Programmazione II

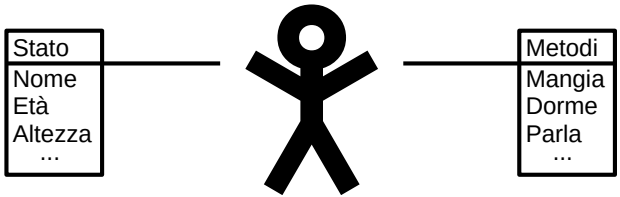
2. Object Orientation (PDJ 2)

Dragan Ahmetovic

Il Paradigma Object Oriented

Differenza tra approccio procedurale e orientato a oggetti

- Nei linguaggi procedurali la logica del programma è strutturata in:
  - Dei dati memorizzati nelle variabili
  - Delle azioni sui dati, definite nelle funzioni
- Nei linguaggi orientati ad oggetti i dati sono strutturati come oggetti aventi sia:
  - Uno **stato**, ovvero una serie di variabili (**attributi**) che caratterizzano l'oggetto
  - Dei comportamenti (**metodi**) che agiscono (ispezionando o modificando) sullo stato dell'oggetto
- Qual è la differenza? (cosa ci permette di fare questa astrazione?)
  - **Encapsulation** (fondamento della OOP): modificare stato dell'oggetto solo attraverso i suoi metodi!



Classi - Archetipi di Oggetti

Per definire questi oggetti in Java scriveremo delle **Classi** di oggetti, che:

- Definiscono quali attributi un oggetto di quella classe avrà
  - ogni oggetto (istanza) della classe potrà avere valori degli attributi diversi
- Definiscono quali metodi potranno essere chiamati su un oggetto di quella classe
  - tutti gli oggetti della classe avranno gli stessi metodi
- Le classi si definiscono con '`public class <NomeClasse> {...}`' (nome inizia con maiuscola)
- Esistono classi che contengono solo metodi statici (non modificano lo stato di un oggetto)

Package

Package - Collezioni di Classi

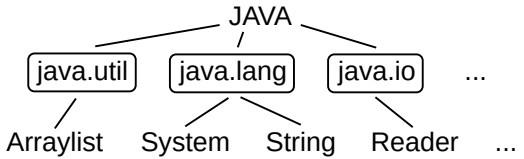
In Java le classi sono organizzate in **Package** (pacchetti), con due obiettivi:

- **Encapsulation** (Incapsulamento) ovvero suddivisione del codice in moduli indipendenti
  - Con **visibilità** esterna solo dei metodi indispensabili per svolgere il compito richiesto
- **Naming** (Assegnamento nomi) separati per evitare di avere casi di omonimia tra classi
  - Nel caso di un conflitto tra questi due obiettivi, è buona norma dare priorità all'incapsulamento
- I package si definiscono con '`package <nomePackage>;`' (nome inizia con minuscola)
  - tutti i file del package dovrebbero essere nella stessa cartella
  - se non viene specificato un package, è implicitamente considerato il *default* package

Package

Fully qualified name

- I package seguono una gerarchia ad albero (per convenzione rispecchiata nel filesystem)
  - si può creare un package dentro un altro con `` `package <nomePackage>.<nomeSubpackage>;` ``
  - il percorso nella gerarchia definisce il **fully qualified name** con cui si riferisce al package
  - Le classi allo stesso livello di gerarchia si vedono senza usare il fully qualified name
- Si può fare **import** dei package o classi per richiamarli direttamente
  - sintassi: `` `import fully.qualified.name.Classe;` `` (* per importare tutte le classi del package)
  - attenzione ai conflitti! (non si possono importare due classi con lo stesso nome)
  - Tutte le classi del package **java.lang** sono automaticamente importate



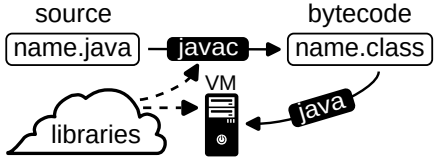
Compilazione ed esecuzione del codice

Java - compilazione

- Java è un linguaggio **compilato**. Genera eseguibili NON in codice macchina ma **bytecode**
  - Per eseguire il **bytecode** serve una **macchina virtuale** (VM) che lo interpreta in istruzioni macchina
  - Il bytecode può girare sulla Java VM di qualsiasi hardware -> **WORM** (Write Once Run Many)
  - Le "librerie" delle classi sono contenute nel percorso visibile dal compilatore e dalla VM

Java - Esecuzione

- Le classi sono definite in file **.java** (in genere una per file)
  - Nome file = nome della classe (necessario per classi con visibilità pubblica)
  - Il comando `` `javac <nomeFile>.java` `` produce il file .class corrispondente alla classe definita
  - Il comando `` `java <nomeClasse>` `` esegue il metodo `` `public static void main(String[] args)` ``



Espressioni

Espressioni

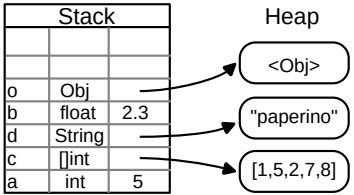
- In Java le espressioni sono eseguite da sinistra verso destra e possono essere costituite da:
  - atomi: variabili, costanti, letterali
  - operatori (+, -, =, ...)
  - invocazioni di metodi
- Le espressioni hanno come risultato un **tipo** e un **valore**
  - 3+2, 2 == 3
- Possibile usarle per costruire altre espressioni (es: come operando, parametro di funzione)
  - 6 + Math.sqrt(i)
- A **compile time** è possibile calcolare il tipo di una qualunque espressione per induzione
  - Java è **type safe** -> non ci possono essere errori di tipo durante l'esecuzione del programma
- Il valore invece potrebbe essere calcolabile solo a **run time**
  - dipende da valori effettivi o condizioni (es: input da utente)

Variabili ed Oggetti

Variabili locali

Le variabili in java hanno un **nome** e un **tipo**. Questi tipi possono essere:

- **primitivi** (int, float, boolean...), generati sulla **stack**
- **riferimenti** (oggetti) con riferimento nella **stack** e contenuto nella **heap**
  - **oggetti** si **inizializzano** con operatore **new** e chiamata al metodo **costruttore** della loro **Classe**
  - es: `` `Object o = new Object();` `` dove l'oggetto o è detto istanza della classe Object
  - **String** e **array** si possono anche creare con un literal (es: `` `s String = "ciao";` ``)
  - A variabile riassegnata o metodo terminato l'oggetto in heap è cancellabile dal **garbage collector**

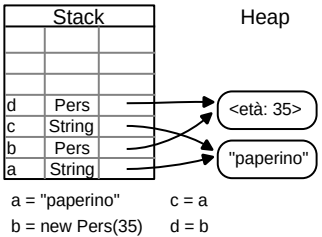


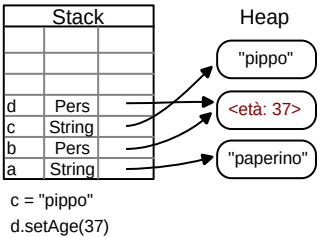
Variabili ed Oggetti

Oggetti Mutabili e Immutabili

Le variabili **riferimento** possono **puntare** allo stesso oggetto

- Se creo una variabile `` `a String = "paperino";` `` questa allocherà uno spazio sulla **heap**
- Se assegno una nuova variabile `c=a` ora queste **punteranno allo stesso spazio** in heap
  - Questo meccanismo risulta in un risparmio di risorse
- Gli oggetti **non mutabili** (es: **String**), non si possono cambiare, ma solo reinizializzare
- Per gli oggetti **mutabili** le modifiche influenzano tutti i riferimenti (es: sia `b` che `d`)



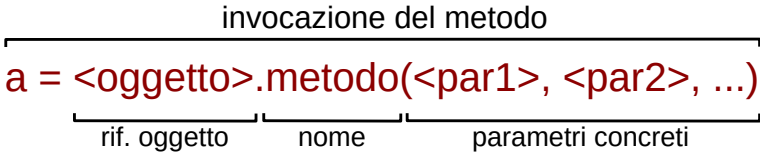


Metodi

Invocazione dei metodi

Per invocare un metodo bisogna:

- Riferirsi al nome del metodo sull'oggetto che lo rende disponibile.
- Passare i parametri richiesti dal metodo (sono passati **per valore**!).
  - però il valore nel caso degli oggetti è il loro puntatore in memoria
  - possono essere modificati (usando i loro metodi o indice per gli array)!
- Usare o assegnare il valore di ritorno a qualche variabile (opzionalmente)
- Sia il riferimento all'oggetto che i parametri possono essere delle **espressioni**!
  - `` `c = a.somma(1,3)` ``
  - `` `f(x).calcola(z,3+5,g(y))` ``

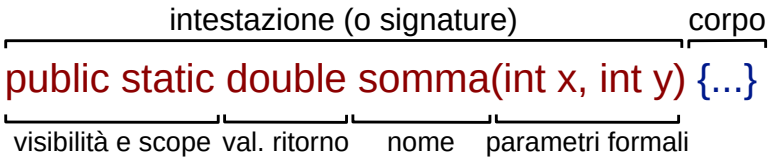


Metodi

Definizione dei Metodi

Per scrivere un metodo bisogna definire la sua **intestazione** (o signature), definita da:

- Nome del metodo
- Parametri **formali** definiti con dal loro tipo di dato e nome
- Tipo di valore restituito (return value). **Deve** esserci uno solo. Se vuoto si usa la keyword **void**
- Keyword di scope e visibilità (ricordate le lettere maiuscole in GO?). Le vedremo più avanti
- Corpo del metodo (effettivo codice imperativo)



Metodi

Overloading dei Metodi

- È possibile l'**overloading** dei metodi - ovvero creare metodi che abbiano lo stesso nome ma parametri diversi.
- Utile per avere un naming e un comportamento coerente e prevedibile.

Il concetto che un unico simbolo può avere diversi significati a seconda del contesto d'utilizzo si chiama **Polimorfismo** ed è un fondamento della OOP

Metodi

Conversioni Implicite e Overloading

- Nel caso di espressioni con tipi diversi in alcuni casi Java può fare **casting implicito**
  - Ad esempio in `3+3.5`, l'intero 3 viene implicitamente castato a float per permettere l'operazione
- Questo è possibile anche nel caso dei **parametri** dei metodi
  - `somma(double a, double b)`, se chiamato con un parametro intero effettua casting implicito
- Tuttavia, nel caso dell'**overloading** dei metodi ci **possono essere molteplici possibilità**
  - Ad esempio dati due metodi `somma(int a, double b)` e `somma(double a, double b)`
  - la chiamata `somma(2, 3)` quale dei due invoca?
- Si effettua la chiamata più specifica (**most specific**)
- Se non sono possibili conversioni o non c'è una più specifica vi è un **errore di compilazione**
  - Ad esempio dati due metodi `somma(int a, double b)` e `somma(double a, int b)`
  - la chiamata `somma(2, 3)` quale dei due invoca?
- Il meccanismo che decide quale metodo invocare si chiama **Dispatching**

Collezioni

Collezioni di Object

- La variante dinamica degli array sono le ArrayList (simili ma più complesse delle slice in GO).
- Posso metterci dentro **oggetti** di tipo diverso
- Per funzionare con i diversi tipi, assumono che i valori siano di tipo **Object** (genitore di tutti i tipi)
- Devo fare **casting esplicito** per usare i metodi del loro tipo giusto

```
1 String s = "lalalala" //nuova stringa
2 List a = new ArrayList(); //nuova Lista, e specificatamente una ArrayList
3 a.add(s); // aggiungo Stringa s
4
5 String b = ((String) a.get(0)); //restituisco l'oggetto all'indice 0 di a.
6 // Per assegnarlo a una String o usare i metodi del tipo String lo devo convertire
```

Collezioni con Generics

- Spesso però non mi serve avere oggetti diversi in una ArrayList.
- Inoltre fare continuamente casting esplicito rallenta la scrittura del codice ed è prono a errori.
- È possibile restringere la collezione ad un **tipo specifico** di dato (es: ArrayList<String>).
- Questo meccanismo si chiama **Generics** e lo vedremo più avanti

```
1 String s = "lalalala" //nuova stringa
2 List<String> a = new ArrayList<>(); //nuova List di String, specificatamente una ArrayList
3 a.add(s); // aggiungo String s
4
5 String b = a.get(0); //restituisco l'oggetto String all'indice 0 dell'ArrayList a
6 //Non serve casting
```

Input e Output

I/O su terminale in Java

- Per Output usiamo System.out.println(...) o simili
- Per input da **riga di comando** si usa il parametro `String[] args` del metodo main
- Per input da **standard input** si usa **java.util.Scanner**
  - Un nuovo oggetto di tipo `Scanner` si inizializza con sorgente `System.in` (standard input)
  - Per vedere se c'è altro da leggere uso `S.hasNext()`, per leggere nuovo contenuto uso `S.next()`
  - È un **iteratore** (oggetto che itera su un insieme e restituisce nuovi elementi)
  - Esistono metodi appositi per leggere tipi primitivi (es: per int `hasNextInt()` e `nextInt()`)

```
1 Scanner S = new Scanner(System.in);
2
3 while(S.hasNext()) {
4   System.out.println(S.next());
5 }
```

Wrapping

Oggetti Wrapper per tipi primitivi

- Gli **oggetti primitivi** non possono essere usati in molti contesti in Java.
- Ad esempio **ArrayList** (o altre **Collections**) lavorano solo su Oggetti
- Per questo, per ogni tipo primitivo vi è un tipo di oggetto corrispondente.
- Es: int->Integer, boolean->Boolean ..
- Ci sono degli appositi metodi per passare da tipo primitivo a oggetto e viceversa
- Questa meccanica si chiama **Wrapping** (avvolgimento) e **Unwrapping** (svolgimento)

```
1 int a = 3;
2
3 //metodi per fare il wrapping da int a Integer
4 Integer b = new Integer(a); //costruttore
5 Integer c = Integer.valueOf(a); //factory method
6
7 //metodo per fare unwrapping da Integer a int
8 int d = b.intValue();
```

Wrapping

Boxing e Unboxing

- Possibile **conversione automatica** da tipo primitivo a oggetto corrispondente
  - Nelle versioni di Java a partire dalla 1.5
- Questa capacità viene chiamata **Boxing** (inscatolamento) o Autoboxing
- L'operazione opposta è **Unboxing**
  - Unboxing viene fatto anche comparando con **==**

```
1 int a = 3;
2 Integer b = a; //boxing di a
3
4 System.out.println(b==a); //unboxing di b
```

Programmazione II

2. Object Orientation (PDJ 2)

Dragan Ahmetovic