboratorio di Programmazione - Luca Tesei

1

DataSet

- Riconsideriamo la classe **DataSet**
- Semplicemente si può aggiungere ad un oggetto **DataSet** dei valori con un metodo **add**
- Si può poi chiedere in qualunque momento la media dei valori o il valore massimo inserito fino a quel momento
- Questo tipo di oggetto potrebbe essere utile in generale, cioè potrebbe calcolare le stesse cose ma non solo per valori **double**, ma anche per oggetti di certe classi

30/11/2004

Laboratorio di Programmazione - Luca Tesei

2

DataSet per BankAccount

- Potremmo voler creare dei **DataSet** per oggetti della classe **BankAccount**

```
public class DataSetBankAccount {  
    ...  
    public void add(BankAccount x) {  
        sum = sum + x.getBalance();  
        if (count == 0 ||  
            maximum.getBalance() <  
                x.getBalance())  
            maximum = x;  
        count++;  
    } ...  
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

3

DataSet per BankAccount

```
...  
public BankAccount getMaximum() {  
    return x;  
}  
...  
private BankAccount maximum;  
...  
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

4

DataSet per Coin

- Potremmo anche voler implementare un **DataSet** che ci fornisca sempre lo stesso servizio, ma stavolta per la classe **Coin**:

```
public class DataSetCoin {  
    ...  
    public void add(Coin x) {  
        sum = sum + x.getValue();  
        if (count == 0 ||  
            maximum.getValue() <  
                x.getValue())  
            maximum = x;  
        count++;  
    } ...  
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

5

DataSet per Coin

```
...  
public Coin getMaximum() {  
    return x;  
}  
...  
private Coin maximum;  
...  
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

6

DataSet

- Il meccanismo usato dalle varie classi **DataSet** è lo stesso in tutti i casi: cambiano solo i dettagli
- In tutti i casi gli oggetti che vengono aggiunti al **DataSet** hanno un certo valore, nel loro stato, che viene usato per calcolare la media e il massimo
- Tutte le classi in questione potrebbero accordarsi su un unico metodo **getMeasure** che dia per ogni oggetto il valore da considerare per il **DataSet** (ogni classe decide cioè il valore da considerare "misura" dei suoi oggetti)

30/11/2004

Laboratorio di Programmazione - Luca Tesei

7

DataSet

- Lo schema del metodo **add** diventerebbe quindi:

```
sum = sum + x.getMeasure();  
if (count == 0 ||  
    maximum.getMeasure() <  
    x.getMeasure())  
    maximum = x;  
count++;
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

8

DataSet

- Ma quale dovrebbe essere il tipo della variabile **x**?
- In linea di principio **x** potrebbe essere un oggetto di una **qualunque** classe che renda disponibile un metodo **getMeasure**!
- Questo tipo di situazione si può modellare in Java tramite il meccanismo delle **interfacce**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

9

Interfacce

- Non facciamo confusione tra l'interfaccia pubblica di una classe, che abbiamo visto essere l'insieme dei suoi metodi e variabili istanza pubbliche, e un' interfaccia dichiarata in questo modo:

```
public interface Measurable {  
    double getMeasure();  
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

10

Interfacce

- Questa dichiarazione dichiara semplicemente un "**contratto**"
- Il contratto **Measurable** dice che per essere rispettato da una certa classe c'è bisogno che essa fornisca un metodo di nome **getMeasure**, senza parametri, che restituisca un **double**
- Un'interfaccia **non è una classe**: non si possono creare oggetti di tipo **Measurable**!

30/11/2004

Laboratorio di Programmazione - Luca Tesei

11

Interfacce

- Tutti i metodi di un'interfaccia non hanno l'implementazione: sono metodi astratti
- Tutti i metodi di una interfaccia, inoltre, sono automaticamente pubblici
- Una interfaccia non può avere variabili istanza
- **Però**:
- Possiamo usare un' interfaccia come un nome di classe qualsiasi nei nostri programmi

30/11/2004

Laboratorio di Programmazione - Luca Tesei

12

DataSet

```
public class DataSet {
    ...
    public void add(Measurable x) {
        sum = sum + x.getMeasure();
        if (count == 0 ||
            maximum.getMeasure() <
                x.getMeasure())
            maximum = x;
        count++;
    } ...
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

13

DataSet

```
...
public Measurable getMaximum() {
    return x;
}
...
private Measurable maximum;
...
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

14

Uso di DataSet

- Una classe come **DataSet** che utilizza il nome di una interfaccia come può essere usata?
- Si possono passare, ai metodi che richiedono parametri di tipo **Measurable**, oggetti **qualsiasi** di una classe che **implementi** (realizzi) l'interfaccia **Measurable**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

15

Implements

- La sintassi per indicare che una certa classe implementa (o realizza) una certa interfaccia è la seguente:

```
public class NomeClasse implements
    NomeInterfaccia {
    /* Solita definizione di classe in
       cui però devono apparire, con
       l'implementazione, tutti i metodi
       dichiarati nell'interfaccia, con gli
       stessi parametri nello stesso
       ordine */
    ...
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

16

Implements

- Ad esempio possiamo far implementare l'interfaccia **Measurable** alla classe **BankAccount**

```
public class BankAccount implements
    Measurable {
    ...
    public double getMeasure() {
        return balance;
    }
    ...
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

17

Implements

- Ma anche alla classe **Coin**!

```
public class Coin implements
    Measurable {
    ...
    public double getMeasure() {
        return value;
    }
    ...
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

18

Implements

- Possiamo far implementare l'interfaccia **Measurable** a tutte le classi che vogliamo
- Un oggetto della classe **DataSet** potrà ricevere oggetti qualunque di queste classi (anche mischiati)

Interfacce

- Una classe può implementare anche più di una interfaccia
- In questo caso basta elencare, separate da virgola, tutte le interfacce che implementa dopo la parola riservata **implements**
- I metodi della classe che corrispondono a quelli dell'interfaccia implementata devono obbligatoriamente essere dichiarati **public**
- Un'interfaccia va definita in un file .java che si chiama con lo stesso nome dell'interfaccia (esattamente come per le classi pubbliche)

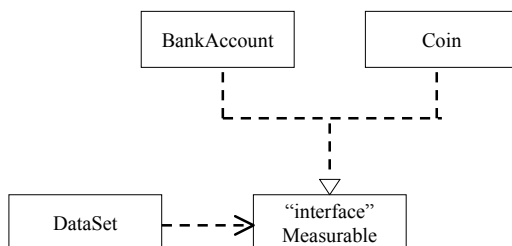
Test

- Consultare il Codice allegato **DataSetPolimorfoTest.java**
- Vediamo che variabili dichiarate di tipo **Measurable**, come ad esempio **max**, possono contenere puntatori a oggetti di classi diverse
- L'importante è che ognuna di queste classi implementi l'interfaccia **Measurable**
- I riferimenti in variabili dichiarate di tipo **Measurable** possono essere usati solo per chiamare metodi dell'interfaccia **Measurable**, anche se in realtà sull'oggetto puntato potrebbero essere chiamati anche altri metodi (ad esempio se è un oggetto di **BankAccount**)

Riduzione accoppiamento

- L'uso di interfacce può diminuire il grado di accoppiamento fra le classi di una applicazione
- Nella notazione grafica UML una interfaccia si indica con un rettangolo contenente la parola "interface" e il nome dell'interfaccia
- Una qualsiasi classe che implementa l'interfaccia è collegata ad essa da una freccia dalla linea tratteggiata e dalla punta a triangolo vuoto

Diagramma UML



Riduzione di accoppiamento

- Come si vede dal grafico la classe **DataSet** non è accoppiata né con **BankAccount** né con **Coin**
- Essa dipende solo dall'interfaccia **Measurable**
- Questo disaccoppiamento rende riutilizzabile la classe **DataSet**
- Ogni classe che implementi l'interfaccia **Measurable** può essere usata con la classe **DataSet** senza che questa ne dipenda

Conversione dei tipi e cast

- Osserviamo la riga di codice:
`bankData.add(new BankAccount(10000));`
- Passiamo il riferimento ad un oggetto di tipo **BankAccount** a un metodo che ha un parametro di tipo **Measurable**
- Ciò è lecito perché abbiamo fatto in modo che la classe **BankAccount** implementasse l'interfaccia **Measurable**
- Conversioni di questo tipo possono essere effettuate automaticamente

30/11/2004

Laboratorio di Programmazione - Luca Tesei

25

Conversione dei tipi e cast

- È lo stesso principio di quando assegnamo un **int** a un **double**: non c'è perdita di informazione e quindi l'assegnamento può essere fatto senza problemi
- Qui l'effetto della conversione consiste nel mascherare alcune potenzialità dell'oggetto facendo in modo che possano essere utilizzate solo le funzionalità che sono specificate nell'interfaccia

30/11/2004

Laboratorio di Programmazione - Luca Tesei

26

Conversione dei tipi e cast

- Crea un oggetto della classe **BankAccount**:
`BankAccount account = new BankAccount(10000);`
- Conversione legittima perché **BankAccount** implementa **Measurable**:
`Measurable x = account;`
- Su **x** però posso chiamare solo i metodi dichiarati in **Measurable**
`x.deposit(100); // Errore!!`

30/11/2004

Laboratorio di Programmazione - Luca Tesei

27

Conversione dei tipi e cast

- ```
System.out.println(x.getMeasure());
```
- Legittimo perché **x** è di tipo **Measurable**
  - Comunque l'oggetto puntato da **x** è ancora un oggetto **BankAccount** (**x == account**):  
`account.deposit(100); // Ok`
  - Posso creare anche un oggetto della classe **Coin** e assegnarlo a **x**:  
`Coin dime = new Coin(0.1, "Dime");`  
`x = dime; // legittimo`

30/11/2004

Laboratorio di Programmazione - Luca Tesei

28

## Conversioni di tipo e cast

- In generale quando si ha un riferimento di tipo **Measurable** non si conosce il tipo vero dell'oggetto a cui punta il riferimento (nessun oggetto può essere creato dall'interfaccia **Measurable**! Solo da classi che la implementano)
- Se però si conosce, ad esempio dal contesto del programma, il vero tipo dell'oggetto riferito da una variabile di tipo **Measurable** si può fare un **cast esplicito**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

29

## Conversioni di tipo e cast

```
Coin dime = new Coin(0.1, "Dime");
Measurable x = dime;
System.out.println(x.getMeasure());
// Cast esplicito:
Coin theSameDime = (Coin) x;
System.out.println(theSameDime.
 getDescription());
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

30

## Conversioni di tipo e cast

- In questo caso particolare siamo sicuri che il cast è corretto
- In generale però se si fa un cast verso un certo tipo e l'oggetto in realtà è di un altro tipo non compatibile allora il programma lancerà un'eccezione
- Il cast esplicito tra numeri era sempre possibile a patto di accettare di perdere informazione
- Il cast fra tipi riferimento ad oggetti non è sempre possibile

## Conversioni di tipo e cast

- Esiste un operatore per verificare se un certo riferimento punta ad un oggetto di un certo tipo
- L'operatore si chiama `instanceof`
- Esempio di uso:

```
if (x instanceof Coin) {
 Coin c = (Coin) x;
 ...
} else
```

## Costanti nelle interfacce

- Le interfacce non possono contenere variabili istanza
- Però possono contenere costanti
- Nel definire una costante in una interfaccia si possono omettere le parole `public static final` perché sono implicite: in quel contesto sono ammesse solo variabili di questo tipo

## Costanti nelle interfacce

```
public interface SwingConstants {
 int NORTH = 1;
 int NORTH_EAST = 2;
 int EAST = 3;
 ...
}
```

## Polimorfismo

- Quando usiamo una variabile il cui tipo è un'interfaccia sappiamo che in realtà l'oggetto puntato è di una delle classi che implementano l'interfaccia

`Measurable x = ...;`

- Quando chiamo il metodo `getMeasure()` su `x` quale metodo viene invocato?
- La JVM va a vedere il tipo A vero dell'oggetto puntato da `x` e, in base a questo, esegue il metodo `getMeasure()` implementato nella classe A!

## Polimorfismo

- Se la classe A è, ad esempio, `BankAccount` viene eseguito il metodo `getMeasure` della classe `BankAccount`
- Se la classe A è `Coin` invece viene eseguito il metodo `getMeasure` della classe `Coin`
- ...
- Questo principio, secondo cui **il tipo effettivo di un oggetto determina il metodo da chiamare**, è detto **Polimorfismo**

## Polimorfismo

- Il termine deriva dal greco e significa multiforme
- In questo caso è usato nel senso che la stessa elaborazione (chiamata ad un metodo di una interfaccia I da un riferimento di tipo I) funziona per oggetti di forme diverse e si adatta alla natura degli oggetti

30/11/2004

Laboratorio di Programmazione - Luca Tesei

37

## Polimorfismo vs Sovraccarico

- Il sovraccarico (overloading) dei nomi si può avere fra diversi metodi di una classe
- Una classe, cioè, può avere diversi metodi con lo stesso nome, ma ognuno deve avere segnatura diversa (per segnatura di un metodo si intende la lista dei tipi dei parametri del metodo stesso)
- Ad esempio per **BankAccount** abbiamo definito due costruttori con segnatura diversa

30/11/2004

Laboratorio di Programmazione - Luca Tesei

38

## Polimorfismo vs Sovraccarico

**BankAccount()**

**BankAccount(double)**

- Quando un certo nome è sovraccarico è il compilatore che decide quale metodo va effettivamente applicato scegliendo quel metodo la cui segnatura fa match con i tipi dei parametri passati nella chiamata
- Nel caso della chiamata di un metodo polimorfico, invece, il compilatore non può decidere nulla!

30/11/2004

Laboratorio di Programmazione - Luca Tesei

39

## Polimorfismo vs Sovraccarico

- Solo al momento dell'effettiva chiamata del metodo durante l'esecuzione si ha a disposizione l'informazione su quale metodo, fra i tanti potenzialmente disponibili, chiamare
- È la JVM che, a runtime, si occupa di andare a guardare il tipo effettivo dell'oggetto e a chiamare quindi il metodo giusto

30/11/2004

Laboratorio di Programmazione - Luca Tesei

40

## Interfacce strategiche

- La soluzione che abbiamo adottato per **DataSet** usando l'interfaccia **Measurable** e il polimorfismo presenta delle limitazioni:
  1. Si può aggiungere l'implementazione dell'interfaccia **Measurable** solo a classi che sono sotto il proprio controllo: ad esempio non si può, a meno di non ricompilare tutti i sorgenti dopo averlo fatto, modificare il codice di una classe della libreria standard come **Rectangle**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

41

## Interfacce strategiche

2. Un oggetto può essere misurato in un unico modo: non si può usare più di una misura. Se si vuole, ad esempio, misurare un conto bancario sia con il saldo che con il tasso di interesse questo non può essere ottenuto con la soluzione che abbiamo visto
- Ripensiamo alla classe **DataSet**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

42

## Interfacce strategiche

- Un **DataSet** deve poter misurare gli oggetti che vi vengono inseriti
- Quando agli oggetti viene chiesto di essere **Measurable**, la responsabilità della misurazione è tutta su di loro
- Da questo scaturiscono le limitazioni che abbiamo visto
- Sarebbe comodo se un'entità diversa dagli oggetti si occupasse della loro misurazione

30/11/2004

Laboratorio di Programmazione - Luca Tesei

43

## Interfacce strategiche

- Realizziamo questo creando una nuova interfaccia:  

```
public interface Measurer {
 double measure(Object anObject);
}
```
- Il metodo **measure** misura un oggetto e restituisce tale misurazione
- Usiamo la proprietà, di tutti gli oggetti in Java, di poter essere visti come oggetti della classe **Object**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

44

## Interfacce strategiche

- Vedremo meglio cosa si intende con questo quando faremo l'ereditarietà
- Per ora ci basta pensare che **Object** rappresenta il minimo comune denominatore di tutti gli oggetti possibili
- La classe **DataSet** migliorata richiederà nel costruttore un oggetto **Measurer** che userà poi per misurare gli oggetti che verranno aggiunti via via

30/11/2004

Laboratorio di Programmazione - Luca Tesei

45

## Interfacce strategiche

```
...
public void add(Object x) {
 sum = sum + measurer.measure(x);
 if (count == 0 || measurer.measure(maximum) <
 measurer.measure(x))
 maximum = x;
 count++;
}
...
private Measurer measurer;
...
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

46

## Interfacce strategiche

- A questo punto possiamo definire tutti i **Measurer** che vogliamo per qualsiasi classe
- Ad esempio prendiamo **Rectangle**
- Vogliamo definire un misuratore per oggetti **Rectangle** che usi l'area come misura
- Basta creare una nuova classe che implementi **Measurer**
- Questa classe non ha bisogno di nessuna variabile istanza e dovrà semplicemente implementare un metodo **measure**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

47

## Interfacce strategiche

```
class RectangleMeasurer implements
 Measurer {
 public double measure(Object anObject) {
 Rectangle aRectangle = (Rectangle)
 anObject;
 double area = aRectangle.getWidth() *
 aRectangle.getHeight();
 return area;
 }
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

48

## Interfacce strategiche

- **RectangleMeasure** è una classe che ha un compito molto preciso e circoscritto
- Serve a misurare esclusivamente oggetti della classe **Rectangle** in base all'area e deve venire passato al costruttore di un oggetto **DataSet** che sarà così in grado di misurare oggetti della classe **Rectangle**
- Il metodo `measure` deve rispettare la segnatura imposta dall'interfaccia quindi dovrà avere un parametro di tipo **Object**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

49

## Interfacce strategiche

- L'implementazione di `measure` della classe **RectangleMeasurer** fa un cast esplicito a **Rectangle**
- Se l'oggetto che viene misurato non è un **Rectangle** sarà sollevata un'eccezione (perdiamo la possibilità di avere **DataSet** a cui vengono aggiunti oggetti di diverso tipo)
- Interfacce come **Measurer** vengono dette **strategiche** poiché gli oggetti delle classi che le implementano mettono in atto una particolare strategia di elaborazione

30/11/2004

Laboratorio di Programmazione - Luca Tesei

50

## Interfacce strategiche

- Ad esempio **RectangleMeasure** ha come strategia quella di misurare i **Rectangle** in base all'area
- Per realizzare una diversa strategia si usa semplicemente un oggetto strategico diverso
- Ad esempio potremmo definire un'altra classe **PerimeterRectangleMeasurer** che utilizza il perimetro come misura

30/11/2004

Laboratorio di Programmazione - Luca Tesei

51

## Interfacce strategiche

- Questo tipo di soluzione si trova spesso nelle implementazioni delle classi che realizzano l'interfaccia grafica e la gestione degli eventi in Java

30/11/2004

Laboratorio di Programmazione - Luca Tesei

52

## Classi interne

- Classi come **RectangleMeasurer** hanno uno scopo limitato e possono tranquillamente venire definite come interne ad altre classi o a metodi
- Nel seguente esempio la classe **RectangleMeasurer** viene definita dentro un metodo `main` al solo scopo di crearne un oggetto da passare a **DataSet**

30/11/2004

Laboratorio di Programmazione - Luca Tesei

53

## Classi interne

```
...
public static void main (String[] args)
{
 class RectangleMeasurer implements
 Measurer {
 ...
 }
 Measurer m = new RectangleMeasurer();
 DataSet data = new DataSet(m);
 ...
}
```

30/11/2004

Laboratorio di Programmazione - Luca Tesei

54

## Test

- Consultare il codice allegato `DataSetStrat.java` e `DataSetStratTest.java` per l'implementazione di `DataSet` con l'interfaccia strategica **Measurer**