1 Lezione del 25-09-25

Continuiamo a vedere gli aspetti del Java legati alle classi.

1.1 Metodi

Java supporta i **metodi** (funzioni definiti all'interno di classi) *statici* e *non statici*. Per questi metodi è supportato l'*overloading*, cioè lo stesso nome finche gli argomenti (cioè complessivamente la *firma*) variano.

Ogni volta che un metodo viene chiamato sullo stack viene creato un *record di attivazione*, cioè una struttura dati che contiene:

- Lo spazio riservato per gli argomenti formali, inizializzati al valore passato negli argomenti di chiamata;
- Lo spazio riservato alle variabili locali del metodo;

Al termine dell'esecuzione del metodo il record di attivazione corrispondente viene distrutto. Chiaramente la natura dello stack implica che i record verranno naturalmente creati e distrutti seguendo l'innestamento delle chiamate di metodo.

1.1.1 Passaggio di parametri

Il passaggio dei parametri ai metodi è *sempre* **per valore**, cioè nuove variabili vengono create col valore passato, e la variabile originale non viene modificata.

Notiamo che nel caso di classi, per valore viene passato il *riferimento* alla classe e non la classe stessa, cioè l'istanza di classe rimane unica e allocata sull'heap.

Ricapitolando, abbiamo quindi che:

- Il metodo chiamato non può modificare il valore di una variabile di tipo primitivo del chiamante;
- Il metodo chiamato non può modificare il valore di una variabile di tipo riferimento del chiamante;
- Di contro, il metodo chiamato *può* modificare il valore di una variabile oggetto di cui ha ricevuto il riferimento del chiamante (ovvero i riferimenti del chiamante e del metodo sono variabili diverse che puntano allo stesso oggetto, come i puntatori del C/C++).

1.1.2 Variabili locali

Le variabili locali, come già accennato, non hanno un valore di default e prima che venga assegnato un valore sono `undefined`. Usare una variabile locale prima che il suo valore sia ben definito comporta un errore.

Esistono casi in cui il compilatore non è abbastanza intelligente da capire che una variabile otterrà sicuramente un valore fra tutti i rami possibili di esecuzione. In tal caso basta inveire contro Oracle ed assegnare un valore qualsiasi alla variabile in fase di inizializzazione.

1.1.3 Riferimento `this`

Il Java supporta il riferimento (non propriamente *puntatore*) this alla classe corrente.

Nel caso si voglia semplicemente usare un campo della classe non è necessario usare `this`, in quanto questo è già nello scope di visibilità del metodo.

```
class Class {
  int a;

  void prolisso() {
    this.a++; // inutile! basta a++;
  }
}
```

`this` diventa allora utile quando gli argomenti formali *oscurano* i campi propri della classe:

```
class Class {
  int a;

  void oscura(int a) {
    this.a++; // il campo della classe
    a++; // l'argomento formale
  }
}
```

L'uso della sintassi `this.campo = campo` è estremamente comune quando si definiscono costruttori che prendono i campi della classe come argomenti:

```
class Studente {
  String nome;
  String cognome;
  String matricola;

  Studente(String nome, String cognome, String matricola) {
    this.nome = nome;
    this.cognome = cognome;
    this.matricola = matricola;
  }
}
```

Una forma più compatta della stessa cosa è la seguente:

```
class Studente {
  String nome;
  String cognome;
  String matricola;

  Studente(String nome, String cognome, String matricola) {
    this(nome, cognome, matricola);
  }
}
```

Un'altro caso di utilizzo di `this` è quando abbiamo effettivamente bisogno di un riferimento esplicito alla classe d'appartenenza del metodo (si pensi a un metodo che iscrive la sua istanza di classe ad una lista):

```
class Class {
  void iscriviti(Class[] lista, int idx) {
    lista[idx] = this;
  }
}
```

1.2 Membri statici

I campi e metodi statici sono utili a gestire informazioni relative all'intera classe e non agli oggetti istanza di classe generati da questa.

L'esempio tipico dei membri statici è per avere comportamento simile a *"fabbriche"*:

```
class Veicolo {
  String propretario;
  static int contatore = 0; // i veicoli istanziati da questa classe

  Veicolo(String proprietario) {
    this.proprietario = proprietario;
    contatore++;
  }
}
```

I valori statici non vengono allocati sull'heap assieme al resto delle istanze di classe, ma nella memoria statica (in quanto esistono una volta sola per tutta la classe).

L'accesso ai membri statici si fa usando la notazione puntuale direttamente sul nome di classe. Si può usare il riferimento di un'istanza di classe per fare la stessa cosa, ma questo è sconsigliato in quanto poco chiaro nel suo scopo (un programmatore che guarda l'accesso al membro non può sapere se questo è statico o meno senza guardare all'interno della classe).

Usare classi di utilità che definscono solo metodi statici è un pattern comune che sostituisce quello che in altri linguaggi sarebbe effettivamente rappresentato dalle funzioni globali:

```
class Math {
  public static void Sqrt(float num) {
    // ...
  }

  // ...
}
```

Ricordiamo che lo stesso metodo `main()` è dichiarato come `public` e `static`.

Se dentro una classe si vuole usare un metodo statico di quella classe, si può chiaramente usare il nome del metodo senza ulteriori qualificatori.

I campi statici possono essere inizializzati all'interno della dichiarazione di classe, prendendo come valori iniziali letterali, altri valori statici o risultati di metodi statici. Chiaramente non si possono usare variabili istanza o risultati di metodi istanza (possibilmente sconosciuti al tempo di definizione della variabile statica).

Chiaramente, nei metodi statici non possiamo usare il riferimento `this`, in quanto non c'è nessuna istanza di classe a cui riferirsi. Di contro, possiamo usare:

- Membri statici;
- Argomenti statici o non statici.

1.3 Pacchetti

I **pacchetti** o *package* servono a:

- Raggruppare classi tra loro correlate;
- Evitare conflitti fra i nomi di classe;

- Organizzare i progetti in maniera modulare.

La convenzione del Java è quella di usare un file per ogni classe, col nome della classe. Usando la direttiva `package` possiamo inserire una data classe all'interno di un pacchetto (buona pratica è rispecchiare questa struttura con le directory del progetto):

```
// ClassA.java
package MioPacchetto;

class ClassA {
  // ...
}

// ClassB.java
package MioPacchetto;

class ClassB {
  // ...
}
```

In questo caso per riferirsi alle classi `ClassA` e `ClassB` qualificheremo con la notazione puntale il pacchetto:

```
MioPacchetto.ClasseA;
MioPacchetto.ClasseB;
```

Pratica piuttosto tipico è includere la notazione puntuale anche nei pacchetti, cioè specificare i nomi di pacchetto in maniera gerarchica (`categoria.pacchetto.sottopacchetto` , ecc...). Questo è estremamente utile se usato assieme alla pratica accennata di prima di rispecchiare la struttura dei pacchetti in quella delle directory, sopratutto in progetti particolarmente complessi con molte componenti.

Una convenzione tipica nell'industria per trovare nomi univoci di pacchetto è quella di usare nomi di dominio al contrario, ad esempio:

```
com.ibm.db. // ...
it.unipi.email // ...
```

questo va effettivamente di pari passo con la qualificazione gerarchica data dai nomi di dominio DNS, ricordando il fatto che nel loro contesto originale vengono messi al contrario.

1.3.1 Pacchetto anonimo

Le classi che non vengono messe esplicitamente in un pacchetto finiscono nel pacchetto senza nome di default. Questo è sconsigliabile in quanto la classe risulterà a questo punto inaccessibile a classi che appartengono a pacchetti con nome, cioè l'accesso ai pacchetti va dall'alto verso il basso e non viceversa (non si può usare l'operatore di risoluzione vuoto come in C/C++).

1.3.2 Pacchetti e librerie

Tutte le librerie Java vengono implementate come pacchetti. Di base, il JDK definisce alcune librerie standard fra cui:

- `java.lang`: contiene alcuni tipi di default del linguaggio;
- `java.io`: contiene strumenti per la gestione dell I/O;

- `java.util`: contiene alcune utilità;
- ecc...

1.3.3 Regole di accesso fra pacchetti

Non sempre una classe in un pacchetto può accedere a classi di altri o del solito pacchetto: le regole di accesso vengono definite dalle qualificazioni della classe.

Una classe *top-level* (cioè definita in un file di classe) può avere 2 possibili qualificazioni:

- `public`;
- non `public` (nessuna qualificazione).

Le classi `public` sono disponibili a classi dello stesso pacchetto o di altri pacchetti, le classi non `public` sono invece disponibili solamente a classi dello stesso pacchetto.

Pssiamo usare i nomi non qualificati per accedere a classi nel pacchetto in cui ci troviamo. Per accedere a classi in altri pacchetti dobbiamo invece usare il nome completamente qualificato.

Un'alternativa può essere quella di **importare** la classe o l'intero pacchetto (in questo caso si includono tutte le classi del pacchetto) usando la direttiva `import` all'inizio del file che definisce la classe corrente.

Nella direttiva import si può usare anche la *wildcard* `*` per importare tutte le classi del pacchetto. Questo rende più esplicito quali pacchetti stiamo importando: con la wildcard si importa solo il pacchetto specificato, senza anche tutti i sottopacchetti.

Siano ad esempio due pacchetti `abc` e `abc.def`, dove il secondo è sottopacchetto del primo. In questo caso potremo dire:

```
1 import abc; // importa anche abc.def
2 import abc.*; // non importa abc.def
3 import abc.def.*; // ok
```

Un pacchetto particolare è `java.lang`, che viene importato sempre in ogni file.

Nel caso 2 o più pacchetti importati contengano classi con lo stesso nome, occorrerà nuovamente usare i nomi completamente qualificati. Questo talvolta può accadere anche con le librerie di default (ad esempio sia `java.util` che `java.sql` definiscono una classe `Date`).

1.3.4 Classi, pacchetti e gerarchia di file

Abbiamo visto come la gerarchia dei file che definiscono classi nelle directory possono rispecchiare la struttura innestata dei nomi di pacchetto, e anzi che questa è prassi consigliata. Tale convenzione permette infatti al compilatore (e all'IDE che scegliamo di usare) di gestire in maniera efficiente la compilazione delle classi.

Vediamo pero che le regole che ci siamo dati finora (una classe per file) sono leggermente stringenti: la prassi effettiva è di definire *al più* una classe di tipo `public` all'interno di un file, e assicurarsi che questa classe dia il nome al file. A questo punto potremmo definire altre classi non `public` all'interno dello stesso file, sempre assicurandosi di seguire la buona pratica di mantenere negli stessi file classi semanticamente legate fra di loro (classi utilità a classi top level, ecc...).

Ritorniamo all'aspetto della compilazione: il compilatore, come abbiamo detto, sfrutterà la gerarchia di directory per trovare le classi importate attraverso il loro nome completamente qualificato. L'insieme di cartelle a partire dalle quali il compilatore (e la JVM, si parla di linking dinamico) effettua questa ricerca viene detto **classpath**.

Se non specificato il classpath è automaticamente preso alla directory corrente. Altrimenti, questo può essere specificato con l'opzione `-cp` o `-classpath` del comando `java`. Più punti di partenza possono essere specificati col separatore `:` in ambiente Unix.

In ogni caso, le classi di sistema fornite assieme alla JVM vengono trovate automaticamente.

1.4 Qualificatori di accesso

Abbiamo visto il qualificatore `public` (e di conseguenza il non `public`) per le classi toplevel. Vediamone gli altri:

- `private`: il campo o il metodo può essere acceduto solo dalla stessa classe in cui è definito;
- Nessun qualificatore: già visto, il campo o il metodo può essere ecceduto dalla stessa classe in cui è definito e da tutte le classi dello stesso package;
- `protected`: il campo o metodo può essere acceduto dalla stessa classe in cui è definito e da tutte le sue sottoclassi (anche in altri package), nonché da tutte le classi nello stesso package;
- `public`: il campo o metodo può sempre essere acceduto.

1.5 Singoletti

La struttura delle classi del Java ci permettono di implementare **pattern** di classi. Uno di questi è il **singoletto**, cioè una classe di cui esiste una sola istanza in tutto il programma.

Questo si presenta più o meno come:

```
public class Singleton {
  private static Singleton m;
  private Singleton() { /* ... */ }; // definiamo costruttore privato

  public static void getInstance() {
    if(m == null) {
      m = new Singleton();
    }
    return m;
  }
}
```

In questo modo impediamo ad altri metodi di definire nuove istanze della classe `Singleton`:

```
Singleton more = new Singleton(); // errore! costruttore privato
```

1.6 Array

Vediamo come funzionano gli array in Java. Questi sono simili a quanto visto in altri linguaggi: contenitori contigui di elementi dello stesso tipo identificati da un indice. La loro dimensione è definita a tempo di esecuzione, cioè:

```
int[] a, // inizializzato un riferimento ma non esiste array
a = new int[5]; // adesso esiste array

// si puo' fare in una volta sola:
int[] b = new int[5];
```

Gli array sono gestiti quindi come le classi, cioè sono allocati nell'heap e li gestiamo attraverso riferimenti sullo stack.

Notiamo che, sebbene la dimensione sia determinata a tempo di esecuzione, la dimensione da lì in poi è fissa: per ridimensionare array bisogna usare le stesse tecniche per gli array del C/C++. Una funzionalità che non ci portiamo invece dal C++ è l'aritmetica dei puntatori, cioè preso un riferimento ad array non possiamo incrementare ma solo selezionare elementi dell'array:

```
int[] a = new int[5];
a[1]; // ok
a++; // errore
```

Nel caso di accessi fuori bound, l'accesso non è permesso e viene lanciata un'eccezione.

1.6.1 Valori di default in array

Per gli array esistono valori di default:

- `false` per i tipi booleani;
- 0 per i tipi numerici;
- `null` per i riferimenti.

Valori espliciti possono essere forniti con la notazione graffa, ed entrambe le forme sotto sono valide

```
int[] a = {2, 3, 4};
int[] a = new int[] {2, 3, 4};
```

1.6.2 Array di riferimenti

Se si creano array di riferimenti, al momento dell'inizializzazione dell'array non esistono oggetti, cioè:

```
Studente[] s1 = new Studente[5];
// esistono 5 riferimenti a studente ma nessuno studente!
s1[0].getMedia(); // errore
```

Ciò che possiamo fare è quindi assegnare ai riferimenti creati nuovi oggetti:

```
s1[0] = new Studente("Mario", "Rossi");
```

Esistono diverse forme di inizializzazione equivalenti:

```
Studente[] s2 = { new Studente("Mario", "Rossi"), new Studente("Luigi", "
    Verdi") };
Studente[] s3 = new Studente[] { new Studente("Mario", "Rossi"), new
    Studente("Luigi", "Verdi") };
```

1.6.3 Array multidimensionali

Per creare un'array multidimensionale si usa la solita sintassi a parentesi quadre multiple:

```
Cell[][] grid = new Cell[HEIGHT][WIDTH];

grid[y][x] = Cell.grass;
```

La struttura generata sarà la classica configurazione a direttorio degli array multidimensionali, dove indirizziamo array con array da sinstra verso destra.

L'ultima dimensione può essere omessa, cioè si può creare un direttorio di dimensione fissa che riferisce ad array di dimensione variabile:

```
int[][] grid = new int[3][]; // ok
int[][] grid = new int[][3]; // errore
```

Si possono ottenere array multidimensionali non rettangolari, ad esempio come:

```
int[][] triangle = new int[3][];

for(int i = 0; i < 3; i++) {
  triangle[i] = new int[i + 1];
}
```

1.6.4 Metodi su array

Esistono metodi di utilità contenuti nella classe `System` pensati per gestire le Array. Questi sono ad esempio:

- `arrayCopy(Object src, int srcPos, Object dest, int destPos, int length)` per la ricopiatura da regioni di array ad altre regioni di array. Secondo la logica discussa in 2.3, dovremo chiamare questa funzione come `System.arrayCopy();`
- ecc...

Esiste anche la classe `Arrays` che espone alcuni metodi di utilità specifici alle array:

- `sort(Object array)` per l'ordinamento di array;
- ecc...

Ulteriori metodi possono essere trovati nella documentazione di java, a `docs.oracle.com/javase/<versione>/docs/api/`.

Inoltre, notiamo che la lunghezza di un'array è accessibile come proprietà usando `.length`.

1.7 Stringhe

Le stringhe in Java non sono le stringhe terminate del C/C++, ma istanze di classe `String`. Le stringhe definite come istanze di questa classe sono **immutabili**: per ottenere stringhe mutabili dobbiamo usare le classi:

- `StringBuffer`: può essere manipolata da più *thread* (è *thread-safe*);
- `StringBuilder`: deve essere manipulata da un solo thread.

Le stringhe, essendo istanze di classe, vengono puntate da riferimenti ed allocate sull'heap.

Si possono concatenare con l'operatore `+`, che può essere anche applicato a stringhe e valori di altro tipo (provocando una conversione implicita dal tipo usato a stringa).

1.7.1 Metodi su stringhe

Come per le array, disponiamo di alcuni metodi utili sulle stringhe:

- `substring(int beginIndex, int endIndex)`: restituisce la sottostringa dall'indice `beginIndex` all'indice `endIndex`;
- `.length()`, definito come metodo istanza e non come proprieta (come era stato per le array), restituisce la lunghezza della stringa;
- ecc...

Possiamo costruire stringhe a partire da array di caratteri possiamo usare più sintassi:

```
String str = new String()
```

```
// questo    codice
class Stocazzo {

}
```

Per effettuare il confronto fra stringhe dobbiamo usare il metodo `equals()`, e non l'operatore `==`, in quanto questo confronta i riferimenti e non i valori delle stringhe riferite.