Le classi in Java

Fondamenti di architettura e programmazione
Corso di laurea in Comunicazione digitale

Oggetti

- Un oggetto è definito dal suo
  - ***stato*** - descrive le sue caratteristiche
  - ***comportamento*** - quello che può fare

Ad esempio: il modello di una moneta

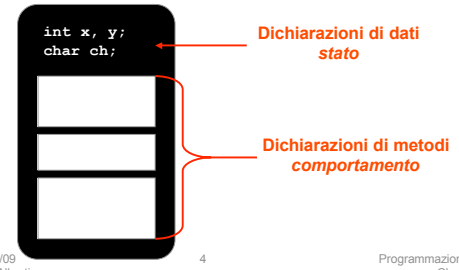
- Una moneta può essere lanciata per consentire una scelta casuale tra due valori: *testa* o *croce*
- Lo stato della moneta è la sua faccia corrente (*testa* o *croce*)
- Il comportamento consiste nel fatto di poter essere lanciata
- Il comportamento può modificare il suo stato

Classi

- Possiamo definire una classe per produrre oggetti specifici
- Ad esempio: una classe per simulare il lancio di una moneta
- Scriviamo la classe `Moneta` per rappresentare il modello di una ***moneta***

Classi

- Una classe contiene la dichiarazione dei *campi* e dei *metodi*: i *membri* di classe



Visibilità dei dati

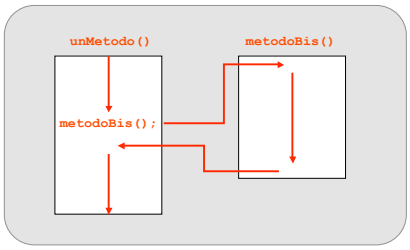
- La ***visibilità*** dei dati indica la parte di programma in cui i dati possono essere usati o referenziati
- Dati dichiarati a livello della classe sono visibili da tutti i metodi della classe
  - Possono essere ***dati di istanza*** o ***dati di classe***
- Dati dichiarati all'interno di un metodo sono visibili solo a quel metodo
  - E vengono chiamati anche ***dati locali***

Scrivere metodi

- La ***dichiarazione di un metodo*** specifica il codice che verrà eseguito quando il metodo viene invocato
- All'invocazione del metodo il flusso di controllo passa al metodo e il codice relativo viene eseguito
- Quindi, il flusso ritorna al punto da cui è partita la chiamata e continua da lì
- L'esecuzione di un metodo può causare il ritorno di un valore all'ambiente chiamante, secondo la sua definizione

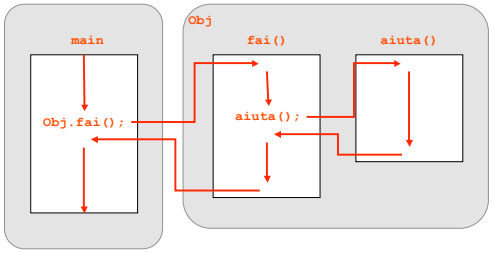
Flusso di controllo nei metodi

- Il metodo invocato può essere della stessa classe
  - Serve quindi solo il nome del metodo



Flusso di controllo nei metodi

- Oppure il metodo chiamato è parte di altra classe



La classe Moneta

- Definiamo una classe `Moneta`:
  - `faccia`, un campo di tipo `int` che rappresenta la faccia corrente
  - `TESTA` e `CROCE`, costanti di tipo `int` che rappresentano i due possibili stati della faccia
  - Il costruttore `Moneta`, per costruire un oggetto
- Definiamo il comportamento della classe con i seguenti possibili metodi:
  - Un metodo `lancia`, per lanciare la moneta
  - Un metodo `dammiFaccia`, per riportare la faccia corrente
  - Un metodo `toString`, per riportare una frase di descrizione per la stampa

La classe Moneta

- `Moneta.java` contiene la definizione della classe
- La classe `Moneta` è usata dalle classi
- `LanciaMoneta.java`
- `ContaMoneta.java`
- Non necessariamente vengono usati tutti i metodi che la classe `Moneta` dispone
  - La classe `ContaMoneta` non usa il metodo `toString`

Dati d'istanza

- Il campo `faccia` nella classe `Moneta` è un **dato d'istanza** perchè ogni istanza (oggetto) della classe ne ha una sua copia
  - Una classe dichiara soltanto il tipo di un dato d'istanza, ma non riserva lo spazio di memoria per archiviarlo
  - È al momento della creazione di un oggetto della classe `Moneta` (quando viene istanziato), che viene anche creato il nuovo campo `faccia`
- Gli oggetti di una classe condividono la definizione dei metodi ma non hanno un unico spazio di memoria per i dati
  - In questo modo due oggetti possono avere stati diversi

Dati d'istanza

- `TestMoneta.java`

Incapsulamento

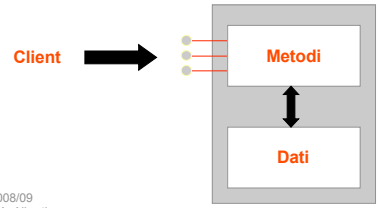
- Un oggetto è un ***astrazione***, che nasconde dettagli al resto del sistema
- Un oggetto può essere visto da due punti di vista:
  - **interno** - la struttura dati o gli algoritmi usati dai metodi
  - **esterno** - l'interazione tra gli oggetti in un programma
- Dal punto di vista esterno, un oggetto è un'***entità incapsulata***, che fornisce un insieme di servizi
- I servizi costituiscono l'***interfaccia*** dell'oggetto

Incapsulamento

- Un oggetto si ***autogestisce***
  - Ogni cambiamento dello stato dell'oggetto (le sue variabili) deve essere ottenuto tramite i metodi di quell'oggetto
  - Una buona programmazione ad oggetti deve rendere difficile, se non impossibile, agli oggetti modificare lo stato di un altro oggetto
- L'utente, o il ***client***, di un oggetto può richiedere i suoi servizi, ma non deve conoscere come questi sono implementati

Incapsulamento

- Un oggetto ***incapsulato*** può essere visto come una ***scatola nera***
- I suoi meccanismi di funzionamento interno sono tenuti nascosti al ***client***, che può solo invocare i metodi che ne costituiscono l'interfaccia



I modificatori di visibilità

- In Java, l'incapsulamento è realizzato mediante l'uso appropriato di ***modificatori di visibilità***
- Un ***modificatore*** è una parola riservata Java che specifica particolari caratteristiche di un metodo o di un dato
  - Il modificatore `final` definisce una costante
  - 3 modificatori di visibilità:
    - `public`
    - `private`
    - `protected`

I modificatori di visibilità

- I membri di una classe dichiarati `public` possono essere manipolati da chiunque
- I membri di una classe dichiarati `private` possono essere manipolati solo dall'interno della classe
- I membri di una classe dichiarati senza alcun modificatore di visibilità hanno una ***visibilità di default*** e possono essere manipolati da ogni classe del pacchetto a cui appartengono

I modificatori di visibilità

- ***Regola generale 1***
  - I dati d'istanza in generale non hanno una visibilità **public**
- ***Regola generale 2***
  - I metodi hanno di norma una visibilità **public** perché possono essere invocati dai client
  - I metodi pubblici si chiamano anche ***metodi di servizio*** o servizi
- ***Regola generale 3***
  - Un metodo definito semplicemente per contribuire alla definizione di un altro si chiama anche ***metodo di supporto***
  - I ***metodi di supporto***, che non devono essere chiamati direttamente dai client, non devono avere una visibilità **public**

Dichiarazione di metodi

- La dichiarazione di un metodo inizia con l'***intestazione***

```
char calc (int num1, int num2, String messaggio)
```

Nome del metodo

Lista dei parametri

Tipo del valore di ritorno

La lista dei parametri specifica il tipo e il nome di ciascun parametro

Il nome di un parametro indica il parametro formale

Dichiarazione di metodi

- L'intestazione di un metodo è seguita dal ***corpo*** del metodo

```
char calc (int num1, int num2, String messaggio)
{
   int somma = num1 + num2;
   char risultato = messaggio.charAt (somma);

   return risultato;
}
```

L'espressione che produce il valore che viene riportato deve essere consistente con il tipo di rientro del metodo

`somma` e `risultato` sono dati locali

Vengono creati ogni volta che il metodo viene invocato e sono distrutti quando il metodo finisce

L'istruzione return

- L'istruzione `return` specifica il valore che verrà riportato
- La sua espressione deve essere conforme al tipo di rientro dichiarato
- Il tipo dell'espressione di rientro di un metodo indica il tipo del valore che il metodo riporta a chi lo ha invocato
- Un metodo che non riporta alcun valore ha tipo di rientro `void`

Passaggio di valori a parametri

- Ad ogni chiamata di un metodo, gli ***argomenti attuali*** sono copiati sui ***parametri formali*** all'atto della chiamata

Chiamata del metodo

Supponendo `calc` un metodo di istanza di una certa classe cui appartiene l'oggetto `obj`

```
ch = obj.calc (2, cont, "Ciao, che fai?");
```

Definizione del metodo

```
char calc (int num1, int num2, String messaggio)
{
   int somma = num1 + num2;
   char risultato = messaggio.charAt(somma);

   return risultato;
}
```


Parametri

- I valori dei parametri attuali non vengono quindi modificati dalla chiamata del metodo
- `TestParametriSemplici.java`

Passaggio di valori a parametri oggetti

- Anche in questo caso i parametri sono passati ai metodi ***per valore***
- All'atto della chiamata del metodo, la copia dei parametri ***attuali*** è *il valore* che viene archiviato nei parametri ***formali***, *dichiarati nell'intestazione del metodo*
- Il passaggio di valore ai parametri è sostanzialmente un assegnamento
- Quando un oggetto viene passato a un metodo, il parametro formale e quello attuale diventano alias
  - Cioè puntano allo stesso oggetto

Passare parametri oggetti a metodi

- Eventuali cambiamenti del valore dei parametri nel metodo non hanno in generale effetto al di fuori di esso, ma tramite i riferimenti si possono modificare campi a oggetti o valori in generale
- `PassaggioParametri.java` con `TestParametri.java`, usa la classe `Num.java`
- Si noti la differenza tra cambiare un riferimento e cambiare l'oggetto a cui punta il riferimento

Prototipi e segnature

- La ***segnatura*** o ***firma*** di un metodo è costituita da
  - Nome del metodo
  - I parametri per numero, ordine e tipo
- Per utilizzare un metodo occorre conoscerne anche il ***prototipo***
  - Segnatura
  - Tipo del valore di ritorno
  - Se il metodo non restituisce nulla si dichiara `void` il tipo della restituzione
- ***Overloading di metodi***, stesso nome diversa segnatura
  - Esempio: il metodo `+` usato per sommare e concatenare

Esempi di prototipi

```
int compareTo(String)

boolean equals(String)

int length()

String toUpperCase()

String substring(int, int)   o più leggibile:
   String substring(int da, int a)
```


Overloading di metodi

- L'***overloading di metodi*** è presente quando si usano gli stessi nomi per indicare metodi diversi
- La ***segnatura*** di ciascun metodo coinvolto dal processo di ***overloading*** deve essere **unica**
- La ***segnatura*** include il ***numero***, ***tipo*** e l'***ordine dei parametri***
- Il tipo di rientro di un metodo **non** fa parte della ***segnatura*** ma del ***prototipo***
- Il compilatore determina la versione del metodo da invocare analizzando i parametri e confrontando con la segnatura

Overloading di metodi

versione 1

```
float prova (int x)
{
   return x + .375;
}
```

versione 2

```
float prova (int x, float y)
{
   return x+y;
}
```

invocazione

```
result = prova (25, 4.32)
```


Overloading di metodi

- Il metodo `println` è sovraccaricato:

```
println (String s)
println (int i)
println (double d)
println (Object o)
```

- E si possono usare le diverse versioni del metodo `println`:

```
System.out.println ("Il totale è:");
System.out.println (totale);
```

- `ProvaOverloading.java`

Overloading di metodi

- Non si può effettuare overloading di un metodo differenziando solo il tipo del valore di rientro
  - `int metodo_f() {<corpo>}`
  - `void metodo_f() {<corpo>}`
- Perché come potrebbe il compilatore comprendere dal contesto cosa chiamare
  - `int x = this.metodo_f();` OK
  - `metodo_f();` ?Qual è il metodo chiamato?

Overloading di metodi o costruttori

- Anche i costruttori possono essere sovrascritti
- Un costruttore sovraccaricato fornisce diversi modi con cui istanziare un nuovo oggetto di una classe
- `Frazione.java`

Overloading e promozione

- In presenza di metodi sovraccaricati la scelta di quello da eseguire avviene valutando il tipo del parametro attuale a confronto con il tipo del parametro formale
- Quando non c'è esatta corrispondenza il parametro attuale viene promosso al tipo più grande
- Esempio con promozione `PromOverloading.java`
- Esempio con riduzione `RidOverloading.java`

Scrivere classi

- Un oggetto ***aggregato*** è un oggetto che contiene riferimenti ad altri oggetti
- Un oggetto di classe `ContoBancario` è un oggetto aggregato perché contiene il riferimento all'oggetto di classe `String`
- Un oggetto ***aggregato*** rappresenta una relazione ***has-a*** *(ha-un)*
- Un conto di banca *ha-un* nome
- `ContoBancario.java` e `TestContoBancario.java`

Scrivere classi

- Talore un oggetto deve interagire con altri oggetti dello stesso tipo
- Sommare due oggetti rappresentanti frazioni della classe **`frazione`**:

```
risultato = fraz_1.somma(fraz_2);
```

- L'oggetto `fraz_1` esegue il metodo `somma` con l'oggetto `fraz_2` che gli viene passato come parametro e ritorna un nuovo oggetto
- `Frazione.java` e `TestFrazione.java`

La classe StringTokenizer

- La classe `StringTokenizer` è definita nel pacchetto `java.util`
- Un oggetto `StringTokenizer` separa una stringa di caratteri in sottostringhe più piccole (*tokens*)
- Il costruttore `StringTokenizer` riceve come parametro la stringa originale da separare
- Per default, il tokenizer separa la stringa di input agli spazi bianchi
- Ogni invocazione del metodo `nextToken` riporta il prossimo token nella stringa di input
- `InputInArray.java`

Decomposizione di metodi

- Un metodo dovrebbe essere relativamente piccolo, in modo da poter essere compreso
- Un metodo potenzialmente complesso dovrebbe venire decomposto in diversi metodi più piccoli per maggior chiarezza
- Quindi, un metodo di servizio di un oggetto può chiamarne altri di supporto per raggiungere l'obiettivo
- **`PigLatin.java`**
- **`PigLatinTranslator.java`**