

Java - Supporto al Corso

Autore: Ferdinando Primerano
Revisione: Stefano Rubinetti
2020/2023

Indice

1 - Basi della programmazione

- 1.1 Concetto di programma e introduzione informale a Java
- 1.2 Codice sorgente (.java) e codice pseudo compilato (.class)
- 1.3 Blocco di codice
- 1.4 Istruzioni
- 1.5 Variabile
- 1.6 Concordanza dei tipi e regole di assegnamento
- 1.7 Chiamata a metodo
- 1.8 Analisi di un primo programma Java
- 1.9 Espressioni
- 1.10 Il tipo boolean e le condizioni
- 1.11 Condizioni composte
- 1.12 Selezione: scegliere fra più cammini possibili e il costrutto if
- 1.13 Scegliere un valore fra due - l'operatore ternario
- 1.14 Gestire una casistica discreta: il costrutto switch
- 1.15 Iterazione: Introduzione al concetto di ciclo, do-while e while
- 1.16 Ciclo for
- 1.17 Saltare una iterazione e uscire prima dal ciclo: continue e break

2 - Introduzione alla scrittura di metodi

- 2.1 Definizione di metodo
- 2.2 Analisi di un metodo di esempio
- 2.3 Uso del return

3 - Classi e Oggetti

- 3.1 Creare un tipo
- 3.2 Uso pratico della classe: dichiarare e creare oggetti
- 3.3 Costruttori, impliciti ed esplicativi
- 3.4 Stato dell'oggetto, unione di dati e codice
- 3.5 Incapsulamento - concetto teorico e necessità pratica
- 3.6 Incapsulamento in pratica: i livelli di accesso
- 3.7 Incapsulare Person - getter e setter
- 3.8 Il rapporto di uso fra oggetti o classi e oggetti e primo riepilogo
- 3.9 Comunicare con l'oggetto: metodi e passaggio di parametri
- 3.10 Il concetto di this
- 3.11 Polimorfismo dei costruttori
- 3.12 La classe come entità autonoma dall'oggetto - variabili di classe
- 3.13 La classe come entità autonoma dall'oggetto - metodi di classe e regole di accesso
- 3.14 Scope di oggetto e scope di classe
- 3.15 Gestire i null
- 3.16 Overloading dei metodi
- 3.17 Gli oggetti dopo la morte: il garbage collector

4 - Vettori

- 4.1 Definizione, parti e fasi della vita di un vettore
- 4.2 Scorrimento di un vettore
- 4.3 I vettori come insiemi e le operazioni su insiemi: map, filter e reduce
- 4.4 Riduzione di un vettore a un valore (reduce)
- 4.5 Operazione di filtraggio di un vettore (filter)
- 4.6 Operazione di trasformazione 1-1 di un vettore (mapping)
- 4.7 Vettori di oggetti

5 - L'oggetto aggregatore nell'architettura del sistema

- 5.1 La metafora del motore
- 5.2 L'oggetto aggregatore
- 5.3 Costruire la classe dell'oggetto aggregatore
- 5.4 Il ruolo del main()
- 5.5 Oggetto aggregatore con lettura da file

6 - Eccezioni

- 6.1 Gestione basilare delle eccezioni
- 6.2 Propagazione delle eccezioni e individuazione del gestore
- 6.3 Try con catch multipli e ordine di gestione
- 6.4 Blocco finally
- 6.5 Riepilogo della gestione delle eccezioni
- 6.6 Caso di studio: l'oggetto di classe Census con gestione e propagazione di eccezioni

7 - Altre strutture dati

- 7.1 Concetto informale di struttura dati
- 7.2 La struttura dati List
- 7.3 Iterable e for-each
- 7.4 Caso di studio: l'oggetto di classe Census con l'uso di liste e for-each
- 7.5 Un cenno ai tipi boxati e il loro uso con le liste e le altre strutture dati
- 7.6 La struttura dati Set
- 7.7 Caso di studio: elenco delle professioni censite nell'oggetto di classe Census
- 7.8 La struttura dati Map
- 7.9 Caso di studio: main() di Census multilingua con l'uso delle mappe
- 7.10 Caso di studio: refactoring del main di Census con metodi statici

8 - Ereditarietà

- 8.1 Creare un tipo da un altro tipo: introduzione concettuale all'ereditarietà
- 8.2 Regole di trasmissione ereditaria fra classi e override
- 8.3 Tipo formale e tipo concreto
- 8.4 Polimorfismo di oggetti, instanceof e casting
- 8.5 La classe Object e la differenza fra oggetti e primitivi
- 8.6 Caso di studio: una "fabbrica" di Date, il riutilizzo dello stesso oggetto
- 8.7 Caso di studio: School e le scelta delle entities
- 8.8 Caso di studio: la scrittura delle classi della gerarchia
- 8.9 I rapporti oltre la gerarchia: il rapporto d'uso
- 8.10 Caso di studio: l'oggetto aggregatore per School
- 8.11 Caso di studio: importazione delle entities di School da file
- 8.12 Conclusioni sul caso di studio School: rapporto di aggregazione ed esercizi

9 - Programmazione a oggetti avanzata

- 9.1 Riepilogo della programmazione a oggetti: ereditarietà, encapsulamento e polimorfismo
- 9.2 Classi astratte e loro uso
- 9.3 Interfacce: il tipo come contratto
- 9.4 Implementazione di interfacce multiple e aggiunta di funzionalità tramite interfacce
- 9.5 Metodi concreti nelle interfacce: default e static
- 9.6 Interfacce funzionali e lambda
- 9.7 Riepilogo della programmazione a oggetti avanzata
- 9.8 Caso di studio: il progetto School con uso di lambda e interfacce
- 9.9 Approfondimenti: definizione di classi e interfacce generiche
- 9.10 Un caso di studio: la struttura dati Deck < X >

10 - I Database e MySQL

- 10.1 Introduzione
- 10.2 Elementi di un database SQL
- 10.3 SQL, le sue estensioni e i DBMS
- 10.4 MySQL Server
- 10.5 Creazione di un database
- 10.6 Manipolare i dati: inserimento, modifica, cancellazione
- 10.7 Interrogazioni su una tabella
- 10.8 Raggruppamento (GROUP BY) e funzioni di gruppo
- 10.9 Relazioni e query su più tabelle: il concetto di JOIN
- 10.10 L'integrità referenziale e il problema degli orfani
- 10.11 Il rapporto 1-1 e l'ereditarietà in MySQL. Forma implicita ed esplicita del Join e concetto di Inner Join
- 10.12 I rapporti 1-n
- 10.13 I self join e gli alias di tabella
- 10.14 Il rapporto n-n
- 10.15 View in MySQL

11 - Estensioni di SQL in MySQL - Stored Procedures, Stored Functions e Triggers

12 - JDBC

- 12.1 Introduzione a JDBC
- 12.2 Connection
- 12.3 Statement
- 12.4 ResultSet
- 12.5 Un primo esempio completo
- 12.6 Queries parametriche in JDBC e PreparedStatement
- 12.7 Object Relational Mapping (ORM) e DAO
- 12.8 ORM per entità complesse: il caso 1-n

13 - JPA

- 13.1 Introduzione a JPA
- 13.2 Una implementazione di JPA - Eclipse Link e sua configurazione
- 13.3 Mappatura e operazioni di base su una entità semplice. Entità gestite e non gestite
- 13.4 Mappatura uno a molti e politica di propagazione.
- 13.5 Mappatura di entità padre ed entità figlio
- 13.6 Operazioni di base con JPQL
- 13.7 Una business logic da completare in JPA
- 13.8 Approfondimenti

14 - Design Pattern

- 14.1 Concetto di Pattern
- 14.2 Factory
- 14.3 Singleton
- 14.4 Adapter
- 14.5 Proxy
- 14.6 Facade
- 14.7 Model View Controller (MVC)

15 - Il linguaggio HTML

- 15.1 Introduzione ad HTML
- 15.2 Basi del linguaggio
- 15.3 Un primo esempio
- 15.4 h1
- 15.5 p
- 15.6 form
- 15.7 input
- 15.8 select
- 15.9 a

15.10 Approfondimenti

16 - CSS

- 16.1 Introduzione a CSS
- 16.2 Basi del linguaggio e uso di CSS in HTML
- 16.3 Regole di stile per il resto
- 16.4 Regole per margini, imbottiture e dimensioni (margin, padding, height, width)
- 16.5 Regole per l'allineamento e la visibilità degli elementi
- 16.6 Propagazione delle regole
- 16.7 Selettori di tag
- 16.8 Selettori di classe
- 16.9 Selettori per ID
- 16.10 Selettori per discendenza
- 16.11 Selettori per attributo
- 16.12 Un esempio pratico di HTML e CSS - ordinare una pizza
- 16.13 Il problema della responsività
- 16.14 Approfondimenti

17 - Bootstrap

- 17.1 Introduzione
- 17.2 Installazione
- 17.3 Grid System ed elementi di uso comune
- 17.4 Colori e loro significato in Bootstrap
- 17.5 Cards
- 17.6 Barre di navigazione
- 17.7 Approfondimenti

18 - Javascript

- 18.1 Introduzione
- 18.2 Variabili in Javascript
- 18.3 Tipi primitivi in Javascript
- 18.4 Il tipo function
- 18.5 Gli oggetti in Javascript
- 18.6 Vettori
- 18.7 Condizioni
- 18.8 Selezione
- 18.9 Iterazione
- 18.10 Ereditarietà classica
- 18.11 Il DOM
- 18.12 Metodi di uso comune del DOM
- 18.13 Proprietà comuni degli elementi del DOM
- 18.14 Eventi in Javascript
- 18.15 Eventi comuni
- 18.16 Un primo esempio
- 18.17 jQuery
- 18.18 L'oggetto \$, \$(document).ready e la definizione di eventi in jQuery
- 18.19 Metodi di manipolazione del DOM in jQuery

19 - XML

- 19.1 Introduzione
- 19.2 XML-Schema e il concetto di validità di un XML

20 - Servlet e Web App

- 20.1 Introduzione ad HTTP
- 20.2 Elementi del protocollo
- 20.3 Servlet
- 20.4 I gestori di verbo e la produzione della response nelle servlet
- 20.5 Uso basilare dell'oggetto HttpServletRequest: i parametri
- 20.6 Le form e le request
- 20.7 Java Server Pages
- 20.8 Struttura generale di una Web Application MVC Front Controller
- 20.9 Un esempio di web application CRUD: CensusWeb
- 20.10 JSTL
- 20.11 Session

21 - Web Service

- 21.1 Il concetto di Web Service - due definizioni
- 21.2 Introduzione a REST
- 21.3 Manipolazione delle risorse e i verbi di HTTP
- 21.4 Un esempio di Web-Service REST per una risorsa semplice
- 21.5 Approfondimenti

22 - Spring

- 22.1 Introduzione a Spring
- 22.2 Installazione e configurazione con Maven
- 22.3 Step nell'avvio di un progetto web SPRINGMVC
- 22.4 Esempio di una request su un DWP SpringMVC
- 22.5 Dependency e Dependency Injection
- 22.6 Passaggio di attributi alle viste: Model in Spring MVC
- 22.7 @RestController e la scrittura di un primo web service REST in Spring
- 22.8 Form in Spring MVC

23 - AJAX

- 23.1 Definizione di AJAX
- 23.2 Un caso concreto - Stock Market

23.3 Il cliente Javascript di Stock Market
23.4 Acquisto di una azione - AJAX senza JSON

1 - Basi della programmazione

1.1 Concetto di programma e introduzione informale a Java

Un programma può essere visto in vari modi. Una prima definizione è quella di *insieme di istruzioni*, dove una *istruzione* è un comando dato alla macchina (il computer). In questo senso un programma è solo una lista di istruzioni.

Una seconda definizione è quella di *traduzione di un algoritmo* in un linguaggio comprensibile alla macchina. Un *algoritmo* è un elenco di passi ordinati per arrivare a un risultato, e il linguaggio che usiamo per farci capire dalla macchina è Java, nel nostro caso (ma non sarà solo Java).

Una terza definizione è quella di una *funzione*, in senso matematico, che trasformi degli input in output: $O = P(I)$, dove O sono gli output, I sono gli input e P è il programma.

Notiamo che O ed I sono insiemi, e potrebbero essere molto vasti. Un input potrebbe essere l'elenco delle fatture emesse in un anno da un professionista, e l'output potrebbero essere le tasse da pagare.

Per finire, e questo sarà molto importante in seguito, possiamo vedere un programma come un *motore* composto di pezzi interconnessi pensati per lavorare assieme per produrre un risultato. Questa definizione in realtà specifica il *come* viene realizzato un programma, più che il cosa sia davvero, ma possiamo cominciare a rifletterci.

In tutti i casi, dovremo confrontarci con un linguaggio di programmazione. Un linguaggio di programmazione è "una lingua per parlare con la macchina", un linguaggio che le macchine, che sono estremamente limitate, possano capire.

I linguaggi di programmazione sono tutti strettamente formali. Siamo sempre in grado di dire se una "frase" in un linguaggio di programmazione ha senso o meno, mentre lo stesso non vale per i linguaggi naturali.

Questo ci costringerà a essere estremamente precisi nell'esprimerci. Un punto in più o in meno significherà il non funzionamento del programma.

Noi scriveremo i nostri programmi in Java. Java è una famiglia di linguaggi di programmazione (ce ne sono varie versioni, noi ci concentreremo sulle versioni dalla 8 in poi) che ricopre una posizione dominante sul mercato, ed è lo standard de facto per le applicazioni su larga scala.

1.2 Codice sorgente (.java) e codice pseudo compilato (.class)

E' codice sorgente tutto quello che vediamo nell'editor di Eclipse. Le righe package, classe, public static void main ecc..., che l'insegnante mostra durante l'introduzione a Java, sono tutte codice sorgente.

Esso non viene eseguito direttamente, ma deve prima essere compilato (pseudo-compilato nel caso di Java, come vedremo negli approfondimenti), per essere eseguito. Quello che scrivete non è quello che arriva al processore, ma è qualcosa che ci si avvicina abbastanza. Un ponte fra programmatore e macchina.

I files .java sono file di codice sorgente, e vengono letti dal programmatore. I file .class sono file compilati e vengono letti (ed eseguiti) dalla macchina. Dando "play" su Eclipse i file Java vengono compilati ("tradotti") in .class, che vengono eseguiti. Eclipse rende trasparente questo processo.

1.3 Blocco di codice

Ogni file di codice sorgente è diviso in blocchi di codice.
Un blocco di codice è un'area del codice sorgente delimitata da {} (parentesi graffe, ottenibili con shift-alt gr-[e shift-alt gr-] rispettivamente, sulle tastiere).

{ apre un blocco di codice, } lo chiude. Un blocco di codice può contenere istruzioni o altri blocchi di codice. Ogni blocco di codice definisce uno scope, vale a dire uno spazio a parte. Vedremo in seguito che questo avrà ripercussioni importanti.

Riportiamo un esempio:

```
{  
    int v = 5;  
    System.out.println(v);  
}
```

Questo blocco di codice stampa il valore 5, ed è costituito da due istruzioni, una di assegnamento e una di stampa.

1.4 Istruzioni

Riconosciamo tre forme principali di istruzione:

- la *dichiarazione* di una *variabile*
- l' *assegnamento* che imposta il *valore* (o *contenuto*) di una variabile.
- la *chiamata a metodo*, che esegue un sottoprogramma.

E queste forme possono combinarsi fra loro, come vedremo a breve. Le istruzioni sono racchiuse in blocchi di codice o separate fra loro dal punto e virgola (;).

1.5 Variabile

Una *variabile* è un indicatore verso un'area di memoria a cui abbiamo dato un *nome* e un *tipo*, e conterrà un valore del tipo corrispondente. Possiamo immaginarla come una scatola con un contenuto.

Il contenuto può cambiare col tempo (da cui il nome "variabile") ed essere molto complesso, ma partiamo dalle basi: per usare una variabile nel mio programma devo prima *dichiararla*.

Una variabile è utilizzabile direttamente nello scope (blocco) in cui è stata dichiarata, e in tutti gli scope contenuti nello scope di dichiarazione, secondo un meccanismo di "scatole cinesi". In pratica, se lo scope s1 contiene la variabile a, ed s1 contiene s2, anche s2 "vedrà" la variabile a. Questo sarà molto importante quando parleremo di alcune variabili particolari che prendono il nome di proprietà. Vediamo qualche esempio di dichiarazione:

```
int v;  
// v dovrà contenere un valore intero  
double d;  
// d potrà contenere un numero con la virgola  
String s;  
// s potrà contenere un testo qualunque
```

Sto dichiarando (dicendo al programma) che terrò in memoria 3 simboli, rispettivamente di tipo int, double e String, con nomi v, d ed s. int, double e String sono i tipi delle variabili. v, d ed s i nomi che ho scelto. Le istruzioni sopra riportate sono dette di dichiarazione.

Una volta dichiarate, le variabili possono ricevere un valore, o "essere assegnate", o "riempite":

```
double pi;  
pi = 3.14
```

Le due istruzioni sopra sono rispettivamente di dichiarazione ed assegnamento.

Alla prima riga ho dichiarato una variabile (una scatola, un'area di memoria) di nome pi, e ho detto che dovrà contenere un numero con la virgola (double).

Alla seconda ho copiato il valore 3.14 nella posizione di memoria che corrisponde alla variabile pi.

Il primo assegnamento di una variabile viene detto inizializzazione. Una variabile deve essere inizializzata prima di essere usata.

Spesso si usa una forma contratta per eseguire dichiarazione e assegnamento allo stesso tempo:

```
double pi = 3.14;
```

Ci sono delle regole, poche ma sacre. La prima è che non posso avere due variabili con lo stesso nome nello stesso scope:

```
{  
    int v = 1;  
    int v = 2;  
}
```

Darà errore. Lo stesso accadrà in questo caso:

```
{  
    int v = 1;  
    {  
        int v = 2;  
    }  
}
```

In quanto lo scope interno riceve già la variabile v da quello esterno, e non può dichiararla una seconda volta.

La seconda regola è che il tipo di una variabile non cambia una volta dichiarato. Java è "a tipizzazione statica". Il suo valore può cambiare (il contenuto), ma mai la forma della scatola. Dichiarato int v, v non potrà mai contenere altro che interi.

1.6 Concordanza dei tipi e regole di assegnamento

C'è un'altra regola che riguarda le variabili, e Java in generale, ed è quella che rende la programmazione Java estremamente sicura ma anche più "rigida" di quanto alcuni vorrebbero. Viene detta concordanza dei tipi. Vediamola con un esempio:

```
int v = 3.14;  
//errore!
```

Quello che abbiamo appena visto è detto Type Mismatch ed è un errore imperdonabile in Java. Stiamo cercando di mettere un numero con la virgola in una "scatola" che può contenere solo interi.

Questo ci porta a enunciare una regola generale per la "concordanza dei tipi" in fase di assegnamento:

Se scrivo a = b; :

- a deve essere una variabile (devo avere un contenitore in cui mettere b)
- b deve essere una espressione dello stesso tipo di a o di un tipo compatibile

Vediamo una serie di esempi:

```
//ESEMPI ERRATI  
6 = 7;  
// 6 non è una variabile  
int a = 7.1;  
//di nuovo: un decimale dentro un intero. Tipi non compatibili  
int h = "Ferdinando";  
// una String dentro un intero. Tipi non compatibili  
String s = 7;  
//Un intero dentro una stringa. Tipi non compatibili  
//ESEMPI VALIDI  
int a = 6;  
double b = 7;  
//funziona. Un intero è un caso particolare dei decimali  
String s = "7";  
//le virgolette rendono il valore "7" un testo.  
double c = a+2;  
//funziona, e per la prima volta il valore di una variabile è determinata da un'altra
```

L'ultimo esempio presenta una espressione calcolata (a+2), invece di un valore fisso. E' un caso comune. Vedremo bene cosa si intende con espressione a brevissimo.

1.7 Chiamata a metodo

La chiamata a metodo è la terza forma principale di istruzione:

```
System.out.println(v);
System.out.println(3);
System.out.println("Hello world");
```

La chiamata a metodo è nella forma : m(a₁, a₂, a₃,... a_n), dove m è il nome del *metodo* (il sottoprogramma che stiamo eseguendo) e a sono gli argomenti, o parametri del metodo, vale a dire le informazioni che gli vengono fornite e su cui lavorerà.

```
System.out.println(v);
// Richiama (esegui) il sottoprogramma System.out.println e "gli passa" il valore della variabile v.Se v = "Pippo", verrà stampata la parola
"Pippo".
```

1.8 Analisi di un primo programma Java

Partiamo dal seguente programma di prova:

```
package first;
public class App
{
    public static void main(String[] args)
    {
        String msg = "Hello world";
        System.out.println(msg);
    }
}
```

La prima riga identifica il package. Un package è una cartella posta nell'albero (sottocartelle) del codice, figlia o discendente della cartella src. E' un contenitore di classi. Le classi poste nello stesso package sono "vicine" e godono di privilegi particolari le une verso le altre, che vedremo in seguito.

La classe è il primo elemento al di sotto del package. In questo caso App è una *classe di avvio*. Una classe di avvio è una classe contenente uno speciale metodo (sottoprogramma) main che vedremo a breve. Corrisponde a un programma che può essere eseguito direttamente.

Non tutte le classi sono classi di avvio. Esiste un secondo uso per le classi, molto più comune, che vedremo in seguito.

Il metodo main citato prima, presente nelle classi di avvio, è il punto di partenza del programma. L'esecuzione comincia da main che è un metodo, vale a dire un sottoprogramma, particolare.

Ad ora non spiegheremo cosa vogliono dire public static void main(String[] args). Ci basti dire che main è il punto di partenza del programma, e corrisponde a un blocco di codice (main) dentro un altro blocco di codice (la classe App).

Dentro main abbiamo String msg = "Hello World";. E' la dichiarazione della variabile msg all'interno del blocco di codice del metodo main (il corpo del nostro programma, in sostanza), e sua inizializzazione al valore "Hello World".

Successivamente abbiamo la chiamata a metodo System.out.println(), passando come parametro msg. Come risultato, viene stampato Hello World.

Notiamo che System.out.println(msg) deve necessariamente essere eseguito come seconda riga, non come prima, del programma main. Il simbolo msg esiste DOPO la prima riga del main, non prima.

Questo concetto è quello di "sequenza" - l'ordine delle istruzioni conta. La sequenza è uno dei tre principi della programmazione classica. Gli altri due sono selezione e iterazione.

1.9 Espressioni

Introduciamo il concetto di espressione partendo dal seguente codice:

```
int base = 4;  
int altezza = 6;  
int area = base * altezza;
```

La variabile area conterrà il valore 24, vale a dire il risultato della *espressione* `base*altezza`.

Una espressione è tutto ciò che restituisce un valore. Somiglia molto alle espressioni algebriche studiate a scuola. L'espressione viene sempre valutata. Valutarla significa dare un valore all'espressione, cioè sostituire le variabili presenti (SE presenti) coi loro valori e svolgere i calcoli.

Tornando all'esempio di prima, l'espressione è **base * altezza** ; , e al momento dell'esecuzione dell'istruzione il programma sostituirà le variabili coi rispettivi valori, e quindi diventerà: **4*6** .

Il simbolo * si legge "moltiplicazione" ed è un operatore binario, vale a dire una operazione definita su due valori per volta.

Come tutto in Java, anche le espressioni hanno un tipo, ma Java è relativamente flessibile nel gestirlo.

L'espressione `base * altezza`, ad esempio, è il prodotto di due interi e di conseguenza restituisce un intero. Lo possiamo mettere un intero in una variabile ("scatola") di tipo intero.

In questo caso viene rispettata la regola di concordanza dei tipi: un intero a sinistra, un intero a destra, ma vediamo anche un altro esempio:

```
int base = 4;  
int altezza = 6;  
double area = base * altezza;
```

Stranamente, anche questo esempio funziona. Un intero è un caso particolare di numero con la virgola (è un numero con la virgola che ha solo 0 dopo la virgola), e di conseguenza posso mettere una espressione intera in una variabile ("scatola") di tipo double.

Vediamo il caso opposto:

```
double base = 4;  
double altezza = 6;  
int area = base * altezza;
```

Questo non verrà accettato dal *compilatore*, vale a dire dal programma usato da Eclipse per tradurre i .java in .class.

Le due variabili base e altezza contengono valori interi ma sono dichiarate come double. Potrebbero contenere valori non interi, di conseguenza potrei non essere in grado di mettere il risultato nella variabile area. Il compilatore se ne accorge e mi impedisce di continuare.

Vediamo qualche esempio di espressione:

```
int areacasa = base * altezza;  
// areacasa è il prodotto di due interi, quindi sarà un intero e posso memorizzarlo in una variabile di tipo int  
//double costocasa = areacasa * costomq;  
// In questo caso sto moltiplicando un numero con la virgola per un intero. Il risultato potrebbe avere la virgola, quindi devo memorizzarlo in  
una variabile di tipo double.  
String annuncio = "Casa sita in "+via+" per metri quadri "+areacasa+" al costo di "+costocasa+" euro";
```

L'ultima è una novità e merita una spiegazione a parte, in quanto non restituisce (non produce) né un int né un double, ma una String. In questo caso l'operatore + non indica la somma algebrica, ma la CONCATENAZIONE.

Le variabili vengono sostituite e accodate ai pezzi di stringa. Il risultato dell'espressione sarà: "Casa sita in Via dei Frassini per metri quadri 24 al costo di 7199,76 euro"

L'operatore + usato per sommare stringhe tra loro, o per sommare stringhe ad altri tipi di valore restituisce SEMPRE una stringa.

Vale la pena aggiungere una piccola nota su String.

String a differenza degli altri tipi di scrive con la lettera maiuscola. Non è un caso – String NON è un tipo primitivo, vale a dire, uno dei tipi base di Java da cui tutto il resto è composto. Java tuttavia lo tratta quasi come se lo fosse. Approfondiremo in seguito: per adesso notiamo questa differenza.

1.10 Il tipo boolean e le condizioni

I tipi int, double e String sono intuitivi. Sono rispettivamente numeri interi, numeri con la virgola (potenzialmente) e testi. C'è un tipo meno intuitivo ma estremamente utile nella pratica, il boolean.
Un boolean può avere solo uno di due valori: true (vero) o false (falso).

Questo risulta estremamente importante nel definire il concetto di *condizione*.

In programmazione la distinzione fra vero e falso è assoluta. Non esistono ambiguità, non c'è margine di interpretazione. Noi vogliamo essere in grado di dire quando una condizione è vera o falsa.
Ma cosa è, formalmente, una condizione?

Una condizione è una espressione che restituisce (calcola) un boolean.

```
int eta = 18;  
boolean maggiorenne = eta > 18;
```

maggiorenne è una variabile di tipo boolean. Può essere solo true o false. eta > 18 è una espressione booleana, vale a dire una condizione. Ma analizziamola bene:

eta > 18;

Questa è comunque una espressione, composta da :

eta: una variabile di tipo intero

>: un operatore binario booleano, vale a dire, un operatore che lavora su due ingressi (in questo caso eta e 18) e restituisce vero o falso
18: un *letterale*. Il tipo più semplice di espressione. Non necessita di calcoli, è uguale a se stesso in fase di valutazione dell'espressione.

Non fatevi ingannare dal fatto che eta e 18 siano interi. L'espressione eta > 18; non restituisce un intero: restituisce un boolean. L'operatore > lavora su due numeri e produce un boolean.

Il tipo di una espressione NON E' per forza il tipo delle sue componenti.

L'espressione eta > 18 è composta da eta, 18 e un operatore. Partiamo da due interi e un operatore, ne esce un boolean, vale a dire vero o falso (true o false). Non esistono ambiguità.

Lo ripetiamo: una espressione che produce un boolean si dice condizione.

1.11 Condizioni composte

Studiamo qualche esempio concreto, partendo da queste variabili:

```
//partiamo da questi dati
int eta = 40;
int statura = 170;
int peso = 70;
String nome = "Friedrich";
String genere = "M";
String professione = "programmatore";
boolean celibe = true;
```

Notiamo che "celibe" è già di suo una condizione (un valore booleano). Le altre variabili NON possono essere usate come condizioni. Ora poniamo delle domande sulla persona descritta da quelle variabili:

- è alto?
- è normopeso?
- è un programmatore ?
- è un uomo celibe ?
- è un programmatore di genere maschile ?
- è un programmatore senior ?
- è un programmatore o un insegnante normopeso ?

A tutte queste domande si risponde tramite condizioni. Partiamo dalla prima: è alto?

Un uomo italiano è sopra la media quando supera i 175 cm, quindi la condizione sarebbe:

```
boolean alto = statura > 175;
//alto = false
```

Ma se fosse una donna? Una donna di nome Friedrich potrebbe essere insolita, ma non si sa mai. Una donna si considera alta sopra i 165 cm e noi dobbiamo tenere conto di due variabili a questo punto: il genere e la statura.

Devo scomporre la condizione in due casi: è un uomo alto più di 175 cm o una donna alta più di 165. Sono rispettivamente:

```
(genere.equals("M") && statura > 175);
(genere.equals("F") && statura > 165);
```

(genere.equals("M") && statura > 175); viene detta condizione composta, vale a dire ricavata a partire da un numero arbitrario di altre condizioni.

&& si legge *and* ed è un operatore logico binario, o operatore booleano binario. Collega due valori booleani e produce un altro valore booleano. Vale a dire, collega due condizioni A e B.

A && B è vero se sia A che B sono veri. La condizione di sopra si legge "la stringa genere è uguale a M e allo stesso tempo la statura è maggiore di 70".

Scomponiamola ulteriormente:

A : genere.equals("M"). equals è un metodo che restituisce un boolean. La scritta genere.equals("M") verrà valutata e trasformata in true o false. Non ci sono altre possibilità.

B : statura > 175, statura è un intero, 175 è un intero, > è un operatore, che mette a confronto due numeri e restituisce un boolean. Quindi statura > 175 alla fine diventerà true o false.

Quindi avremo A && B , vale a dire, boolean && boolean, come è necessario che sia, perché && lavora SOLO su boolean, vale a dire, solo su condizioni.

Possiamo scomporla ancora:

```
boolean uomo = gender.equals("M");
boolean donna = gender.equals("F");
boolean altouomo = (uomo && statura > 175);
```

```
boolean altadonna = (donna && statura> 165);
boolean altoa = altouomo || altadonna;
```

Cosa vuol dire `||` ?
Si legge *or*, ed è un operatore binario logico, o operatore binario booleano.

`(A || B)` è vero se almeno uno fra A e B è vero. `altoa` è vero se è vera `altouomo` o se è vera `altadonna`.
Potremmo anche scriverlo come:

```
boolean altoa =
(genere.equals("M") && statura>175)
||
(genere.equals("F") && statura>165);
```

Vale a dire (maschio AND sopra175) OR (femmina AND sopra165);

Passiamo alla seconda condizione: è normopeso?
Immaginiamo che sia normopeso se altezza-peso ≥ 95 , quale che sia il genere.

```
boolean normopeso = (statura-peso)>=95;
```

`statura - peso` -> int -> int
`int >= 95` -> int \geq int -> boolean

`\geq` è un operatore logico binario che confronta due numeri e restituisce true se $a \geq b$, false altrimenti.

E se volessimo differenziare fra uomo e donna? Supponiamo che un uomo sia normopeso se altezza - peso ≥ 90 , mentre per una donna scegliamo 95. Un uomo a parità di statura tende a pesare di più di una donna.

```
boolean normopeso =
(genere.equals("M") && ((statura-peso)>=90))
||
(genere.equals("F") && ((statura-peso)>=95));
```

E' un programmatore? Questa è semplice:

```
boolean programmatore = professione.equals("programmatore");
```

Ricordiamo che `equals` è un metodo che restituisce un boolean. Vuol dire che il programma sostituirà `professione.equals("programmatore")` con true o false (in questo caso, true);

E' un programmatore senior? Potremmo dire che è un programmatore senior se si parla di un programmatore sopra i 30 anni.

```
boolean seniorprogrammer = professione.equals("programmatore") && eta>30;
```

Le condizioni sono `A = professione.equals("programmatore")` e `B = eta > 30`; `A && B` → devono essere vere entrambe

E' un programmatore o un insegnante normopeso?
Utilizziamo le variabili di prima!

```
boolean res = normopeso && (professione.equals("insegnante") || professione.equals("programmatore"));
```

Ho riutilizzato una condizione calcolata prima (`normopeso`) per evitare di riscriverla. Ma potremmo anche scriverla come:

```
boolean insegnante = professione.equals("insegnante");
boolean programmatore = professione.equals("programmatore");
```

```
boolean res = normopeso && ( insegnante || programmatore);
```

Vale a dire: è vera A (è normopeso), ed è vera almeno una fra insegnante e programmatore.

E se volessimo esprimere la negazione di una condizione? Abbiamo l'operatore `!`, che si legge "not". `!alto`, letteralmente "non alto", in Java. Se `alto` è true, `!alto` è false, e viceversa. `!` è un operatore booleano unario, che lavora cioè su un solo valore di verità per volta.

1.12 Selezione: scegliere fra più cammini possibili e il costrutto if

I programmi visto fino ad ora sono stati strettamente sequenziali. Le righe sono state eseguite nell'ordine in cui sono state scritte, in maniera rigida. Il programma non poteva eseguire altre istruzioni se non quelle, e questo è limitante.

In quasi tutti i casi vorremo che il programma cambi comportamento a seconda dei dati che l'utente inserisce, o delle *condizioni* che essi verificano. Dovremo poter selezionare delle istruzioni che verranno eseguite o no a seconda del valore delle condizioni (vero / falso).

La prima e più basilare forma di selezione è il costrutto if. E' anche in assoluto il più potente e l'unico di cui avremo veramente bisogno, visto che le altre sono forme di comodo. if lavora valutando una *condizione*, vale a dire una espressione booleana che, quindi, potrà essere solo vera o falsa.

La sua forma generale è la seguente:

```
if(condizione)
{
    //istruzioni se la condizione è vera
}
else
{
    //istruzioni se la condizione è falsa
}
```

Il primo blocco, quello dopo if(condizione), prende il nome di blocco if, o di blocco-vero, e viene eseguito quando la condizione è vera. E' la sola parte obbligatoria del costrutto. Il secondo blocco, quello dopo l'else, viene eseguito se la condizione è falsa, e prende il nome di blocco-else o blocco-falso.

Per ovvie ragioni, verrà eseguito solo l'uno o l'altro.

Ora vediamo una serie di condizioni e di relativi if o if-else:

```
if(statura>180)
    System.out.println("Alto");
//solo blocco if, niente else, e non servono le graffe perchè il blocco if si riduce a una riga
//che succede se non è alto? niente. Non verrà eseguita nessuna istruzione e sarà come se l'intero if
//non esistesse
if(statura>180)
    System.out.println("Alto");
else
    System.out.println("Non alto");
//blocco if e blocco else. Uno dei due verrà sicuramente eseguito. Anche in questo caso non servono le graffe
//si tratta di una sola istruzione per blocco
if(statura>180)
    System.out.println("Alto");
else
{
    System.out.println("Non alto ma...");
    if(statura>175)
        System.out.println("Sopra la media");
    else
        System.out.println("Nella media o sotto");
}
//blocco if e blocco else. Il blocco else contiene diverse istruzioni e quindi è necessario un blocco di codice.
//notiamo che se siamo nell'else, il primo if è falso. Quindi alla seconda riga dell'else noi siamo sicuri
//che la statura sarà sotto il metro e ottanta. Se stiamo eseguendo la terza riga dell'else, la sua
//statura è compresa fra 175 e 180.
```

Come abbiamo appena visto gli if-else si possono *innestare*. Possiamo avere un if dentro un if, senza problemi, ma dovremo ricordarci che il *luogo* in cui siamo rifletterà la *condizione* in cui ci troviamo. Se siamo nell'else siamo sicuramente sotto o pari a 180 cm, nell'ultimo esempio.

Ricordiamo anche, ora come prima, che una condizione è solo una espressione booleana, e posso salvarla in una variabile e usarla come condizione nell'if:

```
boolean over180 = statura> 180;
boolean over175 = statura> 175;
if(over180)
    System.out.println("Alto");
else
{
    System.out.println("Non alto ma...");
    if(over175)
        System.out.println("Sopra la media");
    else
        System.out.println("Nella media o sotto");
}
```

1.13 Scegliere un valore fra due - l'operatore ternario

Molto spesso ci troveremo nella condizione di dover scegliere che valore assegnare a una variabile sulla base di una scelta binaria. La variabile v potrà avere uno di due valori, a o b, a seconda di una condizione c. In questi casi, la soluzione ideale è detta operatore ternario, che ha questa forma:

```
variabile = condizione ? valore_se_vera : valore_se_falsa;  
v = c ? a : b;
```

Dove a e b **devono** avere lo stesso tipo di v per la regola di concordanza dei tipi. Si parla comunque di un assegnamento. v, a e b sono tutti dello stesso tipo, mentre c è una condizione, e potrà essere *solo* un boolean.

Tornando all'esempio di prima:

```
String msg = statura > 180 ? "Alto" : "Non alto";  
System.out.println(msg);
```

Sia "Alto" che "Non alto" sono String, così come la variabile msg. statura è un int, 180 è un int, ma statura > 180 restituisce un valore vero o falso, vale a dire un boolean. Quindi questo ternario ha la forma: String = boolean ? String : String, e rispetta la definizione.

Posso lavorare anche con condizioni complesse. Un uomo è considerato alto sopra il metro e ottanta, una donna sopra il metro e settanta. Per capire se un individuo è alto avrei bisogno di tenere conto anche del genere:

```
boolean alto = gender.equals("M") && statura > 180;  
boolean atla = gender.equals("F") && statura > 170;  
String msg = alto || atla ? "Alto/a" : "Non alto/a";  
//Avrei potuto scriverlo anche con gli if  
if(alto || atla)  
    msg = "Alto/a";  
else  
    msg = "Non alto/a";
```

In generale, il ternario è molto espressivo e lo preferiremo sempre le scelte del tipo "bianco o nero" per dare il valore a una variabile. Quando si tratterà di eseguire blocchi di codice, invece, l'if sarà la nostra soluzione di riferimento.

1.14 Gestire una casistica discreta: il costrutto switch

In alcuni casi ci interessa gestire un numero elevato di casi diversi relativi ai valori di una singola variabile. Supponiamo di dover calcolare il costo del biglietto di un museo. Gli studenti pagano otto euro, gli insegnanti sette, i pensionati cinque, tutti gli altri dieci. Potremmo affrontarlo con un if:

```
price = 10 ;
if(profession.equals("student"))
    price = 8;
if(profession.equals("teacher"))
    price = 7;
if(profession.equals("retired"))
    price = 5;
```

Ma è lungo e ripetitivo. In questo caso è preferibile usare un costrutto apposito, lo switch, che serve a definire casi diversi relativamente al valore di una espressione:

```
switch(profession)
{
    case "student":
        price = 8;
    break;
    case "teacher":
        price = 7;
    break;
    case "retired":
        price = 5;
    break;
    default:
        price = 10;
}
```

Vengono valutati i casi della variabile profession. Una volta rilevato il valore che ci interessa (ad esempio "student") verrà eseguito il blocco relativo, che può essere composto anche di più istruzioni senza graffe. E' comunque possibile metterle per isolare uno scope, un ambito ristretto di visibilità per le variabili di lavoro, ed è spesso consigliabile.

Il break alla fine dei vari case, ove presente, termina l'esecuzione dello switch e ci risparmia di valutare le altre condizioni (rivedremo il break nella sezione sui cicli). Se nessuno dei valori enumerati viene riconosciuto (in questo caso, profession non è student né teacher né retired) si passa al blocco (opzionale) default, che viene eseguito solo quando tutti gli altri casi non sono stati verificati.

C'è modo di concatenare più di un caso, ma è considerata cattiva prassi e non lo mostreremo in questa sede.

1.15 Iterazione: Introduzione al concetto di ciclo, do-while e while

Alcuni problemi possono essere risolti solo ciclando, *iterando*, ripetendo istruzioni o blocchi di codice.

Un esempio classico è la ripetizione di una richiesta all'utente fino ad ottenere una risposta congrua (ad esempio, un numero positivo). Un altro esempio è la gestione non di una, ma di una serie di variabili in una *struttura dati*. In entrambi i casi è necessario poter ripetere lo stesso codice (istruzione o blocco) un certo numero di volte, potenzialmente non noto a priori.

Una prima tipologia di ciclo è il do-while. Si tratta di un ciclo che può essere ripetuto da 1 a infinite volte, e che nella sua forma generale è scritto come segue:

```
do
{
    //istruzioni
}
while(condizione);
```

Il ciclo viene eseguito almeno una volta. Al termine di ogni esecuzione verrà valutata la condizione scritta nel while alla fine del ciclo. Questa condizione, detta *condizione di ripetizione*, se vera farà ripartire il ciclo, mentre se è falsa lo farà terminare, facendo passare il programma alla riga subito dopo il while. Vediamo come chiedere all'utente un numero strettamente positivo:

```
int n = 0;
do
{
    n = Integer.parseInt(keyboard.nextLine());
}
while(n <= 0);
```

Il do-while è un ciclo a esecuzione obbligatoria. Verrà eseguito sempre almeno una volta, ma non possiamo tipicamente prevedere quante volte sarà eseguito. Valgono per il do-while, così come per qualunque altro blocco, le regole relative allo scope delle variabili: se avessi dichiarato n dentro il ciclo non avrei potuto usarlo fuori.

Il ciclo while è affine al ciclo do-while, ma potrebbe anche non essere eseguito nemmeno una volta. La differenza rispetto al do-while è che la condizione viene controllata prima della prima esecuzione. Se la condizione di ripetizione è falsa il ciclo viene saltato completamente. Studiamo questo esempio relativo a un conto alla rovescia:

```
int n = Integer.parseInt(keyboard.nextLine());
while(n>0)
{
    n--;
    System.out.println(n);
}
```

Il corpo del ciclo è costituito da due istruzioni - la stampa di n e la sua diminuzione di 1. Supponendo che n all'inizio sia 10, stamperemo 10,9,8,7,6,5,4,3,2,1 e poi n diventerà 0, e usciremo dal ciclo (andremo alla riga 7 del programma). Se l'utente dovesse inserire -1 in n, invece, la condizione sarebbe falsa fin da principio e il ciclo verrebbe saltato completamente. Il ciclo while è quindi a esecuzione opzionale.

Vedremo che il ciclo while tornerà comodo per la lettura di files che potrebbero contenere o meno dei dati, e da cui potremmo leggere da 0 ad n righe - e in effetti il while verrà eseguito da 0 a n volte, una volta per ogni riga.

1.16 Ciclo for

In molti casi vogliamo modificare delle variabili a ogni *iterazione*, vale a dire, a ogni "giro" di un ciclo.

Lo abbiamo visto nel caso del conto alla rovescia: dovevamo decrementare n di uno a ogni iterazione, partendo da un valore iniziale, e continuare fin tanto che n è maggiore di zero. Il while ci permette di farlo, ma il for è un ciclo che nasce per questa tipologia di problema. La sua struttura è la seguente:

```
for(inizializzazione;condizione di ripetizione;aggiornamento)
{
    // corpo del for
}
```

Laddove :

- inizializzazione è un'istruzione, o una serie di istruzioni, da eseguire una sola volta, prima della prima esecuzione del for
- la condizione di ripetizione è esattamente analoga a quella del while: viene eseguita prima della prima esecuzione, e quindi anche il for è un ciclo ad esecuzione opzionale
- aggiornamento è una istruzione, o una serie di istruzioni, che servono tipicamente a modificare le variabili "gestite" dal for

Vediamo due esempi:

```
for(int i=0;i < 10 ;i++) System.out.println(i);
//inizializza i a 0
//proseguì fino a che i è <10;
//a ogni giro aumenta i
//stamperà i numeri da 0 a 9
for(int n=1, i=1; i*n <30; i++, n+=2) System.out.println(i*n);
//inizializza i a 1, n a 1
//proseguì fin tanto che il loro prodotto è inferiore a 30
// stampa n*i
// a ogni giro aumenta i di uno, n di 2. L'aggiornamento viene eseguito DOPO l'esecuzione
// stamperà 1,6,15, 28. Poi i*n sarà> 30 e usciremo dal ciclo
```

Vale la pena di notare che nessuna delle parti del for è obbligatoria. Il seguente for è perfettamente valido:

```
for();
//un ciclo infinito e inutile, a cui manca perfino il corpo, ma da un punto di vista formale comunque corretto
```

Potrebbe mancarne anche solo una:

```
int n = 5;
for(;n<10;n++)
    System.out.println(n);
//Sto usando una variabile esterna al for. Posso farlo: il blocco del for è interno a quello del metodo
```

Potrebbe mancare anche l'aggiornamento:

```
int n = 0;
for(;n <100;)
{
    n+=15;
    System.out.println(n);
}
```

E' piuttosto insolito, e non ci capiterà nella nostra pratica, ma è bene sapere che alcune realtà utilizzano cicli for più complessi o non standard. Resta un'ultima domanda: abbiamo modo di eliminare anche la condizione di ripetizione?

1.17 Saltare una iterazione e uscire prima dal ciclo: continue e break

Java mette a disposizione due parole chiave che permettono di alterare l'esecuzione di un ciclo, saltando un "giro" in un caso e terminando il ciclo anticipatamente in un altro. Queste sono rispettivamente `continue` e `break`.

La parola chiave `continue` indica al programma di passare al prossimo giro del ciclo, quale che sia il ciclo in questione. Come per `break`, possiamo usare `continue` in qualunque tipo di ciclo. Supponiamo di voler sommare solo i numeri positivi fra i primi dieci inseriti dall'utente:

```
int sum = 0;
for(int i = 0 ; i < 10 ; i++)
{
    int n = Integer.parseInt(keyboard.nextLine());
    if(n<=0) continue;
    sum+=n;
}
```

La parola chiave `continue` nel ciclo ci fa saltare alla successiva iterazione. Supponendo di incontrare un numero negativo per `i = 0`, `continue` terminerà l'iterazione alla seconda riga del `for`, non farà eseguire la terza, e andrà in aggiornamento, quindi porterà `i` a 1 e rieseguirà il `for`.

In tutti i casi, inseriti 10 numeri il ciclo terminerà normalmente. `Continue` quindi si limita a farci "saltare un giro", a partire dalla riga a cui è stato scritto.

L'uso di `continue` è scoraggiato da alcuni, perché rende il `for` più difficile da seguire, ma in qualche caso può alleggerire il codice e renderlo più leggibile. Lo sviluppatore dovrà decidere volta per volta. In generale potete immaginare `continue` come "salta un elemento" nello scorrimento di un insieme di dati.

Notiamo che `continue` non è *mai* obbligatoria, può essere al massimo comoda. Possiamo sempre saltare una iterazione applicando le condizioni giuste e selezionando di conseguenza quello che vorremo o non vorremo eseguire.

Di natura diversa è la parola chiave `break`. `break` interrompe il ciclo a prescindere dalla condizione di ripetizione. Possiamo usarlo, e spesso lo faremo, per interrompere una ricerca quando abbiamo trovato il nostro risultato. Supponiamo di avere una lista con 100.000 stature, e di cercare qualcuno sopra il metro e novanta. Una volta trovato non avrebbe molto senso continuare nel ciclo, e a quel punto la soluzione più naturale sarebbe interromperlo prima:

```
int i;
for(i = 0 ; i < 100000 ; i++)
{
    //chiedo una statua
    int n = Integer.parseInt(keyboard.nextLine());
    if(n> 190) break;
    //break termina prima il ciclo
}
System.out.println("Trovato alla posizione "+i);
```

Notiamo che possiamo scriverlo anche senza `break`:

```
int i;
int n=0;
for(i = 0 ; i < 100000 && n < 190 ; i++)
{
    //chiedo una statua
    int n = Integer.parseInt(keyboard.nextLine());
    //break termina prima il ciclo
}
System.out.println("Trovato alla posizione "+i);
```

Lo abbiamo ottenuto lavorando sulla condizione di ripetizione, ma è più "scomodo" e meno elegante.

Usando `break` potremmo anche togliere la condizione di ripetizione in generale, e limitarci a terminare il ciclo al verificare di una condizione, ma sarebbe solo un esercizio di stile. In generale, useremo `break` per interrompere la ricerca su di un insieme, come vedremo in seguito, quando avremo trovato quello che cerchiamo.

2 - Introduzione alla scrittura di metodi

2.1 Definizione di metodo

Un metodo è un **sottoprogramma**. Può appartenere a un tipo o a un oggetto di un dato tipo (vedremo a breve la differenza formale), ma non sarà mai "libero": **avrà sempre un contenitore**.

Definiamo (creiamo) un metodo quando identifichiamo una operazione ripetitiva che vogliamo compiere più volte senza doverla riscrivere ogni volta. In questo caso creiamo il metodo e lo *richiamiamo* a seconda delle necessità. Abbiamo avuto un esempio sotto gli occhi per tutto il tempo:

```
System.out.println("Benvenuti al nostro programma");
System.out.println("Inserire un comando");
//println è un sottoprogramma dell'oggetto out. Il suo compito è stampare.
//le scritte "Benvenuti al nostro programma" e "Inserire un comando" sono i parametri o argomenti
//del metodo println (un solo valore in questo caso).
```

Una prima suddivisione dei metodi è fra:

- i metodi void, fra cui il già noto main(), che eseguono un compito senza produrre un valore di risposta (senza un *ritorno*)
- i metodi non void, che devono produrre (restituire) un valore

Un esempio di metodo void è il già citato println. Stampa, ma non fornisce alcun valore.

Un esempio di metodo non void è nextLine() dell'oggetto di classe Scanner, che abbiamo usato per i primi esercizi. nextLine() restituisce un oggetto di tipo String, letto da una qualche sorgente (spesso la tastiera o un file).

Nello studiare i metodi dovremo sempre chiederci:

- cosa entra? di quali parametri ha bisogno il metodo per funzionare?
- cosa esce? il metodo produce qualcosa? varrà sempre la regola di concordanza dei tipi

Sempre per restare in territorio noto, il metodo println richiede una String da stampare, ma non restituisce niente. Il metodo nextLine() non richiede alcun tipo di parametro, ma produce una stringa, che poi tipicamente andrà a utilizzare:

```
Scanner keyboard = new Scanner(System.in);
System.out.println("Benvenuti al nostro programma");
System.out.println("Inserire un comando");
String cmd = keyboard.nextLine();
```

2.2 Analisi di un metodo di esempio

Una seconda classificazione dei metodi li divide in static (di classe, di tipo) e non static (di oggetto), ma la approfondiremo nel capitolo successivo. Per ora, presentiamo a titolo di esempio un metodo di nome sum (somma) contenuto in una classe di nome "OurMath":

```
//OurMath.java
public class OurMath
{
    public static int sum(int a, int b)
    {
        return a+b;
    }
}
//Main.java
public static void main(String[] args)
{
    System.out.println(OurMath.sum(4,5));
}
```

Analizziamo i punti chiave:

- il metodo sum è static, e appartiene alla classe OurMath
- deve restituire un numero intero. Il tipo di ritorno è l'ultima parola subito prima del nome del metodo
- dovrà ricevere da fuori due parametri, entrambi di tipo intero, che lui chiamerà a e b
- return si legge "restituisci", e termina sempre e comunque l'esecuzione del metodo. Scrivendo return, io ripasso il controllo al *chiamante*, dandogli la risposta che mi aveva chiesto. In questo caso, fornisco la somma di a e b.

Quando in main() scriviamo OurMath.sum(4,5), i valori 4 e 5 vengono copiati in a e b all'interno del metodo sum() della classe OurMath. Il metodo viene quindi eseguito, e produce un risultato. A questo punto il risultato viene sostituito al metodo nel main(). In poche parole, scrivere OurMath.sum(4,5) e 9 sarà la stessa cosa, ma il programma per saperlo dovrà eseguire il metodo, vale a dire eseguire il calcolo.

2.3 Uso del return

La parola chiave `return` è obbligatoria nei metodi con ritorno e opzionale nei metodi `void`. Un metodo può presentare diversi `return` nel suo codice, ma solo uno di questi verrà eseguito, visto che `return` provoca la terminazione del metodo. Per questa ragione, nei metodi `void` `return` viene usato solo per terminare anzitempo un metodo.

Vediamo un esempio. Scriviamo un metodo `void` che chieda base e altezza di un rettangolo e ne stampi l'area:

```
//siamo in Main.java
static void askRectangle()
{
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Insert side 1 of the rectangle");
    int side1 = keyboard.nextInt();
    if(side1 <0)
    {
        System.out.println("Wrong number. Expected positive value");
        return;
    }
    int side2 = keyboard.nextInt();
    if(side2 <0)
    {
        System.out.println("Wrong number. Expected positive value");
        return;
    }
    System.out.println(side1 * side2);
    //non mi serve dare return qui: il metodo è già finito, siamo all'ultima istruzione.
}
```

In questo caso ho scelto di terminare il metodo prima, appena ricevuto un dato insensato. `return` è la maniera "elegante" di farlo. La terminazione anticipata esiste anche per i metodi non `void`. Supponiamo di voler calcolare il prezzo del biglietto di un museo. Il prezzo pieno è di 10 euro, ma gli over 70 pagano 5 euro a prescindere, mentre chi dona il sangue (e decisamente ha meno di settant'anni) ha diritto a uno sconto di un euro. Potremmo scrivere un metodo simile in questo modo:

```
// sempre in Main.java, evito di creare altre classi per ora
// faccio arrivare al metodo l'età del cliente e una string contenente Y o N
public static int ticket(int age, String donor)
{
    if(age> 70)
        return 5;
    //se sono qui, la condizione sopra era falsa. return avrebbe terminato il metodo
    return donor.equals("Y") ? 9 : 10;
}
```

Il metodo potrebbe terminare alla propria seconda riga, a seconda dei valori di input. Avrei anche potuto scriverlo in questo modo:

```
public static int ticket(int age, String donor)
{
    int res = 10;
    if(age> 70)
        res = 5;
    else
        if(donor.equals("Y"))
            res=9;
    return res;
}
```

Questa soluzione è forse più elegante, perché ho un unico `return` in fondo al metodo, ed è evidente che a essere restituita (calcolata) è la variabile `res`. Avere molti `return` rischia di rendere il codice meno leggibile e il debug più difficile. Di contro, la prima soluzione è più efficiente, richiedendo meno calcoli e meno controlli.

Ci si aspetta che sappiate usare entrambe le soluzioni, a seconda dei casi, visto che entrambi gli stili sono usati nella pratica e preferiti in un team piuttosto che in un altro.

3 - Classi e Oggetti

3.1 Creare un tipo

Supponiamo di voler gestire un archivio di persone.

Notiamo che in Java non esiste un tipo primitivo per memorizzare i dati di una persona (i tipi primitivi, lo ricordiamo, sono **byte, short, int, long, float, double, boolean, char**).

C'è una buona ragione per questo: programmi diversi richiedono di definire lo stesso oggetto fisico (una persona) in maniera diversa. Un programma potrebbe avere bisogno di registrare gli orientamenti politici o religiosi della persona trattata (col suo consenso) o averne il divieto assoluto. Per il programma di una palestra potremmo avere bisogno di registrare peso e statura della persona (per il calcolo della fitness), mentre non ci interesserà assolutamente registrando i clienti di un negozio online.

In effetti, *non esiste un solo modo di definire un oggetto* . Nel momento in cui parliamo di un oggetto, fisico o meno (anche un conto in banca è un oggetto!), siamo costretti a decidere cosa ci interessa sapere e a scartare il resto. Questo è detto anche problema di modellizzazione, cioè di creazione di un modello su cui lavorare, che si adatti ai nostri scopi.

Java, quindi, non offre un tipo Person, ma ci permette di crearlo seguendo le nostre necessità. Come prima cosa, dovremo definire un *modello* di Person. Questo ci porta al concetto di classe-modello o classe-tipo. In Java, potrebbe essere qualcosa di questo tipo:

```
// definito in Person.java. Il file deve chiamarsi come la classe
class Person
{
    String name,surname,dateofbirth;
}
```

Notate che la classe modello non ha, tipicamente, un main. E' pensata per fornire a Java un "concetto" - in questo caso il concetto di Person.

Ho optato per un censimento estremamente basilare della persona. Sto dicendo che una variabile di tipo Person conterrà a sua volta tre variabili, date, surname e dateofbirth, tutte di tipo String.

E' importante notare che non sto parlando di una persona specifica: sto enunciando qualcosa che varrà per tutte le Person che andrò a creare. Sto dicendo "ogni persona avrà nome, cognome e data di nascita", ma non è detto che abbiamo lo stesso valore.

3.2 Uso pratico della classe: dichiarare e creare oggetti

Ora ho creato la classe, vale a dire il modello, lo "stampo" per realizzare una persona. Come lo uso?

```
Person ferdinando = new Person();
ferdinando.name = "Ferdinando";
ferdinando.surname = "Primerano";
ferdinando.dateofbirth = "05/02/1980";
```

Chiariamo gli elementi: Person è la classe (il modello), ferdinando è l'oggetto. Person è la categoria, ferdinando è l'individuo. Person è il tipo, ferdinando è una variabile che contiene un oggetto di quel tipo.

Ma perché new?

Perchè non semplicemente Person ferdinando = Person();? E perchè le parentesi tonde? Vediamo cosa succede sotto il cofano. new è un operatore, ed è l'unico modo in Java di creare oggetti. Facendo new io devo dire al compilatore **cosa** sto creando, vale a dire, il suo *tipo*, la sua *classe*.

In questo caso sto dicendo a Java di creare un oggetto di tipo Person.

In termini tecnici direi che ho **istanziato la classe Person**. L'oggetto ferdinando è **una istanza della classe Person**. Istanziare una classe è sinonimo di "creare un oggetto di quella classe".

Nella pratica, il compilatore *alloca* la memoria necessaria, prepara in memoria una "scatola" adatta a contenere Person, e di conseguenza anche name, surname e dateofbirth. Potete immaginare una scatola di nome ferdinando, con dentro tre scatole di nome name, surname e dateofbirth, rispettivamente. Questa operazione viene eseguita quando scrivo new. Notiamo quanto segue:

```
Person ferdinando;
```

NON crea nessun oggetto. Io qui ho *dichiarato la variabile*, ma la variabile non contiene ancora niente. Potete immaginarla come una scatola vuota, che non contiene nulla. Un giorno conterrà una Person, probabilmente, ma per ora è *null*.

null è un termine particolare, ed equivale all'assenza di valore. Non è l'oggetto ferdinando vuoto, non è zero, non è la stringa vuota "". Quelli sono tutti valori non nulli. null significa "assenza". **Il valore predefinito di tutti gli oggetti è null**.

```
String a;
Person p;
```

a un giorno conterrà una stringa, p un giorno conterrà una persona, ma per ora sono entrambe null. Tecnicamente, "non puntano a niente". Non equivalgono a nessuna area di memoria.

3.3 Costruttori, impliciti ed esplicativi

Resta un'ultima domanda: perché new Person()? Cosa sono le parentesi?

In realtà, creando un oggetto stiamo sempre, **sempre**, usando un metodo particolare, detto costruttore, che si trova dentro la relativa *classe*. Il costruttore ha come caratteristiche quelle di avere lo stesso nome della classe e di non avere un tipo di ritorno (come invece accade agli altri metodi).

Lo stiamo usando anche quando non lo vediamo, quando non lo scriviamo. In effetti, la classe scritta sopra in realtà ha questa forma:

```
class Person
{
    String name,surname,dateofbirth;
    public Person()
    {
        //non faccio niente qui
    }
}
```

Le righe dalla 5 alla 8 non le avete scritte, ma è come se ci fossero. Viene detto "costruttore implicito" o "di default". Si occupa di costruire l'oggetto. In questo caso, non fa assolutamente niente, ma esiste comunque e Java lo fornisce in assenza di altri costruttori. *Se non ci sono costruttori, c'è il costruttore vuoto*. Java ha sempre bisogno di sapere come costruire un oggetto. Un qualche costruttore viene sempre *invocato*.

Ed è qui che le cose si fanno complesse. Se noi forniamo un costruttore esplicito, il costruttore di default viene disabilitato, e per essere usato deve essere riscritto. Un esempio chiarirà tutto:

```
class Person
{
    String name,surname,dateofbirth;
    public Person(String n, String s, String d)
    {
        name = n;
        surname = s;
        dateofbirth = d;
    }
}
```

La classe Person ora dispone di un costruttore esplicito. Il costruttore implicito è stato disabilitato. Il seguente codice produrrà un errore:

```
Person p1 = new Person();
// ERRORE: non c'è più quel costruttore
Person p2 = new Person("Ferdinando", "Primerano", "05/02/1980");
//funziona
```

Ma ora andiamo ad analizzare in dettaglio cosa fa quel costruttore. Tecnicamente parlando, il costruttore è un metodo che si occupa di costruire l'oggetto, ma a partire da cosa? In questo caso, da tre *parametri*.

Un parametro, lo ricordiamo, è tutto quello che "arriva da fuori". In questo caso, al metodo Person arrivano tre parametri {n,s,d}, tutti di tipo String. Sono gli *input* del metodo.

Il costruttore a questo punto copia il valore di n in name, il valore di s in surname e il valore di d in dateofbirth. Equivale a quello che abbiamo fatto prima manualmente. Ora servirebbe rispondere alla domanda: a chi appartengono name, surname e dateofbirth? In questo caso all'oggetto p2. In generale, quelle sono le *proprietà* dell'oggetto che sto creando.

I parametri sono "usa e getta". Esistono solo nel costruttore e vengono gettati appena il costruttore termina. Le proprietà invece sono "locali all'oggetto", o "di proprietà dell'oggetto", ed esistono per tutto il tempo per cui esiste l'oggetto. L'oggetto p2 conterrà i suoi valori ("Ferdinando", "Primerano", "05/02/1980") fin tanto che sarà in uso, e li perderà solo quando verrà distrutto. Nessuno tuttavia mi vieta di cambiarli.

3.4 Stato dell'oggetto, unione di dati e codice

Introduciamo un concetto teorico importante: lo *stato* di un oggetto.

Lo stato di un oggetto è *il valore delle sue proprietà in un dato momento*. Lo stato dell'oggetto p2, che è una Person, è composto da name="Ferdinando", surname="Primerano" e dateofbirth="05/02/1980". Questo potrà variare nel tempo, se ad esempio scelgo di cambiare nome:

```
p2.name = "Federico";
//Notiamo che p2 è sempre una Person. Ha sempre e comunque una proprietà name, ma il suo valore può cambiare .
```

Un altro concetto teorico importante è quello dell'oggetto come mondo a sé. L'oggetto creato (p2) conosce solo ciò che lo riguarda (in questo caso name, surname e dateofbirth). Non conosce il mondo circostante, se non tramite i parametri che riceve. Vedremo un esempio concreto a breve, parlando di metodi.

Per ora, Person è stato solo un modo elegante per contenere tre variabili. Già così ne varrebbe la pena, ma vogliamo fare di più. **Vogliamo riunire in un posto solo (l'oggetto) sia lo stato (i suoi valori) che i sottoprogrammi atti a manipolarli (e di nostro interesse), vale a dire dei metodi.**

L'idea è che un oggetto sia in grado sia di contenere dei valori (il suo stato) sia di eseguire dei *calcoli su se stesso*. Possiamo immaginare l'oggetto come una entità in grado di riflettere su se stessa, e di produrre dei risultati o di svolgere dei compiti.

Vediamo un esempio concreto: l'oggetto p2 conosce la propria data di nascita (dateofbirth) ma non conosce la propria età. E' in grado di calcolarla?

La risposta è sì: conoscendo la mia data di nascita e l'anno corrente (esiste il modo di ricavare agevolmente), sono in grado di calcolare quanti anni ho. p2 è in grado di calcolare i propri anni, vediamo come:

```
class Person
{
    String name,surname,dateofbirth;
    public Person(String n, String s, String d)
    {
        name = n;
        surname = s;
        dateofbirth = d;
    }
    public int getAge()
    {
        // anno corrente
        int year = Calendar.getInstance().get(Calendar.YEAR);
        // anno di nascita
        int yearofbirth = Integer.parseInt(dateofbirth.split("/")[2]);
        // ipotizzo che dateofbirth sia nel formato dd/mm/yyyy
        // la splitto per / e prendo la terza parte (l'elemento di posizione 2)
        // essendo l'elemento di posizione 2 una stringa, devo parsarla a intero
        // ipotizzo che abbia già compiuto gli anni, per evitare di ricavare
        // mese e giorno
        // "circa" questa età, con uno scarto di un anno
        return year - yearofbirth;
    }
}
```

Lasciando allo studente il compito di scrivere meglio il metodo getAge(), andiamo a studiarlo. Notiamo che getAge() è un metodo (sottoprogramma) che restituisce (produce) un intero. In questo caso, ricava l'anno attuale chiedendolo a un metodo di un oggetto di classe Calendar, di cui ci occuperemo in futuro, e vi sottrae l'anno di nascita. Il risultato di questa operazione viene poi restituito al *chiamante*.

3.5 Incapsulamento - concetto teorico e necessità pratica

Gli oggetti creati fino ad ora hanno permesso a determinati altri oggetti e classi di accedere alle loro proprietà e metodi. Questo può portare a problemi sgraditi:

```
Person p = new Person("F","P", "05/02/1980");
p.dateofbirth = null;
```

Abbiamo appena manomesso, da fuori, il funzionamento dell'oggetto p. La prossima chiamata a `getAge()` produrrà un errore, essendo `dateofbirth` null. L'oggetto p non si è comportato come un mondo a parte, essendo troppo "aperto" verso il mondo esterno, che in effetti lo ha usato male.

A tale scopo vogliamo restringere l'accesso che gli altri oggetti o classi hanno verso un oggetto di tipo Person. Vogliamo che sia l'oggetto stesso a permettere o a negare determinate azioni (ad esempio, la modifica di una data di nascita), di modo da mantenere *coerente* il proprio *stato*, e quindi poter funzionare correttamente.

Se siamo un oggetto, dobbiamo garantirci due cose:

- che il mondo esterno non conosca il nostro funzionamento interno (il *come* lavoriamo), se non per le chiamate a metodo. L'oggetto Person rende noto il *cosa* fa, non il *come*. Questo viene detto *applicazione dell'information hiding*.
- che il mondo esterno non possa cambiare il nostro funzionamento intervenendo sulle nostre proprietà. E' l'oggetto a essere responsabile del proprio stato.

Il principio dietro questo processo di chiusura viene detto *incapsulamento*, e serve a garantire quell'approccio da "mondo indipendente e autonomo" di cui avevamo parlato in precedenza, e che prima era appunto solo teorico. Ma come arrivarcì?

3.6 Incapsulamento in pratica: i livelli di accesso

Proprietà e metodi di una classe dispongono di un livello di accesso, scelto fra quattro, che indica chi potrà accedervi. Di seguito riportiamo un esempio esplicativo:

```
// versione 1 - package
// se non c'è niente prima del tipo della variabile (String) voi leggete "package"
// le proprietà name,surname e dateofbirth sono accessibili dall'oggetto stesso e dalle classi e oggetti
// interne allo stesso package della classe Person
class Person
{
    String name,surname,dateofbirth;
}
// versione 2 - public
// va dichiarato esplicitamente. Se scrivo public le proprietà sono accessibili a chiunque, e chiunque
// le può modificare. E' l'opposto del concetto di encapsulamento, e non si usa quasi mai per le proprietà
class Person
{
    public String name,surname,dateofbirth;
}
// versione 3 - private
// va dichiarato esplicitamente. E' l'opposto di public. Queste proprietà sono visibili solo
// all'interno dello stesso oggetto.
// le proprietà sono tipicamente private. Vedremo dopo perchè
class Person
{
    private String name,surname,dateofbirth;
}
//main
public static void main(String[] args)
{
    Person p = new Person();
    // ERRORE: field name is not visible
    p.name = "Ferdinando";
}
// versione 4 - protected. Va dichiarato esplicitamente. protected equivale a package
// ma in aggiunta alle classi del package, anche le classi figlie possono accedere alla proprietà o al metodo
// di classi figlie parleremo quando introdurremo l'ereditarietà
// per ora non avremo ragione di usare questa parola chiave
class Person
{
    protected String name,surname,dateofbirth;
}
```

3.7 Incapsulare Person - getter e setter

Vogliamo garantire che Person funzioni bene, in maniera autonoma e senza intromissioni esterne. Potremo provare con una soluzione private:

```
class Person
{
    private String name,surname,dateofbirth;
    public Person(String n, String s, String d){... }
    public int getAge(){...}
}
//main
public static void main(String[] args)
{
    Person p = new Person("F", "P", "05/02/1980");
    // questo non verrà permesso,
    // p.dateofbirth = "XXXX";
    System.out.println(p.getAge());
    // ma rischia di essere troppo restrittiva. E se volessi rileggere la data di nascita?
    // errore: neanche questo verrà permesso. dateofbirth non è visibile neanche in lettura
    // System.out.println(p.dateofbirth);
}
```

Come abbiamo visto private è molto stringente. L'ideale sarebbe avere accesso totale in lettura ma limitato in scrittura. Questo non è rappresentato da un livello di accesso, ma viene ottenuto tramite metodi specifici noti come getter e setter. Un *getter* è un metodo che si occupa di fornire accesso in lettura a una proprietà che normalmente sarebbe non visibile. Comincia, per convenzione, con get, ed è piuttosto semplice da scrivere. Vediamo come risolvere il problema di sopra usando i getter.

```
class Person
{
    private String name,surname,dateofbirth;
    public Person(String n, String s, String d){... }
    public int getAge(){...}
    //un getter, un metodo che si occupa di leggere una proprietà nascosta
    public String getName()
    {
        return name;
    }
    public String getSurname()
    {
        return surname;
    }
    public String getDateofbirth()
    {
        return dateofbirth;
    }
}
//main
public static void main(String[] args)
{
    Person p = new Person("F", "P", "05/02/1980");
    System.out.println("Mr "+p.getName()+" born on "+p.getDateofbirth()+" aged:"+p.getAge());
}
```

Notiamo che getName(), getSurname() e getDateofbirth() hanno visibilità **public**. Tutti potranno chiedere all'oggetto Person quelle informazioni (e quindi resteranno disponibili) ma ad ora nessuno potrà modificarle dopo la creazione dell'oggetto. In effetti, una volta impostate quelle proprietà, non potranno più essere cambiate da fuori, almeno a livello teorico.

Con questo tipo di encapsulamento siamo sicuri che una Person creata in maniera consistente non possa essere resa inconsistente in seguito, ma ci siamo anche impediti di effettuare modifiche. Rischia di essere ancora troppo vincolante. A tale scopo offriremo un accesso in scrittura limitato tramite dei metodi convezionali che prendono il nome di *setter*. I setter iniziando con la parola set, e permettono agli altri oggetti di cercare di cambiare i valori delle proprietà, anche di quelle private, ma con una differenza fondamentale: è l'oggetto stesso (Person) a occuparsi del cambiamento, ed eventualmente anche a rifiutarsi di eseguirlo. Vediamo un esempio:

```
class Person
{
    private String name,surname,dateofbirth;
    public Person(String n, String s, String d){... }
    public int getAge(){...}
    //un getter, un metodo che si occupa di leggere una proprietà nascosta
    public String getName()
    {
        return name;
    }
```

```

    }
    public String getSurname()
    {
        return surname;
    }
    public String getDateofbirth()
    {
        return dateofbirth;
    }
    //setter: metodi che permettono al mondo esterno di modificare delle proprietà non visibili
    //potremmo averli o non averli. Non avendoli, una proprietà privata risulta essere di "sola lettura"
    public void setName(String name)
    {
        // qui sto provando a cambiare il nome
        // ma voglio assicurarmi che sia un nome sensato, o almeno, non nullo
        if(name!=null)
            this.name = name;
        //dovrebbe ricordarvi qualcosa. E' il procedimento dei costruttori, ma lavoro su un campo solo
        //essendo un metodo void non ho un return, o almeno non è necessario
    }
    public void setSurname(String surname)
    {
        // qui sto provando a cambiare il nome
        // ma voglio assicurarmi che sia un nome sensato, o almeno, non nullo
        if(surname!=null)
            this.surname = surname;
        //dovrebbe ricordarvi qualcosa. E' il procedimento dei costruttori, ma lavoro su un campo solo
        //essendo un metodo void non ho un return, o almeno non è necessario
    }
    public void setDateofbirth(String date)
    {
        if(date!=null && date.split("/").length==3)
            this.date = date;
    }
}
//main
public static void main(String[] args)
{
    Person p = new Person("Fernando", "Primeran", "05/02/1980");
    // p.name = "Ferdinando", non funziona, name è privata
    p.setName("Ferdinando");
    // funziona. setName è public, ed è l'oggetto p che cambia la proprietà name per conto di main()
    p.setSurname("Primerano");
    //idem
    p.setDateofbirth(null);
    //il metodo viene eseguito ma dateofbirth non viene cambiato. Cambiarlo avrebbe portato l'oggetto
    //a essere in uno stato inconsistente
}

```

In questo modo abbiamo conservato la *visibilità* che ci permette di utilizzare le informazioni interne all'oggetto e allo stesso tempo garantito la *consistenza dello stato*. E' la soluzione comune che troverete nei team di sviluppo: proprietà private, getter e setter ove necessario. Torneremo su setter e getter in seguito, perché dovremo rivedere in particolare il metodo setDateofbirth, e in generale tutti quei setter, dato che si rifiutano di svolgere il loro lavoro (modificare il dato) senza renderne conto al *chiamante*, che invece avrebbe diritto di esserne informato. Questo significherà offrire un ritorno, o almeno un'eccezione, anche nei setter.

3.8 Il rapporto di uso fra oggetti o classi e oggetti e primo riepilogo

Prima di proseguire, torniamo al concetto di "chiamante"? E' un altro concetto chiave. Gli oggetti (ma anche le classi di avvio, le classi con main, quelle che possiamo eseguire direttamente) hanno fra di loro un rapporto di **uso**. Vediamo un esempio concreto:

```
public static void main(String[] args)
{
    Person p = new Person("F", "P", "05/02/1980");
    System.out.println(p.getAge());
    // 40, nel 2020
}
```

Noi diremo che il metodo main() *chiama* il metodo getAge() dell'oggetto p di classe Person, o che lo **usa**. In effetti, la classe main() sta usando l'oggetto p. In questo caso main è il chiamante (il cliente) e p è il chiamato (il fornitore di un servizio, in questo caso, il calcolo della propria età). Abbiamo visto questo schema dal primo giorno :

```
String name = keyboard.nextLine();
// in questo caso main() è il chiamante del metodo nextLine()
// dell'oggetto keyboard, di classe Scanner. nextLine() produrrà
// una stringa, leggendola da tastiera, e il main la salverà nella
// variabile name.
```

Tornando ai metodi, precisiamo che sono diversi dalle proprietà. Le proprietà sono *dati memorizzati nell'oggetto*, e il loro valore costituisce lo stato dell'oggetto. Sono il suo contenuto.

I metodi sono *dati calcolati*, o comunque *servizi offerti dall'oggetto al mondo*, in aggiunta alle proprietà. In questo caso l'oggetto di classe Person offre come servizio il calcolo della propria età.

Cominciamo quindi a dividere l'oggetto creato dalla classe Person in tre parti:

- I suoi costruttori, che sono i modi in cui l'oggetto può essere creato, e sono dei metodi speciali
- Le sue proprietà, che sono dati memorizzati, e che tipicamente vengono ricevuti dall'esterno tramite i costruttori. Il loro valore nel tempo costituisce lo stato dell'oggetto.
- I suoi metodi, che sono sottoprogrammi e producono tipicamente dati calcolati. Un metodo è un servizio che viene offerto, tipicamente, al resto del mondo o a qualche altro metodo. Si dice in gergo che i metodi sono il *comportamento* dell'oggetto.

L'oggetto quindi ragiona su se stesso e fornisce uno stato e un comportamento. I dati sono stati legati ai sottoprogrammi che possono manipolarli, formando una unità funzionale che verrà usata dal resto del programma.

L'oggetto cerca anche di funzionare come scatola nera: offre un servizio senza spiegare come lo offre. Il resto del mondo non deve interessarsi di come l'oggetto calcola l'età, così come non ci chiediamo come faccia Scanner a leggere da tastiera. Sappiamo solo cosa ci restituisce nextLine() (una String), e come invocare il metodo, per ottenere il servizio.

3.9 Comunicare con l'oggetto: metodi e passaggio di parametri

In alcuni casi l'oggetto non basta a se stesso. L'oggetto p dell'ultimo esempio è in grado di calcolare la propria età, ma non può dire se è adulto o meno. L'età per essere considerati adulti può variare dai 16 ai 21 anni, con 18 come numero più comune, ma l'oggetto non ha modo di saperlo. Creando l'oggetto p, non gli abbiamo fornito informazioni su dove si trova.

Inoltre, se p dovesse spostarsi in uno stato diverso, potrebbe non essere più considerato adulto (e non poter bere, ad esempio), quindi è necessario che alla domanda "sei un adulto?" segua un'altra domanda: "quale è il limite di età per essere considerato tale?". In altre parole, ci serve un parametro.

```
class Person
{
    String name,surname,dateofbirth;
    public Person(String n, String s, String d)
    {
        name = n;
        surname = s;
        dateofbirth = d;
    }
    public int getAge()
    {
        // anno corrente
        int year = Calendar.getInstance().get(Calendar.YEAR);
        // anno di nascita
        int yearofbirth = Integer.parseInt(dateofbirth.split("/")[2]);
        // ipotizzo che dateofbirth sia nel formato dd/mm/yyyy
        // la splitto per / e prendo la terza parte (l'elemento di posizione 2)
        // essendo l'elemento di posizione 2 una stringa, devo parsarla a intero
        // ipotizzo che abbia già compiuto gli anni, per evitare di ricavare
        // mese e giorno
        // "circa" questa età, con uno scarto di un anno
        return year - yearofbirth;
    }
    //threshold = soglia
    public boolean isAdult(int threshold)
    {
        return getAge()>=threshold;
    }
}
```

Un metodo semplice, ma che merita un minimo di analisi.

Il metodo si chiama isAdult. Restituisce (produce) un valore booleano, ma richiede un numero intero per produrlo (threshold) che deve arrivare da fuori. L'oggetto non dispone dell'informazione relativa al paese che gli sta chiedendo se è adulto o meno.

Notiamo anche che il metodo isAdult() utilizza il metodo getAge(). Un metodo di un oggetto può sempre richiamarne un altro dello stesso oggetto. isAdult() dipende internamente da getAge(), esternamente da threshold. getAge() è qualcosa che riesce a calcolare da solo, mentre per threshold è dipendente dal chiamante.

```
Person p1 = new Person("F", "F", "05/02/1980");
Person p2 = new Person("A", "B", "05/02/2003");
System.out.println(p1.adult(18));
System.out.println(p1.adult(21));
// true, true
System.out.println(p2.adult(18));
System.out.println(p2.adult(21));
// true, false
```

threshold ha preso i valori 18, 21, 18 e 21 rispettivamente. Il sottoprogramma isAdult è stato eseguito prima su p1 (due volte) e poi su p2 (due volte).

3.10 Il concetto di this

Torniamo brevemente sui costruttori, e impariamo a conoscere un nuovo termine, *this*. I costruttori hanno spesso questa forma:

```
public Person(String name, String surname, String dateofbirth)
{
    this.name = name;
    this.surname = surname;
    this.dateofbirth = dateofbirth;
}
```

In questo caso i parametri (vengono da fuori, locali al metodo, verranno usati e buttati via) hanno lo stesso nome delle proprietà (restano dentro l'oggetto e costituiscono lo stato). Il costruttore Person deve poter distinguere fra le due cose. La riga `this.name = name;` si legge come segue: "prendi la variabile temporanea `name` e copia il suo valore all'interno della mia proprietà `name`, che invece esisterà anche fuori dal costruttore". `this.name` è la proprietà, `name` è il parametro.

Questo è necessario solo in caso di omonimia. Negli altri metodi, o in generale quando non abbiamo parametri con lo stesso nome delle proprietà, non c'è differenza fra scrivere `this.name` e `name`. Il `this` in questo caso è implicito.

Resta aperta una questione: cosa è `this`?

`this` è come dire "me stesso", "io". Quando scrivo `this` dentro Person, io mi sto riferendo all'oggetto stesso. Vediamolo con un esempio: supponiamo di dover determinare la persona più vecchia fra due. Un modo di farlo è il seguente:

```
// siamo sempre dentro Person.java, dentro la classe Person, che quindi
// definisce l'oggetto Person
public Person older(Person other)
{
    return getAge() > other.getAge() ? this : other;
}
```

Questo metodo restituisce un oggetto di tipo Person. Un metodo può produrre, letteralmente, qualunque tipo, sia esso primitivo (`int, double...`) o oggetto. Interpretiamolo: calcolo la mia età (`getAge()`, che implicitamente è `this.getAge()`) e la confronto con `other.getAge()` (chiedo all'altro quanto è vecchio). Se la mia età è superiore alla sua, restituisco, non la mia età, ma me stesso per intero (un oggetto di tipo Person). Altrimenti, restituisco lui (di nuovo un oggetto Person, e non potrebbe essere diversamente: il metodo deve produrre Person).

```
Person p1 = new Person("F", "P", "05/02/1980");
Person p2 = new Person("A", "B", "05/01/1990");

System.out.println(p2.older(p1).name);
//stamperà "F"
//viene eseguito il metodo older di Person.java
//con this = p2 e other = p1
//il ternario è falso, e viene restituito other, vale a dire, p1
//quindi, p2.older(p1) = p1
//del risultato (p1) stampo il nome ("F")
```

Come avete visto, in alcuni casi dovremo restituire o comunque usare "this" per indicare l'oggetto in cui ci troviamo. Nel caso sopra `this` era `p2` perché ho scritto `p2.older(p1)`. Se avessi scritto `p1.older(p2)` avremmo avuto `this = p1` e `other = p2`. Il risultato non sarebbe cambiato comunque, in questo caso.

3.11 Polimorfismo dei costruttori

Sempre restando sui costruttori, notiamo che possiamo avere diversi costruttori fin tanto che abbiamo parametri diversi. Questo viene detto *polimorfismo dei costruttori* (polimorfismo - molte forme), e si traduce nella possibilità di avere diversi costruttori, purché abbiano fra loro diversi parametri. Significa che posso avere un costruttore, come quello appena scritto, che usa tre stringhe, e un altro che ne usa una sola:

```
class Person
{
    String name,surname,dateofbirth;
    public Person(String name, String surname, String dateofbirth)
    {
        this.name = name;
        this.surname = surname;
        this.dateofbirth = dateofbirth;
    }
    //mi aspetto che mi arrivi una stringa del tipo nome,cognome,data
    public Person(String csv)
    {
        String parts[] = csv.split(",");
        name = parts[0];
        surname = parts[1];
        dateofbirth = parts[2];
    }
    public int getAge()
    {
        // anno corrente
        int year = Calendar.getInstance().get(Calendar.YEAR);
        // anno di nascita
        int yearofbirth = Integer.parseInt(dateofbirth.split("/")[2]);
        // ipotizzo che dateofbirth sia nel formato dd/mm/yyyy
        // la splitto per / e prendo la terza parte (l'elemento di posizione 2)
        // essendo l'elemento di posizione 2 una stringa, devo parsarla a intero
        // ipotizzo che abbia già compiuto gli anni, per evitare di ricavare
        // mese e giorno
        // "circa" questa età, con uno scarto di un anno
        return year - yearofbirth;
    }
    //threshold = soglia
    public boolean isAdult(int threshold)
    {
        return getAge()>=threshold;
    }
}
```

La classe sopra può creare una Person a partire sia da tre stringhe (tre valori separati) sia da una stringa sola (ad esempio: "F,P,05/02/1980").

```
Person p1 = new Person("F","P", "05/02/1980");
Person p2 = new Person("F,P,05/02/1980");
//si arriva allo stesso risultato ma sto usando due costruttori diversi
```

3.12 La classe come entità autonoma dall'oggetto - variabili di classe

Fino ad ora, abbiamo parlato di classe come modello per l'oggetto, ma la classe, anche una classe tipo, può definire proprietà e metodi propri, che vengono appunto detti "di classe". Si indicano con la parola chiave `static`. Da un punto di vista teorico, ciò che definisco come static **non riguarda il singolo oggetto ma l'intera categoria**. Riguarda il **tipo stesso**, non l'oggetto del tipo.

Cerchiamo di chiarire: tutti gli esseri umani hanno un'età (in questo caso, un dato calcolato tramite il metodo `getAge()`, non un dato memorizzato). Ogni essere umano ha la propria, per quanto alcuni abbiano la stessa età. In generale, l'età è una proprietà dell'oggetto `Person`. C'è tuttavia una caratteristica che ci accomuna tutti, ed è un limite alla longevità, che è di circa 120 anni. Realisticamente parlando, non abbiamo notizie confermate di esseri umani che abbiano superato i 120 anni.

Questa proprietà, e questo valore, non riguarda il singolo oggetto, ma tutti gli oggetti della categoria. Riguarda, in breve, **la classe Person**, non il singolo oggetto `Person`. Possiamo vederne l'utilizzo scrivendo un metodo `isValid()` per l'oggetto `Person`. `isValid()` o `valid()` è un metodo boolean comune che mettiamo negli oggetti che rappresentano concetti reali, più o meno fisici, e restituisce `true` se l'oggetto è in uno stato sensato (ad esempio, un essere umano la cui data di nascita sia accettabile, e che abbia nome e cognome), `false` altrimenti.

Un modo di scriverlo, per la classe `Person`, sarebbe il seguente:

```
//in Person.java
public boolean isValid()
{
    return name!=null && surname!=null && getAge() > 0 && getAge() < 120;
}
```

120 è un limite che riguarda non il singolo oggetto, ma l'intera categoria. Per esprimere meglio, possiamo fare come segue:

```
class Person
{
    //costante (FINAL) di classe (STATIC) di tipo int
    //una "variabile" final non può essere cambiata
    final static int MAXAGE = 120;
    String name,surname,dateofbirth;
    public Person(String name, String surname, String dateofbirth)
    {
        this.name = name;
        this.surname = surname;
        this.dateofbirth = dateofbirth;
    }
    //mi aspetto che mi arrivi una stringa del tipo nome,cognome,data
    public Person(String csv)
    {
        String parts[] = csv.split(",");
        name = parts[0];
        surname = parts[1];
        dateofbirth = parts[2];
    }
    public int getAge()
    {
        // anno corrente
        int year = Calendar.getInstance().get(Calendar.YEAR);
        // anno di nascita
        int yearofbirth = Integer.parseInt(dateofbirth.split("/")[2]);
        // ipotizzo che dateofbirth sia nel formato dd/mm/yyyy
        // la splitto per / e prendo la terza parte (l'elemento di posizione 2)
        // essendo l'elemento di posizione 2 una stringa, devo parsarla a intero
        // ipotizzo che abbia già compiuto gli anni, per evitare di ricavare
        // mese e giorno
        // "circa" questa età, con uno scarto di un anno
        return year - yearofbirth;
    }
    //threshold = soglia
    public boolean isAdult(int threshold)
    {
        return getAge() >= threshold;
    }
    public boolean isValid()
    {
        return name!=null && surname!=null && getAge() > 0 && getAge() < MAXAGE;
    }
}
```

Il metodo `isValid()` può fare uso delle proprietà di classe. *L'individuo conosce la categoria, l'oggetto conosce la propria classe*. In questo caso, l'oggetto chiederà alla propria classe il valore della proprietà `MAXAGE`, che oltre a essere di classe (static) è anche final (fissa). Non può essere

cambiata in seguito, una volta impostata.

Questo è un utilizzo classico delle variabili static, che spesso sono anche final. Imposto valori validi o limiti che valgono per tutti gli individui di un dato gruppo, per tutti gli oggetti di una data classe.

Non avrebbe avuto senso mettere MAXAGE in tutti gli oggetti. Sarebbe stato uno spreco di memoria, una ripetizione e un rischio - qualche oggetto avrebbe potuto avere una versione diversa di MAXAGE, eventualmente.

In termini di memoria, si crea un nuovo oggetto a ogni new, mentre la classe resta sempre una:

```
Person p = new Person("F,P,05/02/1980");
Person q = new Person("A,B,01/01/2001");
// in questo programma ci sono due oggetti (p e q) ma una sola classe (Person)
```

3.13 La classe come entità autonoma dall'oggetto - metodi di classe e regole di accesso

Una classe tipo può anche avere metodi statici, per quanto questo si usi di meno. Sono quei metodi che appartengono alla classe (Person) e non all'oggetto che stiamo creando (in questo caso, gli oggetti p e q). Vediamo un esempio concreto:

```
//entra un vettore di persone, trovo il più vecchio
public static Person olderInGroup(Person[] people)
{
    //il vettore di persone potrebbe essere vuoto.
    //in questo caso, restituisco null
    if(people.length==0)
        return null;
    // altrimenti imposto il primo come risultato
    Person res = people[0];
    // poi parto dal secondo (i=1) e lo confronto con tutti gli altri
    for(int i=1;i <people.length;i++)
        if(people[i].getAge()>res.getAge())
            res=people[i];
    return res;
}
// confronto res, il mio più vecchio attuale, con tutti i people[i]
// se people[i] è più vecchio, sostituisco res e continuo il confronto
// notiamo che people è un vettore di oggetti. Ogni singolo elemento people[i]
// è un oggetto di tipo Person.
```

Esiste una regola, antica e sacra, e che va imparata a memoria. *L'individuo conosce la categoria, la categoria non conosce l'individuo.* Questo significa che nei metodi di oggetto posso usare variabili e metodi di classe (esempio: metodo isAdult()). Nei metodi di classe, **non posso**, invece, usare metodi o proprietà di oggetto, perché essendo di classe io non riguardo il singolo oggetto, ma l'intera categoria. Non ha senso, ad esempio, scrivere getAge() o this.getAge() nel metodo sopra, perché non saprei a chi applicarlo. Nei metodi statici non esiste l'oggetto, esiste solo la classe. E in effetti io non ho mai eseguito getAge() di me stesso in quel metodo, ma *getAge() delle persone che mi sono state fornite come parametro*. La regola è che il metodo statico può lavorare solo con parametri, con proprietà statiche e con altri metodi statici. Nei metodi statici, non esiste "this", perchè stiamo lavorando su tutta la categoria.

Ma come si usa un metodo statico? Il modo corretto sarebbe il seguente:

```
// posto people = vettore di Person
Person older = Person.olderInGroup(people);
```

Viene richiamato sulla classe (lettera grande), non sull'oggetto. Il più vecchio viene calcolato partendo da people (vettore di Person, insieme di Person), non su un "this".

3.14 Scope di oggetto e scope di classe

Quanto visto prima ci porta a una rivelazione: una classe tipo definisce *due scope*, due contenitori:

- lo scope di oggetto, costituito dalle proprietà di oggetto e dai metodi di oggetto, e che ha accesso anche allo scope di classe, e in cui esiste "this"
- lo scope di classe, che ha accesso solo a metodi e proprietà di classe, e in cui non esiste this

Esplicitando:

```
class Person
{
    // scope di classe. Una proprietà final, MAXAGE
    // e un metodo che trova il più vecchio dato un vettore di Person
    // (older)
    final static int MAXAGE = 120;
    public static Person olderInGroup(Person[] people)
    {
        //il vettore di persone potrebbe essere vuoto.
        //in questo caso, restituisco null
        if(people.length==0)
            return null;
        // altrimenti imposto il primo come risultato
        Person res = people[0];
        // poi parto dal secondo (i=1) e lo confronto con tutti gli altri
        for(int i=1;i < people.length;i++)
            if(people[i].getAge()>res.getAge())
                res=people[i];
        return res;
    }
    // fine scope di classe. notiamo che olderInGroup
    // non ha accesso a name, surname, getAge() ecc
    // non confondiamo l'accesso a people[i].getAge()
    // che è l'accesso al parametro people
    // con getAge() di this, cioè dell'oggetto in cui sono
    // perché io NON SONO in nessun oggetto. Io sono la categoria
    // inizio scope di oggetto
    // proprietà di oggetto
    String name,surname,dateofbirth;
    // costruttori
    public Person(String name, String surname, String dateofbirth)
    {
        this.name = name;
        this.surname = surname;
        this.dateofbirth = dateofbirth;
    }
    //mi aspetto che mi arrivi una stringa del tipo nome,cognome,data
    public Person(String csv)
    {
        String parts[] = csv.split(",");
        name = parts[0];
        surname = parts[1];
        dateofbirth = parts[2];
    }
    // metodi
    public int getAge()
    {
        // anno corrente
        int year = Calendar.getInstance().get(Calendar.YEAR);
        // anno di nascita
        int yearofbirth = Integer.parseInt(dateofbirth.split("/")[2]);
        // ipotizzo che dateofbirth sia nel formato dd/mm/yyyy
        // la splitto per / e prendo la terza parte (l'elemento di posizione 2)
        // essendo l'elemento di posizione 2 una stringa, devo parsarla a intero
        // ipotizzo che abbia già compiuto gli anni, per evitare di ricavare
        // mese e giorno
        // "circa" questa età, con uno scarto di un anno
        return year - yearofbirth;
    }
    //threshold = soglia
    public boolean isAdult(int threshold)
    {
        return getAge()>=threshold;
    }
    // metodo di oggetto che fa uso di una proprietà (MAXAGE)
    // dello scope di classe. Non è un problema. L'oggetto conosce la classe
    public boolean isValid()
    {
```

```

        return name!=null && surname!=null && getAge()> 0 && getAge() <MAXAGE;
    }
}

```

Riassumendo: i metodi di oggetto possono accedere a metodi e proprietà di classe, metodi e proprietà di oggetto e parametro. I metodi di classe possono accedere solo a metodi e proprietà di classe e a parametri. Non hanno un oggetto predefinito "sotto" ("manca this").
Un esempio per chiarire:

```

class C
{
    // scope di classe C
    public final static int V1 = 100;
    public int static SM1(int a){...};
    public int static SM2(int b){...};
    // scope dell'oggetto di classe C
    int v2;
    public int om1(){};
    public int om2(int c){};

    //il metodo SM1 può accedere a: SM1 (può chiamare se stesso), SM2 e V1.
    //inoltre ha accesso al parametro a
    //il metodo SM2 ha accesso a SM1, SM2 e V1. Inoltre ha accesso al parametro b
    // Il metodo om1 ha accesso a TUTTO. Metodi e proprietà di classe, metodi e proprietà di oggetto
    // Idem per om2, e in aggiunta ha accesso anche a un parametro solo proprio, c.
}

```

Per riassumere un discorso complicato e anti-intuitivo:

Una classe tipo definisce quindi in primis lo stato (proprietà) e il comportamento (metodi) di un oggetto che verrà creato in seguito con l'operatore new. Definisce, in altre parole, uno scope di oggetto.

In aggiunta, definisce operazioni (metodi) e valori (proprietà, spesso final) che non riguardano il singolo oggetto, ma l'intera categoria. Definisce, in altre parole, uno scope di classe.

I due scope comunicano in maniera asimmetrica. Dall'oggetto posso sempre leggere la classe, dalla classe non posso leggere l'oggetto.

3.15 Gestire i null

A differenza dei tipi primitivi, che hanno sempre un valore, almeno di default, gli oggetti possono *sempre* essere null. Posso sempre inviare "null" come valore di un oggetto, e questo porta a una eccezione comune, `NullPointerException`, che è la dimostrazione di una scarsa attenzione da parte del programmatore.

Vediamo un esempio:

```
//supponiamo di avere ancora il costruttore implicito, o "vuoto"
Person p = new Person();
//p.name ora è null. è possibile, le stringhe SONO oggetti
System.out.println(p.name.equals("Ferdinando"));
```

Il programma non restituirà "false". Il programma restituirà una eccezione, vale a dire `NullPointerException`, e andrà in crash, perchè in effetti noi stiamo provando a chiamare un metodo (`equals`) su un oggetto (`p.name`, di tipo teorico `String`) che semplicemente non esiste. L'eccezione sarà qualcosa di questo tipo: **Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.equals(Object)" because "p.name" is null at entities.Main.main(Person.java:10)**.

Il programmatore a questo punto deve isolare la riga (nel mio caso la dieci) e verificare che tutti gli oggetti ivi presenti siano non nulli. In questo caso, nell'espressione `p.name` ci sono **due** oggetti da controllare: l'oggetto `p` di tipo `Person` e l'oggetto `name` di tipo `String` contenuto dentro `p`. Sappiamo da prima che è `name` a essere null (`p` è stato appena creato), ma un modo sporco e rapido per vedere quale dei due è null sarebbe il seguente:

```
System.out.println(p==null);
System.out.println(p.name==null);
```

Da scrivere prima dell'istruzione incriminata (prima della riga 10, nel mio caso). `NullPointerException` è un'eccezione estremamente comune ma anche semplice da risolvere. Spesso i primi tempi non si vorrà credere che uno degli oggetti sia nullo, essendo sicuri dei propri passaggi. In questo caso, come in quasi tutti gli altri, "ha ragione lui". Stampare tutte le variabili potrà aiutarvi a risparmiare diverse ore di debug. L'errore è da cercare in cima, alla catena di passaggi che ha portato alla (mancata) creazione dell'oggetto.

3.16 Overloading dei metodi

Abbiamo già visto il polimorfismo dei costruttori, vale a dire la possibilità di avere diversi costruttori, quindi diverse maniere di creare un oggetto. Lo stesso identico discorso vale per qualunque altro metodo dell'oggetto o della classe. Posso avere più metodi con lo stesso nome, a patto che abbiano parametri diversi. Questo ci torna spesso comodo.

Vediamo un esempio concreto:

```
class Time
{
    int hour, minutes, seconds;

    public Time(int hour, int minutes, int seconds)
    {
        this.hour = hour;
        this.minutes = minutes;
        this.seconds = seconds;
    }

    public Time(int hour, int minutes)
    {
        this.hour = hour;
        this.minutes = minutes;
    }

    public int toSeconds()
    {
        return hour * 60 * 60 + minutes * 60 + seconds;
    }

    public int difference(Time other)
    {
        return toSeconds() - other.toSeconds();
    }
}
```

Si tratta di una classe semplice che rappresenta un istante nel tempo nel corso di una giornata, nel formato ore-minuti-secondi. Ci sono due costruttori, uno completo e uno ridotto, nel caso in cui i secondi non dovessero interessarci (resteranno a 0, valore di default per il tipo int), e due metodi relativamente interessanti.

Il primo, int toSeconds(), converte l'orario in secondi, cominciando a contare dalla mezzanotte. Notiamo che ha bisogno solo di dati dell'oggetto, quindi non necessita di parametri. Il prefisso "to" è comune in questi casi, quando trasformiamo un oggetto in un altro tipo (in questo caso, un oggetto Time è stato "trasformato" in un intero).

Il secondo, int difference(Time other), calcola la differenza fra due oggetti di tipo Time. Necessita di un tempo come parametro, perché l'oggetto conosce solo se stesso, e ha bisogno di sapere che cosa sottrarre. In un certo senso, è come implementare l'operazione $t_1 - t_2$, con t_1 e t_2 oggetti di tipo Time. Il risultato è un intero, in questo caso dei secondi.

Ma se volessimo il risultato in minuti? O in ore? Una prima soluzione potrebbe essere creare vari metodi: differenceMinutes, differenceHours, ecc., ma possiamo usare il polimorfismo dei metodi per evitarlo. Una soluzione è la seguente:

```
//siamo sempre in Time.java
//il metodo di prima
public int difference(Time other)
{
    return toSeconds() - other.toSeconds();
}

public int difference(Time other, String unit)
{
    int res = 0;
    switch(unit)
    {
        case "h":
            res = difference(other) / (60 * 60);
            break;
        case "m":
            res = difference(other) / 60;
            break;
        case "s":
            //in questo caso il metodo si riduce a quello prima
            res = difference(other);
            break;
        default:
```

```

        res = difference(other);
    }
    return res;
}

```

Come notate abbiamo due metodi difference, che possono convivere nella stessa classe (in realtà, nello stesso oggetto), perché hanno parametri diversi. Questa è una forma di polimorfismo (pluralità di forme) dei metodi, e viene detta in gergo **overloading**. Il resto del programma potrà chiamare indifferentemente l'uno o l'altro metodo.

```

//siamo in main()
//uso il costruttore ridotto
Time t = new Time(9, 30);
//differenza in secondi. uso il primo metodo
Time t2 = new Time(10,30);
System.out.println(t.difference(t2));
//differenza in minuti. Uso il secondo
System.out.println(t.difference(t2, "m"));
//di nuovo differenza in secondi, ma sto chiamando il secondo metodo
System.out.println(t.difference(t2, "s"));

```

L'overloading è definito come: presenza di metodi con lo stesso nome ma parametri diversi nella stessa classe. E' una forma di polimorfismo, analogamente a quanto visto coi costruttori. E' anche una domanda molto popolare ai colloqui, per cui consiglio di imparare a memoria la definizione.

Nella pratica, ricordiamoci che se vogliamo due volte lo stesso metodo in una classe, dobbiamo almeno cambiare i suoi parametri.

3.17 Gli oggetti dopo la morte: il garbage collector

Gli oggetti in Java vengono creati esplicitamente ma distrutti automaticamente.

Prendiamo il seguente codice di esempio:

```

Person q;
{
    Person p = new Person();
    q = p;
}
System.out.println(q);

```

Questo programma dispone di un solo oggetto, con due riferimenti. I riferimenti sono p e q. Fin tanto che p e q rimangono in un qualche scope attivo, l'oggetto continua a esistere.

L'oggetto viene creato dentro il blocco di codice interno, col nome p, ma viene poi referenziato da q, che è esterno. L'oggetto continua quindi a esistere al di fuori del blocco di codice esterno, ed esisterà fin tanto che q rimarrà "in scope".

Concretamente, p e q sono "riferimenti" a zone di memoria. Fin tanto che il programma continua ad averne bisogno da qualche parte (in un metodo, in un oggetto, in una classe) queste zone di memoria continueranno a essere preservate.

Ma la memoria ha un costo, e Java ha una reputazione, in parte meritata, per fagocitare memoria (parte della ragione per cui i telefoni Android montano molta più RAM di alcuni concorrenti). Sarebbe il caso di liberare memoria a un certo punto, e questo in Java non è lasciato al programmatore.

Java dispone di un meccanismo noto come garbage-collector (letteralmente, "raccoglitrice di immondizia"), che mantiene un elenco dei riferimenti agli oggetti. Nel momento in cui una area di memoria (un oggetto) non ha più riferimenti (non è più collegato a una variabile) la memoria relativa viene considerata "liberabile", e potrebbe essere deallocata da un momento all'altro. Questo è il compito del garbage collector, che rimuove gli oggetti inutilizzati per fare spazio a quelli che verranno.

Il processo di rimozione è trasparente in Java e il programmatore non se ne rende conto. Utilizzando le regole di "buona educazione" nella dichiarazione delle variabili (dichiarare le variabili solo ove servono, evitare gli scope globali se possibile) il garbage collector farà il suo lavoro senza che noi ce ne rendiamo conto. Una programmazione avventata invece può generare memory leak, con programmi che allocano ma non riescono a de-allocare, con possibili rallentamenti o crash.

4 - Vettori

4.1 Definizione, parti e fasi della vita di un vettore

Definiamo **vettore** una lista ordinata di dimensione finita di elementi dello stesso tipo. Riconosciamo tre momenti potenzialmente separati nell'uso di un vettore: la **dichiarazione**, la **creazione** e il **riempimento**.

Un vettore è un tipo particolare di oggetto, e in effetti ha proprietà e metodi propri, e può essere null. Vediamo qualche esempio:

```
// ho dichiarato una variabile di nome v, di tipo "vettore di interi"  
// le quadre servono a indicare a Java che stiamo creando un vettore  
// e non un singolo intero  
int v[];  
  
v = new int[3];  
// ORA esiste il vettore. Prima esisteva solo una "scatola vuota"  
// vale a dire, null, e il compilatore mi avrebbe perfino impedito di usarla  
// ora esiste una "impalcatura", una struttura che contiene tre scatole  
// che a loro volta conterranno degli interi  
// le scatole sono numerate: 0,1,2. Si comincia a contare da 0  
v[1] = 5;  
// terza fase: riempimento. Ho messo il valore 5 alla posizione 1  
// vale a dire la seconda posizione del vettore. Ho riempito  
// la scatola centrale. Le altre due restano col loro valore di default  
// che nel caso degli interi è 0  
// il vettore ora è [0, 5, 0]
```

Possiamo accedere ai singoli elementi del vettore tramite la loro **posizione** all'interno della lista. Questo numero è anche detto, in gergo, **indice**.

```
// dichiarazione e creazione contestuale del vettore  
String[] names = new String[3];  
names[1] = "Ferdinando";  
String greeting = "Hello "+names[1];  
// names[0] e names[2] sono oggetti e sono al loro valore di default  
// vale a dire null.  
// names è [null, "Ferdinando", null]  
// creare il vettore non significa creare il suo contenuto!  
// il vettore è uno scaffale, il contenuto è costituito dagli oggetti che ci metteremo dentro
```

Vale per i vettori, come per tutto il resto, la regola di concordanza dei tipi. names = "Ferdinando" produce un errore - names NON è una String, è un vettore di String. names[1] = 5; produce un errore. L'elemento names[1] è una String, mentre 5 è un intero.

A questo si aggiunge un limite dovuto alla natura finita e fissa dei vettori. names[3]="Pino" produrrà un errore. Gli elementi del vettore hanno posizione (indice) 0, 1 e 2, perchè cominciamo a contare da 0. names[3] produrrà ArrayIndexOutOfBoundsException, letteralmente, siamo "fuori" dal vettore. Non esiste quello scaffale.

In alcuni casi possiamo creare un vettore e riempirlo allo stesso tempo. E' il caso del metodo split già visto in precedenza. split è un metodo dell'oggetto di classe String che divide la String in base a un pattern e restituisce un vettore di String.

```
String[] parts = "10/02/2020".split("/");  
//parts è ["10", "02", "2020"], con posizioni 0, 1 e 2 rispettivamente;
```

Ma dove è il new? Dove è il dimensionamento del vettore? Vengono eseguiti da split, in maniera trasparente. Ci sono, ma non si vedono. Un altro modo di riempire un vettore è la seguente forma contratta, che dichiara, crea e riempie un vettore:

```
int v[] = new int[]{6, 8, 10};
```

Il vettore è stato dichiarato, creato e riempito. Il contenuto è quello che abbiamo messo fra le graffe, quindi sarà lungo 3, e i valori sono 6,8 e 10 alle posizioni 0,1 e 2 rispettivamente.

Ricapitolando e aggiungendo gli ultimi tasselli, posto String[] names = new String[]{"A", "B"}:

- names non è una String. E' di tipo String[], vale a dire, vettore di String.
- names.length è una proprietà di tipo int dei vettori, e memorizza il numero di elementi. In questo caso, names.length = 2
- Scrivendo names prendo tutto il vettore, scrivendo names[1] prendo l'elemento alla posizione 1 (il secondo), scrivendo names[i] prendo l'elemento alla posizione i (i deve essere un numero intero). Se i > names.length-1 o i < 0, avrà un errore.

- Il numero fra le quadre viene detto indice dell'elemento nel vettore. 1 è l'indice della String "B" nel vettore names. i è l'indice, names[i] l'elemento, L'indice è sempre un intero non negativo, l'elemento in questo caso è una String.

4.2 Scorrimento di un vettore

Una categoria piuttosto ampia di problemi (detti lineari) possono essere ricondotti a un'operazione di scorrimento di un vettore, vale a dire al ciclare tutti gli elementi, partendo tipicamente dal primo e arrivando all'ultimo, di solito in sequenza.

E' estremamente intuitivo. Si parte con l'elemento 0, poi l'1, il 2... fino ad arrivare all'elemento vettore.length-1 (l'ultimo), e su ogni elemento viene eseguita una qualche operazione.

Di solito per farlo si utilizza un ciclo for, che dispone degli strumenti per mandare automaticamente avanti l'indice, facendoci così passare da una "cella" all'altra del vettore.

Nella sua forma più basilare, lo scorrimento può essere scritto così:

```
for(int i=0;i < vettore.length ; i++)
// fai qualcosa con vettore[i], vale a dire con l'elemento i-esimo
// i andrà da 0 fino all'ultimo indice del vettore
// quindi vettore[i] andrà dal primo all'ultimo elemento
```

Ora, prendendo int v[] = new int[] {40, 30, 20, 25, 17, 18, 21, 34}; come esempio, vediamo una serie di applicazioni dello scorrimento. Si consiglia di studiarle tutte perché trovano spessissimo applicazioni nella pratica.

```
//Stampa del vettore
for( int i=0;i < v.length;i++)
    System.out.println(v[i]);

//Somma del vettore
int sum = 0;
for( int i=0;i < v.length;i++)
    sum+=v[i];

// media
int avg = sum / v.length;

//Conteggio dei maggiorenni
int adults = 0;
for(int i=0; i<v.length; i++)
    if(v[i]> 17)
        adults++;

//ricerca del massimo
int max = v[0];
for(int i=1;i < v.length; i++)
    if(v[i]> max)
        max=v[i];

// allo stesso modo è semplice trovare l'età minima
// così come l'età media degli adulti, applicando
// gli algoritmi di conteggio, somma e ricerca (o "filtering")
```

Per terminare, notiamo che i è locale al for. Questo ci permette di non "sporcare" lo scope esterno al for con altre variabili che potrebbero andare in conflitto con quelle che abbiamo già "in circolo".

4.3 I vettori come insiemi e le operazioni su insiemi: map, filter e reduce

I vettori sono un modo primitivo e scomodo ma utile didatticamente di trattare il concetto di "insieme". Non trattiamo più un singolo intero, una singola String e neanche un singolo oggetto ma un insieme arbitrariamente lungo di elementi dello stesso tipo. Come abbiamo visto, i cicli e in particolare lo scorrimento sono gli strumenti principe per trattare questa tipologia di dati.

Un vettore NON corrisponde al concetto matematico di insieme (quello semmai è Set, che vedremo in seguito), ma ci permette di cominciare a ragionare sulle problematiche relative. In particolare nella pratica ci capiterà spesso di dover *trasformare* insiemi, di dover *filtrare* insiemi e di dover *ridurre* insiemi a singoli numeri o elementi. Queste operazioni possono essere chiamate, con una certa approssimazione: map, filter e reduce.

4.4 Riduzione di un vettore a un valore (reduce)

Partiamo dal fondo. L'operazione di riduzione più comune è la somma, e la abbiamo appena vista, ma potrebbe essere divertente generalizzarla:

```
// ArrayHelper.java
class ArrayHelper
{
    public static int sum(int v[])
    {
        int res = 0;
        for(int i=0; i <v.length; i++)
            res+=v[i];
        return res;
    }
}
// main()
public static void main(String[] args)
{
    System.out.println(ArrayHelper.sum(new int[]{1, 2, 3}));
}
```

Stamperà 6. E' anche un buon momento per ripassare i metodi, il passaggio di parametri e la parola chiave "static", e spiegare cosa si intende con "Helper Class".

Il metodo sum riceve come parametro un vettore di interi, e ne restituisce la somma. In gergo questa viene detta una riduzione (reduce). Il metodo sum non appartiene un oggetto ArrayHelper, ma alla classe stessa. E in effetti il main non crea un oggetto ArrayHelper, utilizzando direttamente la classe.

Il vettore new int[]{1, 2, 3} viene creato solo per diventare il parametro del metodo (andrà a finire nel parametro v), e poi viene perso. In questo caso abbiamo usato la forma di creazione e riempimento contestuale.

Resta da chiarire il concetto Helper class? Una Helper class è un insieme di metodi, tipicamente statici, quindi di classe, che tornano comodi in vari progetti e in vari casi. In questo caso, ad esempio, abbiamo creato uno strumentino senza pretese che calcola la somma di un vettore di interi, e potrà essere riutilizzato in molti contesti (ad esempio, la somma dei click di un sito, memorizzati in un vettore).

Questa è una piccola Helper class a tema vettori (array, in inglese). Conterrà, alla fine dei lavori, qualche metodo di uso comune coi vettori.

4.5 Operazione di filtraggio di un vettore (filter)

Un'operazione altrettanto comune è la rimozione di alcuni elementi da un vettore, generando un sottoinsieme del vettore originario. Questo viene detto "filter", o filtering, e si traduce nell'applicare una condizione a tutti gli elementi del vettore, e nel tenere quelli che la soddisfano. I vettori sono estremamente scomodi per questo, in quanto per ricavare un vettore da un altro, prima devo sapere quante posizioni mi serviranno. Supponiamo di voler tenere tutte le cifre sotto una data soglia, e modifichiamo la nostra helper class:

```
// ArrayHelper.java
class ArrayHelper
{
    public static int sum(int v[])
    {
        int res = 0;
        for(int i=0; i <v.length; i++)
            res+=v[i];
        return res;
    }

    // under thres -> quelli sotto il limite
    public static int[] underThres(int v[], int thres)
    {
        int count = 0;
        int res[];
        for(int i=0; i <v.length;i++)
            if(v[i] <thres)
                count++;
        // ho dovuto contare quanti erano sotto la soglia
        res = new int[count];
        int k=0;
        for(int i=0; i <v.length;i++)
            if(v[i] <thres)
            {
                // l'elemento alla posizione i finisce
                // alla posizione k. La posizione cambia col filtro!
                res[k] = v[i];
                k++;
            }
        return res;
    }
}

// main()
public static void main(String[] args)
{
    int[] kids = ArrayHelper.underThres(new int[]{8, 20, 12}, 18);
    System.out.println(ArrayHelper.sum(kids) / kids.length);
}
```

Abbiamo calcolato l'età media dei minorenni, prima filtrando gli elementi e tenendo solo quelli sotto i 18 anni, ottenendo un altro vettore (kids). kids, partendo da quei dati, è [8, 12]. Applico a kids il metodo sum.

Notiamo che underThres, che è una operazione di filter, non restituisce un singolo valore, ma un intero vettore, mai più grande del vettore originario.

4.6 Operazione di trasformazione 1-1 di un vettore (mapping)

A volte vogliamo ottenere un vettore grande quanto quello originario ma trasformato. Questo significa applicare una trasformazione a tutti gli elementi del vettore, uno dopo l'altro. Ad esempio, potremmo voler applicare l'IVA (10%) a un vettore di prezzi di tipo double:

```
// ArrayHelper.java
class ArrayHelper
{
    public static int sum(int v[])
    {
        int res = 0;
        for(int i=0; i <v.length; i++)
            res+=v[i];
        return res;
    }

    // under thres -> quelli sotto il limite
    public static int[] underThres(int v[], int thres)
    {
        int count = 0;
        int res[];
        for(int i=0; i <v.length;i++)
            if(v[i] <thres)
                count++;
        //ho dovuto contare quanti erano sotto la soglia
        res = new int[count];
        int k=0;
        for(int i=0; i <v.length;i++)
            if(v[i] <thres)
            {
                // l'elemento alla posizione i finisce
                // alla posizione k. La posizione cambia col filtro!
                res[k] = v[i];
                k++;
            }
        return res;
    }

    public static double[] addPercentage(double v[], double percentage)
    {
        // il vettore ottenuto sarà grande quanto quello originale
        double res[] = new double[v.length];
        for(int i=0; i <v.length; i++)
            res[i] = v[i] * (1+percentage);
        // la posizione non cambia col mapping
        return res;
    }
}

// main()
public static void main(String[] args)
{
    double taxprices = ArrayHelper.addPercentage(new double[]{10, 20, 30});
}
```

taxprices sarà 11, 22, 33. Il vettore originario è stato mappato su un nuovo vettore.

4.7 Vettori di oggetti

Un vettore può contenere tipi primitivi ma anche oggetti complessi.

Non è concettualmente diverso dall'avere un vettore di String (sono anch'esse oggetti), ma può essere più complicato in termini di gestione e di riconoscimento dei tipi.

Diamo un'occhiata a questo codice, in cui applichiamo anche l'incapsulamento Person e per cui forniamo solo i getter:

```
// Date.java
public class Date
{
    int day,month,year;
}

// Person.java
public class Person
{
    private String name,surname;
    private Date dateofbirth;
    //data in cui ha preso la patente
    private Date datedrivinglicense;

    public String getName()
    {
        return name;
    }

    public String getSurname()
    {
        return surname;
    }

    public String getDatedrivinglicense()
    {
        return datedrivinglicense;
    }

    public String getDateofbirth()
    {
        return dateofbirth;
    }
}

public static void main(String args[])
{
    // potete dedurre i costruttori da questa riga
    Person p = new Person("F", "P", new Date(05, 02, 1980), new Date(01, 06, 2000));
}
```

E analizziamo i tipi:

- p è di tipo Person
- p.dateofbirth è il cinque Febbraio 1980, ed è di tipo Date
- p.datedrivinglicense è primo giugno 2000, ed è di tipo Date. Lo ottengo tramite getDatedrivinglicense(), essendo datedrivinglicense private, quindi non visibile
- p.datedrivinglicense.year è 2020, ed è di tipo int. Per arrivarci da fuori passo per getDatedrivinglicense(): p.getDatedrivinglicense().year

Ora passiamo allo step successivo, un insieme (un vettore) di Person:

```
public static void main(String args[])
{
    Person a = new Person("F", "P", new Date(05, 02, 1980), new Date(01, 06, 2000));
    Person b = new Person("A", "B", new Date(05, 02, 1990), new Date(01, 06, 2015));
    Person c = new Person("C", "D", new Date(04, 03, 2000), new Date(01, 06, 2018));

    Person[] people = new People[]{a,b,c};
    //quante persone hanno preso la patente dopo il 2010? è una operazione
    //di filtraggio e riduzione, un conteggio. La faccio direttamente qui
    int count = 0;
    for( int i=0; i<people.length; i++)
        if(people[i].getDatedrivinglicense().year>2010)
            count++;

    System.out.println(count);
}
```

Studiamo brevemente il tipo dell'espressione nell'if.

- people: Person[], vettore di Person
- people[i] : Person, singola persona, alla posizione i del vettore
- people[i].datedrivinglicense : Date, oggetto date della persona alla posizione i del vettore
- people[i].datedrivinglicense.year : int, primitivo, anno di conseguimento della patente della persona alla posizione i del vettore people

Occorre sviluppare una familiarità con questa struttura a "scatole cinesi", o se preferite "ad albero" dell'informazione. E' la rappresentazione nel codice di un rapporto di **composizione**. Un vettore è composto di molti oggetti, in questo caso di Person. Person è composto anche di due date. Ogni data è composta di giorno, mese, anno. E così via.

5 - L'oggetto aggregatore nell'architettura del sistema

5.1 La metafora del motore

Da un punto di vista matematico, un programma è una funzione che trasforma l'input in output: $O = P(I)$, dove I ed O sono insiemi.

Dal punto di vista del codice, un programma è un insieme di istruzioni ordinate divise in vari files.

Dal punto di vista del progettista, un programma è un insieme di *componenti* che interagiscono fra loro per arrivare a un risultato. Questa è la prospettiva dell'ingegneria del software ed è quella che anima la filosofia di Java. Un componente sarà quasi sempre un oggetto, ma potrebbe anche essere direttamente una classe, come abbiamo visto con ArrayHelper.

Noi vedremo il programma come un *motore*, costituito da vari pezzi interconnessi, ciascuno incaricato di svolgere una, e una sola, funzione. Diremo che ogni pezzo ha una *responsabilità*, quella di offrire un determinato *servizio* al resto del mondo. Questo viene anche detto principio di responsabilità o *separazione dei compiti* (separation of concerns, SOC).

Torniamo alle prime lezioni. Pensiamo a Scanner. Scanner ha un solo compito: leggere. Non si occupa di stampe o di calcoli avanzati. Quella è stata la responsabilità del main, che si è anche appoggiato a System.out.println quando si è trattato di stampare.

Negli ultimi lavori illustrati in questo testo, abbiamo notato una suddivisione dei compiti simile: l'oggetto di classe Person aveva il compito di memorizzare i dati di una persona e in alcuni casi di ragionarci sopra (calcolarne l'età, ad esempio), mentre la classe del main() si limitava a fare domande e a stamparne i risultati, anche se in qualche caso si è occupato anche di fare dei calcoli.

C'era una interazione fra chi ragionava e memorizzava (Person) e chi usava quei dati (main()). Si chiama, correttamente, rapporto di uso, e ne abbiamo già parlato. Ora ci è più chiaro che anche main() ha le sue responsabilità.
E' lecito chiedersi come suddividere le responsabilità. Non c'è una risposta definitiva, ma ci sono delle linee guida:

- L'oggetto risponde di se stesso. Nel nostro caso, quei calcoli e quei dati che riguardano una Person vanno nella classe (e poi nell'oggetto) Person. L'oggetto si occupa solo di sé e di niente altro - non parla direttamente con l'utente, e non conosce neanche gli altri oggetti del suo tipo a meno che non gli vengano passati (public boolean older(Person other))
- il main() interagisce con l'utente e chiede agli altri oggetti le risposte. Chiederà agli oggetti di tipo Person quanto sono vecchi, ad esempio. Il main cercherà di non fare calcoli se non è strettamente necessario.

Notiamo che manca un anello alla nostra architettura. Fino ad ora, i calcoli sui vettori, i calcoli sugli insiemi, sono stati fatti dal main. Ora stiamo dicendo che il main sarebbe meglio non facesse calcoli, quindi neanche quelli sugli insiemi. Chi dovrebbe occuparsene?

5.2 L'oggetto aggregatore

I calcoli che riguardano il singolo verranno eseguiti dal singolo stesso. Quei calcoli che riguardano un gruppo invece verranno eseguito da un oggetto che conterrà un vettore di singoli. Un oggetto che contiene un vettore di altri oggetti complessi viene detto "aggregatore" (e la classe viene detta "aggregatrice").

Una classe in grado di gestire diverse Person, magari a fini statistici, potremmo strutturarla così:

```
public class Census
{
    //Censimento, in inglese
    Person[] content;

    public Census(Person[] content)
    {
        this.content = content;
    }
}
```

Questa è la struttura base per la creazione di un oggetto aggregatore. L'oggetto della classe Census richiederà, per funzionare, un vettore di Person, il suo content. Una volta ricevuto, lo salverà nel proprio *stato*. Successivamente, sarà l'oggetto di tipo Census a fare i calcoli. main() userà un oggetto census, limitandosi a "fargli delle domande".

Partiamo con qualcosa di semplice. Facciamo in modo che un oggetto di classe Census possa calcolare l'età media delle persone al suo interno. E' qualcosa che la singola Person **non poteva fare**. Un oggetto Person conosce solo se stesso, non gli altri. Di conseguenza deve occuparsene qualcun altro, qualcuno che conosca tutti gli oggetti Person. Questo qualcuno è l'oggetto di classe Census, una volta che avremo scritto il metodo corrispondente:

```
// Census.java
public class Census
{
    //Censimento, in inglese
    Person[] content;

    public Census(Person[] content)
    {
        this.content = content;
    }

    public double averageAge()
    {
        double sum = 0;
        for(int i=0;i < content.length; i++)
            sum+=content[i].getAge();
        //content[i] è una delle Person che gli abbiamo fornito
        //l'oggetto census dispone al suo interno di una lista
        //arrivata dall'esterno tramite il costruttore
        return sum / content.length;
    }
}

//Main.java
public static void main(String[] args)
{
    //un nuovo costruttore, la cui implementazione si lascia
    //allo studente
    Person a = new Person("F", "P", "05/02/1970");
    Person b = new Person("A", "B", "01/01/1990");
    Person c = new Person("C", "D", "01/01/2010");
    Census census = new Census(new Person[]{a,b,c});
    System.out.println("The average age is:"+census.averageAge());
    // stamperà 30
}
```

Questa suddivisione del lavoro è ottimale. Main si occupa solo di fare le richieste e gli altri elementi del programma gli rispondono. Notiamo che:

- main(), metodo statico della classe Main, usa un oggetto di tipo Census, census
- census contiene al suo interno, e usa, un vettore di tipo Person[]
- il vettore stesso contiene, quindi usa, 3 oggetti di tipo Person

Adesso abbiamo un criterio per definire **dove** mettere i metodi.

Ciò che riguarda un solo oggetto, più eventuali parametri in entrata, andrà in Person. La responsabilità di un oggetto di classe Person è di memorizzare i dati di una persona e di eseguire principalmente i calcoli che riguardano lui stesso. Ciò che riguarda un gruppo di Person andrà

scritto nella classe Census (per averlo nell'oggetto census). L'oggetto di classe Census contiene i dati di n oggetti, il nostro intero archivio di riferimento. main() si limita a interagire da un lato con l'utente, dall'altro con census.

5.3 Costruire la classe dell'oggetto aggregatore

Cominciamo definendo quello che ci interessa sapere. Supponiamo di voler classificare il nostro bacino demografico in base al genere, alla fascia di età e alla professione. Sarà necessario modificare la classe Person (ad ora, non abbiamo il genere), e sviluppare dei metodi adatti in Census. Aggiungiamo quindi String gender in Person (modificando il costruttore di conseguenza), e modifichiamo Census.java in questo modo.

```
// Census.java
public class Census
{
    //Censimento, in inglese
    Person[] content;

    public Census(Person[] content)
    {
        this.content = content;
    }

    // classificazione per genere: restituisco un vettore
    // con i soli uomini o le sole donne
    // il mio parametro, gender, è il genere desiderato
    // questo è un filtro parametrizzato. Il parametro è gender
    // lo confronto col getter di gender di Person, che avremo scritto in precedenza
    public Person[] byGender(String gender)
    {
        int count = 0;
        for( int i = 0 ; i<content.length; i++)
            //se il genere è quello cercato, lo conto
            if(content[i].getGender().equals(gender))
                count++;
        Person[] res = new Person[count];
        int k = 0;
        for(int i =0 ;i <content.length; i++)
            if( content[i].getGender().equals(gender))
            {
                res[k] = content[i];
                k++;
            }
        return res;
    }
}
```

Si tratta di un filtro standard. Notiamo che questo metodo produce un vettore e lo restituisce (al main() in questo caso). Notiamo che per ora non abbiamo parlato di professione o di età. **Notiamo anche che l'oggetto di classe Census non stampa niente**. In effetti, conviene prendere una buona abitudine: solo il main() stampa, solo il main usa la tastiera.

5.4 Il ruolo del main()

Ma cosa ci farà ora il main, con questo census? **Il main() fa' da tramite fra l'utente e gli altri oggetti del programma.**

```
// main.java
public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);
    //Ometto la creazione delle persone per brevità
    //Person a = ...;
    //
    Census census = new Census(new Person[]{a,b,c,d,e});
    System.out.println("Welcome to Census Main");
    do
    {
        System.out.println("Insert command, Q to quit");
        cmd = keyboard.nextLine();
        switch(cmd.toUpperCase())
        {
            case "G":
            {
                //ricerca per genere
                System.out.println("Insert M for males, F for Females");
                String gender = keyboard.nextLine();
                //i dati vengono calcolati da census. E' lui che applica il filtro
                Person[] results = census.byGender(gender);
                //io (main) mi limito a stampare
                for(int i = 0 ; i < results.length; i++)
                    System.out.println(results[i]);
            }
            break;
            //totali per genere
            case "GT":
            {
                System.out.println("Known people:"+census.content.length);
                System.out.println("Males:"+census.byGender("M").length);
                System.out.println("Females:"+census.byGender("F").length);
            }
            break;
            case "Q":
                System.out.println("Goodbye");
        }
    }while(!cmd.equals("Q"));
}
```

L'oggetto di classe Census produce *vettori*, e in futuro potrà anche produrre totali (operazioni di riduzione), a seconda dei bisogni del cliente finale. E' census a conoscere tutte le persone, e quindi a poter calcolare risultati di insieme. Ricopre il ruolo che nei nostri primi esercizi è stato ricoperto da main(), ma è una cattiva scelta di architettura lasciare troppe responsabilità a main. Lo rende ingestibile, e rende difficile modificare ed estendere il progetto in seguito.

Il metodo main() si occupa di interagire con l'utente (ha la keyboard) e di stampare i risultati che gli arrivano da census. Non fa molto altro, e ci va bene così.

Resta in realtà un ultimo punto che non gradiamo: main() deve creare gli oggetti Person per passarli a census sotto forma di vettore. Sarebbe molto più interessante qualcosa di questo tipo:

```
Census census = new Census("census.txt");
```

Il che ci porta alla fine del percorso di *refactoring* (migliorare il come senza cambiare il cosa) di questo progetto che in realtà abbiamo già visto svolto con altri strumenti. Main delega all'oggetto di classe Census un ultimo compito: aprire un file di testo e ricavarne degli oggetti da usare. E' una scelta sensata, anche se non ancora ottimale, ma è quella che vedremo a breve.

5.5 Oggetto aggregatore con lettura da file

Supponiamo di avere il seguente contenuto in census.txt:

```
3
F,P,05/02/1980,Teacher,M
A,B,06/03/1990,Teacher,F
T,M,06/06/2002,Student,F
```

Notiamo che ogni riga di questo file, tranne la prima, corrisponde allo stato di un oggetto persona. Possiamo trovare nome, cognome, data di nascita e genere. Una riga è sufficiente per costruire un oggetto Person.

E' sbagliato dire che una riga è l'oggetto. E' corretto dire che possiamo trasformare la riga in un oggetto. Essendo tre righe dopo la prima, non una sola, la loro traduzione naturale in Java è un vettore di tre elementi, ed è naturale dare a Census, che si occupa di lavorare su vettori di Person, la possibilità di caricarle da un file. In questo caso, tramite un costruttore apposito:

```
// Census.java, completo
public class Census
{
    //Censimento, in inglese
    Person[] content;

    //Primo costruttore. Riceve il vettore di Person
    public Census(Person[] content)
    {
        this.content = content;
    }

    //Secondo costruttore. Ricava il vettore da un file
    public Census(String filename)
    {
        try
        {
            Scanner reader = new Scanner(new File(filename));
            int n = Integer.parseInt(reader.nextLine());
            // a questo giro, sono io (Census) a creare il vettore
            // il vettore di Person non mi viene dato. Mi viene dato
            // il file da cui leggerlo, e io procedo a crearlo
            content = new Person[n];
            for(int i=0;i < n; i++)
                content[i] = rowToPerson(reader.nextLine());
            //il costruttore di appoggia a un altro metodo
            //che trovate sotto, e che di mestiere nella vita
            //riceve una stringa come quella che ho appena letto dal file
            //(reader.nextLine())
            //la spezza e la converte in una persona
            //la persona appena creata viene messa alla posizione i
            //di content.
            reader.close();
        }
        catch(Exception e)
        {
            //qualcosa è andato storto. Termino
            System.out.println("Problems opening "+filename+". Terminating");
            e.printStackTrace();
            System.exit(-1);
        }
    }

    // entra una riga del file, esce un oggetto di tipo Person
    // questo metodo viene usato dal costruttore
    public Person rowToPerson(String row)
    {
        String[] parts = row.split(",");
        String name = parts[0];
        String surname = parts[1];
        String date = parts[2].split("/");
        //spezzo la data per ricavarne le parti per ricostruire
        //l'oggetto Date
        int day = Integer.parseInt(date[0]);
        int month = Integer.parseInt(date[1]);
        int year = Integer.parseInt(date[2]);
        String profession = parts[3];
        String gender = parts[4];
        // ho letto una riga dal file, e la ho trasformata in un
        //oggetto di tipo persona
        return new Person(name, surname, new Date(day,month,year),profession, gender);
    }
}
```

```

    }

    public Person[] byGender(String gender){...}

    //aggiungiamo anche il filtro per fasce di età
    public Person[] byAgeRange(int minage, int maxage)
    {
        int count = 0;
        for( int i = 0 ; i<content.length; i++)
            //se il genere è quello cercato, lo conto
            if(content[i].getAge() >= minage && content[i].getAge() <= maxage)
                count++;
        Person[] res = new Person[count];
        int k = 0;
        for(int i =0 ;i <content.length; i++)
            if(content[i].getAge() >= minage && content[i].getAge() <= maxage)
            {
                res[k] = content[i];
                k++;
            }
        return res;
    }
}

```

Usando la versione nuova e migliorata di Census, main non ha bisogno di creare oggetti Person, ma può lavorare passando a census solo il nome del file (una stringa) che lui procederà ad importare in un vettore. Successivamente, census sarà in grado di rispondere alle nostre domande ("fare domande a census" significa chiamare i suoi metodi), o per essere più precisi, alle domande di main(). Main si occuperà poi di riportarci le risposte, curando la comunicazione con l'utente finale.

Riportiamo di seguito una versione di main() di esempio con le ultime modifiche:

```

// main.java con l'ultimo componente Census
// e la ricerca per fascia di età
public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);
    Census census = new Census("census.txt");
    System.out.println("Welcome to Census Main");
    do
    {
        System.out.println("Insert command, Q to quit");
        cmd = keyboard.nextLine();
        switch(cmd.toUpperCase())
        {
            case "G":
            {
                //ricerca per genere
                System.out.println("Insert M for males, F for Females");
                String gender = keyboard.nextLine();
                //i dati vengono calcolati da census. E' lui che applica il filtro
                Person[] results = census.byGender(gender);
                //io (main) mi limito a stampare
                for(int i = 0 ; i <results.length; i++)
                    System.out.println(results[i]);
            }
            break;
            //totali per genere
            case "GT":
            {
                System.out.println("Known people:"+census.content.length);
                System.out.println("Males:"+census.byGender("M").length);
                System.out.println("Females:"+census.byGender("F").length);
            }
            break;
            case "A":
            {
                System.out.println("Insert min age");
                int minage = Integer.parseInt(keyboard.nextLine());
                System.out.println("Insert max age");
                int maxage = Integer.parseInt(keyboard.nextLine());
                // notate che questa variabile results è locale a questo blocco di codice
                // in questo modo non va in conflitto con la variabile results del blocco "G"
                Person[] results = census.byAgeRange(minage, maxage);
                for(int i = 0 ; i <results.length; i++)
                    System.out.println(results[i]);
            }
            break;
            case "Q":

```

```
        System.out.println("Goodbye");
    }
}while(!cmd.equals("Q"));
}
```

6 - Eccezioni

6.1 Gestione basilare delle eccezioni

Possiamo definire eccezione un "errore durante il flusso del programma", vale a dire una interruzione del normale flusso di esecuzione del programma, a seguito di una istruzione che non è riuscita a completare correttamente la sua esecuzione.
Vediamo un esempio:

```
int a = Integer.parseInt("A");
```

Il metodo statico parseInt della classe Integer non riuscirà a convertire il valore "A" in un numero, e segnalerà correttamente una NumberFormatException. E' un errore molto comune, sia parlando con l'utente che leggendo da file.
Di fronte a una eccezione, un metodo si trova di fronte a due possibilità:

- propagare l'eccezione, lasciando che sia il chiamante (o qualcun altro) a gestirla
- gestirla lui stesso

Integer.parseInt() sceglie di propagare l'eccezione, giustamente. Deve informare il chiamante che la chiamata non è andata a buon fine, di modo che il chiamante possa decidere cosa fare. Supponendo di dover chiedere un numero all'utente, main() *dove* sapere che l'input non è andato a buon fine, di modo da poter ripetere la domanda. In questo caso parseInt() propaga l'eccezione, mentre main() la gestisce.

La gestione viene effettuata utilizzando una struttura try-catch, che somiglia in qualcosa a un if-else:

```
//main.java
boolean valid = false;
do
{
    try
    {
        int n = Integer.parseInt(keyboard.nextLine());
        valid = true;
    }
    catch(NumberFormatException e)
    {
        System.out.println("Invalid number");
    }
}
while(!valid);
```

Il blocco try è il blocco di "esecuzione normale". Se va tutto bene, verrà eseguito l'intero blocco try e il catch verrà saltato, un po' come accade con if-else. Se invece si verifica una eccezione (e quella eccezione in particolare, come vedremo in seguito), il blocco try terminerà alla riga a cui si è verificata l'eccezione e verrà eseguito il blocco catch, che in questo caso si limita a stampare il messaggio "Invalid number".

Ma cosa significa quel catch(NumberFormatException e)?

Il metodo parseInt non si è limitato a segnalare un errore. Ha impacchettato i dettagli dell'errore creando un oggetto (NumberFormatException è un tipo, quindi e è un oggetto di classe NumberFormatException) che ha poi passato al chiamante (in questo caso main()). In questo modo main() è "informato dei fatti" e può gestire al meglio l'eccezione.

Due metodi sempre presenti nell'oggetto Exception (che non ha bisogno di chiamarsi e) e molto comodi sono e.printStackTrace(), che ricostruisce tutto l'iter che ha portato all'eccezione, ed e.getMessage(), che restituisce un messaggio di errore riassuntivo su quanto accaduto. In fase di sviluppo tendiamo a mettere e.printStackTrace() in tutti i catch, per capire davvero cosa è successo. La stack trace (in sostanza, l'elenco delle chiamate a metodo che ha portato all'eccezione) merita un discorso a parte, che faremo in seguito.

6.2 Propagazione delle eccezioni e individuazione del gestore

Torniamo all'esempio di Census. Census può caricare le proprie persone da un file, ma il file potrebbe non esistere. Nella prima versione che abbiamo scritto, se non trova il file Census termina l'intero programma (System.exit(-1)), il che è qualcosa di aggressivo e insensato. Ricordiamoci che Census è "al servizio" di main(). In questo momento, l'oggetto di classe Census sta uccidendo se stesso e il main(). Sarebbe più corretto segnalare al main() che non ha trovato il file, e a quel punto la palla tornerebbe al main. Questo significherebbe *propagare*, non gestire, quella specifica eccezione, vale a dire FileNotFoundException:

```
// Census.java, riporto il solo costruttore
public Census(String filename) throws FileNotFoundException
{
    Scanner reader = new Scanner(new File(filename));
    int n = Integer.parseInt(reader.nextLine());
    // a questo giro, sono io (Census) a creare il vettore
    // il vettore di Person non mi viene dato. Mi viene dato
    // il file da cui leggerlo, e io procedo a crearlo
    content = new Person[n];
    for(int i=0;i <n;i++)
        content[i] = rowToPerson(reader.nextLine());
    //il costruttore di appoggia a un altro metodo
    //che trovate sotto, e che di mestiere nella vita
    //riceve una stringa come quella che ho appena letto dal file
    //((reader.nextLine())
    //la spezza e la converte in una persona
    //la persona appena creata viene messa alla posizione i
    //di content.
    reader.close();
}
```

La clausola throws viene aggiunta al nome del metodo e ne fa parte integrante. Il metodo, in questo caso un costruttore, avvisa il chiamante (chi utilizzerà il metodo) che potrebbe verificarsi un errore, e che lui ritiene di non avere la *competenza* per gestirlo. Il metodo Census, costruttore dell'oggetto di classe Census, propaga l'eccezione, non la gestisce. E' questo il significato di "throws".

Per essere ancora più precisi, l'unica eccezione che dichiara di non gestire è FileNotFoundException, una fra le tante possibili eccezioni. In teoria, Census dovrebbe essere in grado di gestire gli altri casi senza disturbare il chiamante.

La palla ora passa a main(), che può decidere di gestire l'eccezione (la strada giusta) o di propagarla a sua volta. In realtà, in main() è sbagliato propagare le eccezioni. Significa che il programma intero andrà in crash alla prima eccezione. main(), o qualcuno prima di lui, dovrà gestire tutte le eccezioni se vogliamo evitare i crash di sistema.

Come diventa main(), ora che Census lo costringe a gestire una eccezione? Riportiamo le prime righe:

```
public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);
    String filename = "census.txt";
    boolean valid = false;
    // devo dichiararlo fuori dal while, per poterlo usare fuori.
    Census census;
    while(!valid)
    {
        try
        {
            census = new Census(filename);
            valid = true;
        }
        catch(FileNotFoundException e)
        {
            //Il costruttore propaga questa eccezione. Io sono costretto a gestirla
            System.out.println("File "+filename+" not found. Could you specify a different name?");
            filename = keyboard.nextLine();
        }
    }
    //Il resto prosegue uguale
```

Questa può essere una gestione più intelligente dell'eccezione. Nel caso in cui il file non fosse presente, il main() chiederà all'utente di inserire un file diverso, e così via fino a trovare un archivio valido. Forse sarebbe anche il caso di permettere all'utente di uscire, magari lasciando il filename vuoto, ma dovrebbe rendere l'idea.

E' main() ad avere l'autorità per gestire questa eccezione. Sarebbe stato sbagliato farlo fare a Census, anche perché census non dovrebbe parlare direttamente con l'utente (quello è il ruolo del main()).

I più attenti si staranno chiedendo come mai sono costretto a gestire FileNotFoundException (Eclipse mi impedisce di avviare il programma altrimenti), mentre non ero costretto a gestire, ad esempio, NumberFormatException su ogni parseInt, anche nei programmi passati. E' la differenza fra eccezioni checked (che vanno gestite o propagate in maniera esplicita) e unchecked (che non devono necessariamente essere gestite o propagate, ma lo si può fare optionalmente).

In generale, gli errori più comuni ed evitabili tramite normali if (ad esempio NullPointerException) sono unchecked, mentre gli errori più specifici sono checked. La scelta è fatta dal programmatore che scrive l'eccezione, perché in effetti anche le eccezioni sono definibili dal programmatore, come vedremo in seguito.

6.3 Try con catch multipli e ordine di gestione

Un blocco try potrebbe generare più di una tipologia di errore, e avere di conseguenza diversi blocchi catch, ciascuno con una propria gestione separata. Prendiamo questo codice:

```
String s[] = new String[] {"123", "100", "60A", null};  
  
for(int i=0;i <s.length;i++)  
{  
    System.out.println("Digits:"+s[i].length()+" value:"+Integer.parseInt(s[i]));  
}
```

Stampo la lunghezza del numero e poi il suo valore, parsato. Questo codice può generare due tipologie di errore diverse: NumberFormatException quando arriverà a "60A" (non potrà parsare) e NullPointerException quando arriverà a null. Potremmo scegliere due fornire due gestioni diverse, e due avvisi diversi:

```
String s[] = new String[] {"123", "100", "60A", null};  
  
for(int i=0;i <s.length;i++)  
{  
    try  
    {  
        System.out.println("Digits:"+s[i].length()+" value:"+Integer.parseInt(s[i]));  
    }  
    catch(NumberFormatException e)  
    {  
        System.out.println("Invalid number "+e.getMessage());  
    }  
    catch(NullPointerException e)  
    {  
        System.out.println("Warning: null value found, skipping term");  
    }  
}
```

Questo programma produrrà il seguente output:

```
Digits:3, value:123  
Digits:3, value:100  
Invalid number For input string: "60A"  
Warning: null value found, skipping term
```

E' stato eseguito solo uno dei blocchi catch, quello relativo all'eccezione verificatasi.

Portiamo il tutto al livello successivo, creando un errore di logica nel programma. Modifichiamo il for in "for(int i=1; i <= s.length; i++)". In questo modo avremo un'altra eccezione, ArrayIndexOutOfBoundsException, visto che il for arriverà oltre la fine del vettore (l'ultimo elemento è s.length-1). Invece di gestire specificamente quell'errore, posso gestire l'errore Exception che è un caso più generale. Nella pratica, è il "bisogno" di ArrayIndexOutOfBoundsException. Vedremo in seguito cosa significa nello specifico.

```
String s[] = new String[] {"123", "100", "60A", null};  
  
for(int i=0;i <= s.length;i++)  
{  
    try  
    {  
        System.out.println("Digits:"+s[i].length()+" value:"+Integer.parseInt(s[i]));  
    }  
    catch(NumberFormatException e)  
    {  
        System.out.println("Invalid number "+e.getMessage());  
    }  
    catch(NullPointerException e)  
    {  
        System.out.println("Warning: null value found, skipping term");  
    }  
    catch(Exception e)  
    {  
        System.out.println("Unspecified exception on the element, skipping");  
    }  
}
```

L'output di questo programma sarà:

```
Digits:3, value:123  
Digits:3, value:100
```

```
Invalid number For input string: "60A"
Warning: null value found, skipping term
Unspecified exception on the element, skipping
```

Il catch(Exception e) ha gestito ArrayIndexOutOfBoundsException, ma non ha gestito NullPointerException né NumberFormatException. Exception è un caso più generale di entrambi, e normalmente potrebbe gestire anche loro ma abbiamo scritto prima i casi specifici. *I catch vengono valutati in ordine di scrittura.* Questo significa che dovremo sempre mettere prima i casi specifici, e in fondo i casi generali (tipicamente, Exception come estrema ratio). Mettendo per primo Exception gli altri catch non sarebbero mai raggiungibili, in quanto l'errore verrebbe intercettato ("consumato") dal blocco catch di Exception.

Per terminare il discorso sui catch multipli, possiamo abbinare la gestione di diversi tipi di eccezione usando una forma nota come multi-catch:

```
String s[] = new String[] {"123", "100", "60A", null};

for(int i=0;i <= s.length;i++)
{
    try
    {
        System.out.println("Digits:"+s[i].length()+" , value:"+Integer.parseInt(s[i]));
    }
    catch(NullPointerException | NumberFormatException e)
    {
        System.out.println("Warning: null value found or invalid number, skipping term");
    }
    catch(Exception e)
    {
        System.out.println("Unspecified exception on the element, skipping");
    }
}
```

In questo caso abbiamo associato la gestione di NullPointerException a quella di NumberFormatException, lasciando separata solo la gestione delle eccezioni "generiche". La sintassi è Eccezione1 | Eccezione2 e, ed è autoesplicativa.

6.4 Blocco finally

Il blocco finally è un elemento opzionale della struttura try-catch.

Quello che scriviamo nel blocco finally, che viene posto sempre dopo i catch, viene eseguito sempre, sia che sia stato eseguito il blocco try, sia che sia stato eseguito il blocco catch. Viene eseguito perfino nel caso in cui uno dei blocchi try o catch, abbiano eseguito un return.

Viene tipicamente utilizzato per chiudere file aperti (reader.close()) e in generale per "fare pulizia" dopo il lavoro dei try e dei catch. Modifichiamo il programma di prima e vediamo l'output:

```
String s[] = new String[] {"123", "100", "60A", null};  
for(int i=0;i<=s.length;i++)  
{  
    try  
    {  
        System.out.println("Digits:"+s[i].length()+" , value:"+Integer.parseInt(s[i]));  
    }  
    catch(NullPointerException | NumberFormatException e)  
    {  
        System.out.println("Warning: null value found or invalid number, terminating");  
        return;  
    }  
    catch(Exception e)  
    {  
        System.out.println("Unspecified exception on the element, skipping term");  
    }  
    finally  
    {  
        System.out.println("Moving to the next element");  
    }  
}
```

Che produce il seguente output:

```
Digits:3, value:123  
Moving to the next element  
Digits:3, value:100  
Moving to the next element  
Warning: null value found or invalid number, terminating  
Moving to the next element
```

Ho optato di terminare il programma quando trovo un valore non valido. I primi due valori sono validi, il terzo attiva il primo blocco catch (un blocco catch per eccezioni multiple), che stampa un messaggio di errore e invoca return. Nonostante il return termini il metodo, il finally viene comunque eseguito un'ultima volta.

In generale, mettiamo nel finally ciò che deve essere *sempre* fatto, quale che sia il flusso di esecuzione del programma.

6.5 Riepilogo della gestione delle eccezioni

- Una eccezione è un evento che interrompe la normale esecuzione del programma, un "errore". Java lo vede come un oggetto contenente tutte le informazioni sull'evento, per permettere al programmatore di gestirlo correttamente.
- Può essere gestita (try-catch) o propagata (throws) dal metodo in cui potrebbe verificarsi.
- Le eccezioni si dividono in checked e unchecked, a gestione o propagazione obbligatoria ed esplicita (FileNotFoundException) o implicita e opzionale (NullPointerException). Non siamo costretti a dire try-catch per NullPointerException, e neanche throws, mentre siamo costretti a fare una delle due cose per FileNotFoundException.
- Il metodo incaricato di gestire l'eccezione va deciso in fase di progettazione. Si rispetta il "principio di competenza": un metodo propaga l'eccezione se non sa come gestirla, lasciandone la gestione ai livelli superiori, e la gestisce se ritiene di avere sufficienti informazioni per farlo. Tutte le eccezioni dovrebbero essere gestite o evitate prima di arrivare a main o da main stesso.
- Un blocco try può avere tanti blocchi catch quante sono le tipologie di eccezione producibili nel blocco try stesso.
- I blocchi catch vengono *valutati* in ordine. Il primo blocco catch che riconosce l'eccezione avvenuta come propria verrà eseguito, e verranno saltati gli altri. In gergo, diremo che il blocco catch "consuma" l'eccezione. Per questa ragione, si mettono i catch specifici in cima, prima degli altri, e quelli più generici sul fondo.
- un blocco catch può gestire più tipi di eccezione diverse (multi catch)
- In fondo ai catch è possibile avere un blocco finally, che verrà eseguito sia in caso di try che in caso di catch

6.6 Caso di studio: l'oggetto di classe Census con gestione e propagazione di eccezioni

L'oggetto di classe Census funzionava fin tanto che il file census.txt era perfetto. Immaginiamo di avere il seguente census.txt:

```
5
F,P,02/1980,Teacher,M
F
T,M,06/06/2002,Student,F
```

Di tre righe se ne salva una. Tutte le righe dovrebbero avere, in un mondo ideale, nome, cognome, data (valida), professione e genere. Solo l'ultima delle righe rispetta questa condizione, e quando le righe arriveranno al metodo rowToPerson(String row) ci troveremo di fronte a varie eccezioni. Ad esempio, alla prima riga avremo ArrayIndexOutOfBoundsException, visto che la data richiede tre parti (giorno, mese, anno) e quella riga ne fornisce solo due.

Lo stesso varrà per la seconda (ci servono diversi campi, è presente solo una "F"). La terza verrà assorbita, poi avremo una brutta sorpresa: non ci sono più righe. Il file dichiarava 5 righe, ne ha presentate solo 3. Un'altra eccezione. Non solo: e se alla prima riga ci fosse una parola, non un numero?

Sceglieremo di non modificare rowToPerson, ma di renderci conto che ci sono diverse eccezioni unchecked che potrebbero mandare in crash il programma. Andiamo a vedere il costruttore che carica da file e vediamo come evitarlo:

```
// Census.java, classe per l'oggetto aggregatore di Person
// ho deciso che Census potrà propagare non solo FileNotFoundException, ma una classe di eccezioni generale che è "Exception"
// Stiamo dicendo che nella creazione di un census potranno verificarsi diversi errori
// che notificherò a main. Terò invece per me gli errori sulla singola riga (li "gestirò")
public Census(String filename) throws Exception
{
    //provo ad aprire il file. Se non ci riesco, il controllo torna a main()
    //ho dichiarato throws Exception. FileNotFoundException è un caso particolare di eccezione
    //e quindi lo sto propagando implicitamente
    Scanner reader = new Scanner(new File(filename));
    // cerco di leggere il numero di Person, ma se la prima riga non fosse numerica?
    // non posso continuare e lo notifico a main. Il main riceverà una NumberFormatException.
    // di nuovo, throws Exception include anche NumberFormatException
    int n = Integer.parseInt(reader.nextLine());
    content = new Person[n];
    for(int i=0;i <n;i++)
    {
        try
        {
            Person p = rowToPerson(reader.nextLine());
            //provo a creare la persona. Il metodo rowToPerson potrà darmi errori
            //nel qual caso la posizione i del vettore content resterà nulla!
            content[i] = p;
        }
        catch(Exception e)
        {
            //qui c'è una novità. La singola riga posso anche scartarla senza notificare il main
            //o almeno, faccio questa scelta
            //mi limito a stampare un messaggio di errore
            e.printStackTrace();
        }
    }
    reader.close();
}
//main.java
public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);
    String filename = "census.txt";
    boolean valid = false;
    // devo dichiararlo fuori dal while, per poterlo usare fuori.
    Census census;
    while(!valid)
    {
        try
        {
            census = new Census(filename);
            valid = true;
        }
        catch(Exception e)
        {
            //Il costruttore propaga questa eccezione. Io sono costretto a gestirla
            System.out.println("File "+filename+" not found or invalid data. Could you specify a different name?");
        }
    }
}
```

```
        System.out.println("error detail:");
        e.printStackTrace();
        filename = keyboard.nextLine();
    }
}

//per il main non è cambiato molto, ma invece di gestire una singola tipologia di errore ne gestisce una categoria
//molto più generica, Exception, che è un modo carino per dire "quasi qualunque errore". Se qualcosa è andato storto,
//stampiamo l'errore e chiediamo all'utente un altro filename.
//... il resto prosegue come prima.
```

7 - Altre strutture dati

7.1 Concetto informale di struttura dati

Raramente un programma lavora su variabili singole e slegate fra loro. Tipicamente lavoriamo su insiemi di variabili legate fra di loro da un qualche tipo di connessione o di rapporto. In effetti, quando raggruppiamo più variabili sotto una stessa "etichetta" stiamo creando una *struttura dati*.

Una *struttura dati* è la forma che diamo a un insieme di variabili collegate fra loro in qualche modo. Ne abbiamo viste, ad ora, due:

- L'oggetto (e per estensione la classe), che riunisce sotto uno stesso cappello più variabili, potenzialmente di tipi diversi, ciascuna col proprio nome. Le variabili sono legate dal fatto di appartenere allo stesso oggetto, e possiamo accedervi per nome (person.name). La classe definisce anche le regole di accesso al contenuto dell'oggetto, secondo le regole di encapsulamento di cui abbiamo parlato in precedenza, e ha regole automatiche per definire uno scope di oggetto e uno di classe.
- Il vettore, che riunisce sotto lo stesso cappello più variabili dello stesso tipo, identificate non tramite il nome ma tramite la posizione (v[3] -> quarto elemento del vettore v). Rispetto all'oggetto, il vettore offre il concetto di ordine delle variabili. Come l'oggetto, il vettore ha un contenuto finito di variabili, e questo non cambia.

Notiamo che le strutture dati si possono combinare fra loro. Abbiamo visto un vettore dentro un oggetto (content) che era a sua volta composto di oggetti (content era composto da n Person). Possiamo sempre innestare una struttura dati dentro un'altra, creando gerarchie di oggetti potenzialmente complesse.

Uno dei problemi da affrontare in fase di progettazione è la scelta della struttura dati adatta. Ci siamo già abituati a rappresentare i concetti del mondo reale, fisici o meno, con classi e oggetti, e abbiamo trattato liste e insiemi usando i vettori. In questo capitolo esploreremo alternative principalmente a questi ultimi.

7.2 La struttura dati List

Una lista può essere vista come una evoluzione di un vettore. In termini pratici, una lista è una sequenza ordinata di elementi dello stesso tipo, ma a differenza del vettore non è necessario dimensionarla: la lista nasce vuota e può espandersi o contrarsi a seconda delle necessità.

La sintassi per creare una lista, in questo caso di stringhe, è la seguente:

```
List<String> names = new ArrayList<String>();
```

I simboli < e > vengono detti "parentesi angolari", in questo caso, e servono a indicare il tipo dell'elemento della lista. In questo caso, la lista contiene oggetti di tipo String. Ci sono alcune precisazioni da fare prima di proseguire:

- Non esistono liste di tipi primitivi. Non posso scrivere List < int >. **Le liste contengono solo oggetti**. Vale ogni volta in cui useremo le parentesi angolari.
- List < > è il tipo formale della lista, e useremo sempre questo. ArrayList è il tipo concreto, un caso particolare di lista. I due tipi non sono identici ma sono *compatibili*. Tecnicamente diremo che un ArrayList è un List. Il resto del discorso deve essere rimandato al capitolo sulla ereditarietà, vale a dire al rapporto fra i tipi.
- Come per i vettori, le liste sono oggetti, con metodi e proprietà. names è un oggetto di tipo List < String >, names.get(0) è il primo elemento della lista, ed è di tipo Person

Ora confrontiamo vettori e liste con codice a cui siamo abituati:

```
// creazione
Person v = new Person[2];
List <Person> l = new ArrayList <Person> ();
// inserimento di un elemento
v[0] = "Ferdinando";
l.add("Ferdinando");
// inserimento di un secondo elemento
v[1] = "Pippo";
l.add("Pippo");
// inserimento di un terzo
// il vettore darà errore
// la lista no
v[2] = "Terzo";
l.add("Terzo");
// accedere al primo elemento:
System.out.println(v[0]);
System.out.println(l.get(0));

// rispettive lunghezze. length non cambia, size() si
System.out.println(v.length);
System.out.println(l.size());
```

7.3 Iterable e for-each

Tanto i vettori quanto le liste appartengono a una categoria (un super-tipo, in realtà) di nome Iterable.

Iterable indica un oggetto in grado di essere attraversato (su cui possiamo fare scorrimento) dal primo elemento all'ultimo. In particolare, gli oggetti di tipo Iterable possono essere usati all'interno di un comodissimo ciclo di nome for-each (in realtà, una variante del for), che scorre gli elementi dal primo all'ultimo senza dichiarare esplicitamente la posizione.

Vediamo gli algoritmi noti usando questo nuovo ciclo su oggetti iterable (per ora, liste e vettori):

```
String [] v = new String[]{"A","B","C"};
List <String> l = new ArrayList <String> ();
l.add("A");
l.add("B");
l.add("C");
// stampo tutti gli elementi

// "per ogni stringa s della lista l"
// s non è la posizione, è il singolo elemento del contenitore
for(String s:l)
    System.out.println(s);

// "per ogni stringa s del vettore v"
// s non è la posizione, è il singolo elemento del contenitore
for(String s:v)
    System.out.println(s);

// un esempio per gli interi
int n[] = new int[]{4, 5, 1};
int sum = 0;
// in questo caso i NON è la posizione (0,1,2) ma il valore (4,5,1)
// il for usato in questo caso non è il for standard, ma il for each
for(int i:n)
    sum+=i;
```

A questo punto combiniamo liste e for-each per realizzare un'operazione di filter (estrazione di sottoinsieme) standard:

```
List <Person> people = new ArrayList <Person> ();
// supponiamo che people sia stato riempito...
List <Person> adults = new ArrayList<Person> ();
for(Person person:people)
    if(people.isAdult(18))
        adults.add(person);
// Notiamo che adults è vuoto all'inizio, e si espande per accogliere gli elementi di people che rispettano la condizione.
```

7.4 Caso di studio: l'oggetto di classe Census con l'uso di liste e for-each

Cominciamo con una osservazione: non ci serve più sapere prima quanto è grande il vettore. Memorizzeremo le persone in una List < Person > content interna a Census, che si espanderà per contenere i dati. Il file quindi potrebbe essere il seguente (census.txt):

```
F,P,01/02/1980,Teacher,M  
F  
T,M,06/06/2002,Student,F
```

E proseguiremo modificando l'oggetto di classe Census. Per comodità, Census ora restituirà List < Person >, non vettori:

```
import java.util.List;  
import java.util.ArrayList;  
import entities.Person;  
  
public class Census  
{  
    //creo la lista direttamente qui, come proprietà di oggetto  
    List <Person> content = new ArrayList <Person>();  
    public Census(String filename) throws Exception  
    {  
        Scanner reader = new Scanner(new File(filename));  
        //non mi serve sapere quante righe avrò. Provo a leggerle tutte  
        while(reader.hasNextLine())  
        try  
        {  
            Person p = rowToPerson(reader.nextLine());  
            // content si dilata per accogliere il nuovo elemento, p, se siamo riusciti a crearlo  
            content.add(p);  
        }  
        catch(Exception e)  
        {  
            e.printStackTrace();  
        }  
        reader.close();  
    }  
  
    public List <Person> byAgeRange(int minage, int maxage)  
    {  
        List <Person> res = new ArrayList <Person>();  
        for(Person p:content)  
            if(p.getAge() > minage && p.getAge() < maxage)  
                res.add(p);  
        return res;  
    }  
  
    public List <Person> byGender(String gender)  
    {  
        List <Person> res = new ArrayList <Person>();  
        for(Person p:content)  
            if(p.getGender().equals(gender))  
                res.add(p);  
        return res;  
    }  
}  
  
// main.java. E' quasi identico  
// riporto solo parte del do-loop centrale  
// ...  
do  
{  
    System.out.println("Insert command, Q to quit");  
    cmd = keyboard.nextLine();  
    switch(cmd.toUpperCase())  
    {  
        case "G":  
        {  
            System.out.println("Insert M for males, F for Females");  
            String gender = keyboard.nextLine();  
            //stampo direttamente il risultato di census.byGender(gender)  
            for(Person p: census.byGender(gender))  
                System.out.println(p);  
        }  
        break;  
        //totali per genere  
        case "GT":  
        {  
    }
```

```

        System.out.println("Known people:"+census.content.size());
        System.out.println("Males:"+census.byGender("M").size());
        System.out.println("Females:"+census.byGender("F").size());
    }
    break;
    case "A":
    {
        System.out.println("Insert min age");
        int minage = Integer.parseInt(keyboard.nextLine());
        System.out.println("Insert max age");
        int maxage = Integer.parseInt(keyboard.nextLine());
        for(Person p:census.byAgeRange(minage,maxage))
            System.out.println(p);
    }
    break;
    case "Q":
        System.out.println("Goodbye");
    }
}
while(!cmd.equals("Q"));

```

Leggendo questo codice, e la sua semplicità, ci si potrebbe chiedere: perché usare il for normale e i vettori quando sembrano essere dominati completamente da liste e for each?

I vettori sono meglio ottimizzati, più veloci in termini di esecuzione. E' preferibile utilizzare vettori quando ci troviamo di fronte a insiemi di lunghezza fissa e immutabili, e quando vogliamo usare i tipi primitivi (int, double...). Il for classico, che scorre le posizioni, è effettivamente meno usato del for-each, ma è l'ideale quando, ad esempio, dobbiamo ordinare un vettore, e siamo costretti a ad assicurarc che l'elemento alla posizione i sia inferiore o superiore all'elemento alla posizione i-1. In generale, quando la posizione è importante utilizziamo il for standard, in tutti gli altri casi il for-each potrebbe essere preferibile.

7.5 Un cenno ai tipi boxati e il loro uso con le liste e le altre strutture dati

Abbiamo detto che non possiamo creare `List < int >`. Come creare una lista di interi quindi? Java dispone di versioni "a oggetti" dei dati primitivi. Si tratta di classi che definiscono oggetti contenenti un solo valore, un primitivo, ma il valore è comunque contenuto in un oggetto. In gergo vengono detti tipi "boxati", perché abbiamo creato una "scatola" attorno a un primitivo. Un oggetto di un tipo boxato gode di alcune regole particolari che ne facilitano la gestione. Vediamo qualche esempio:

```
int a = 6;
// un normale intero
Integer b = 6;
// il tipo boxato degli int
// è un oggetto!
// b ha problemi di identità: si vede sia come un intero che come un oggetto
System.out.println(b.equals(a));
System.out.println(b==a);
// posso usare sia equals che ==. e b dispone di una serie di metodi anche interessanti
Integer c = new Integer(6);
System.out.println(c==a);
// è come dire Integer c = 6;

// posso mettere un primitivo in un tipo boxato:
// e questo viene detto "boxing"
b = a+1;
// e viceversa, ridurre un tipo boxato a un primitivo
// "unboxing"
int d = b;
// il tutto viene eseguito in maniera trasparente.
```

Il famoso `Integer.parseInt()` è un metodo statico della classe `Integer`, che nel tempo libero serve anche per creare i tipi boxati di `int` ("interi che si credono oggetti"). Possiamo usare sia la classe `Integer` che gli oggetti di tipo `Integer`. Questo offre anche una soluzione al problema della lista di interi:

```
List <Integer> integers = new ArrayList <Integer> ();
integer.add(4);
// viene eseguito un boxing隐式. 4 sarebbe un primitivo, ma viene inserito un oggetto di tipo Integer
// la lista non può contenere primitivi.
// viceversa:
int sum = 0;
for(int i:integers)
    sum+=i;
//unboxing隐式
```

I tipi boxed sono **int - Integer, double - Double, float - Float, byte - Byte, long - Long, short - Short, boolean - Boolean, char - Character**. Per noi saranno principalmente un modo per usare al meglio le liste e altre *strutture dati*.

7.6 La struttura dati Set

Liste e vettori sono entrambi iterabili e condividono il concetto di insieme ordinato. Hanno un'altra caratteristica comune: ammettono duplicati. Posso inserire più volte lo stesso elemento nello stesso vettore o nella stessa lista. Non sempre questo è desiderabile. Supponiamo ad esempio di voler memorizzare una lista di interessi, rappresentati come stringhe. Non avrebbe senso aggiungere due volte lo stesso elemento nello stesso elenco, dovremmo impedirlo. Ci è anche indifferente conoscere l'ordine degli elementi, se non li consideriamo in ordine di "importanza".

La soluzione a questa tipologia di problema è la struttura dati *Set*, che corrisponde al concetto matematico di insieme in maniera rigida. Un insieme matematico non ha duplicati e non deve necessariamente avere un ordine (non tutti gli insiemi hanno una relazione d'ordine di interesse). La sintassi per la sua creazione somiglia a quella della lista:

```
Set<String> interests = new HashSet <String> ();
interests.add("G1");
interests.add("G1");
interests.add("A");
// stampereà 2
System.out.println(interests.size());
for(String i:interests)
    System.out.println(i);
// output: A, G1. Non ha rispettato l'ordine di inserimento
```

Di nuovo, Set è il tipo formale, HashSet quello concreto. HashSet è "un caso" di Set, e quindi è compatibile con Set (si rimanda al capitolo sull'ereditarietà per i dettagli), mentre non è vero il contrario: tutti gli HashSet sono Set, ma non tutti i Set sono HashSet. Una operazione estremamente comoda, presente sia sulle List che sui set, è il metodo contains(). Contains restituisce true o false a seconda che un elemento sia presente o meno in un insieme:

```
interests.contains("A");
// true

interests.contains("B");
// false
```

Set possiede inoltre metodi adatti a simulare le operazioni insiemistiche di base (intersezione, unione, differenza), come vedremo nell'esempio seguente.

7.7 Caso di studio: elenco delle professioni censite nell'oggetto di classe Census

Vogliamo generare una "lista" di professioni presenti negli oggetti Person dell'oggetto di classe Census, senza ripetizioni. Non si parla di una lista, quindi, ma di un insieme. Vogliamo tutte le professioni prese singolarmente, e non ci interessa in maniera particolare l'ordine.
Abbiamo già una lista con tutte le Person (content), ma le professioni si ripetono. Le estraiamo in questo metodo:

```
// siamo in Census.java
public Set <String> getProfessions()
{
    Set <String> res = new HashSet <String> ();
    // content arrivava da file
    for(Person p:content)
        res.add(p.getProfession());
    // anche se aggiungo "Teacher" mille volte l'insieme conterrà teacher una volta sola
    // gli insiemi rimuovono i duplicati
    return res;
}
```

Anche questa può essere letta come una operazione di riduzione. Partiamo da una lista di Person e ne ricaviamo un Set di String, collassando m oggetti di tipo Person su n String.

Ora supponiamo di voler verificare se abbiamo in archivio *tutte* le professioni che ci interessano, ad esempio Teacher, Student e Programmer. Potrebbero essere in archivio, potrebbero non esserci. Un modo per verificare la loro presenza potrebbe essere il seguente:

```
// siamo sempre in Census
public boolean hasAllProfessions()
{
    Set <String> archiveProfessions = getProfessions();
    return archiveProfessions.contains("Teacher") && archiveProfessions.contains("Student") && archiveProfessions.contains("Programmer");
}
```

Ma sarebbe più interessante caricare la lista delle professioni richieste da un altro file, e verificare poi se in census.txt sono presenti tutte:

```
// sempre in Census.java
// verifico di avere tutte le professioni nel file "filename"
// immaginiamo che il nostro filename contenga:
// Teacher
// Programmer
// Student
// Student
// Abbiamo due volte Student, ma non è un problema. Il Set elimina i doppiioni
public boolean hasAllProfessions(String filename) throws FileNotFoundException
{
    Set <String> required = new HashSet <String> ();
    Scanner reader = new Scanner(new File(filename));
    while(reader.hasNextLine())
        required.add(reader.nextLine());
    // in questo modo possiamo cambiare l'elenco delle professioni di interesse semplicemente cambiando il file
    // ora devo verificare di averle tutte. Utilizzo il metodo containsAll(s2), che lavora sui set
    // restituisce true se un set è contenuto nell'altro
    reader.close();
    // true se required (le professioni richieste, caricate da filename)
    // è contenuto in getProfessions() (le professioni trovate nel file census.txt)
    // vale a dire, se le professioni richieste sono un sottoinsieme, potenzialmente sovrapponibile, di quelle presenti
    // altrimenti false
    return getProfessions().containsAll(required);
}
```

In teoria, quando possiamo ammettere duplicati, utilizziamo le List o i vettori (tipicamente le prime). Quando non vogliamo duplicati, e non ci interessa l'ordine, Set può essere una scelta migliore. In pratica spesso si usano le liste per convenzione, a meno di non dovere eseguire lavori di insiemistica specifici.

7.8 La struttura dati Map

Le mappe (struttura Map) sono una struttura dati potente che ricorda in qualcosa un incrocio fra classi e liste. Una Map è definita come un *insieme di coppie chiave-valore*, con chiave di tipo K e valore di tipo V, potenzialmente diversi. Ogni mappa esprime il concetto matematico di *funzione*, mettendo in relazione una chiave k con un valore v. Le chiavi k non si possono ripetere, mentre i valori sì. La sintassi è:

```
Map <K , V> m = new HashMap <K , V>();
```

Map < K, V > è il tipo formale. HashMap è il tipo concreto. HashMap è un caso particolare di Map, ed è compatibile con Map, ma non viceversa. Tutte le HashMap sono Map ma non tutte le Map sono HashMap. Cerchiamo di chiarire con un esempio:

```
Map <String, String> p = new HashMap <String, String> ();
// Mappa, insieme di coppie. In questo caso, K = String, V = String.
// non posso inserire un valore per volta, ma sempre e solo una coppia
p.put("name", "Ferdinando");
p.put("surname", "Primerano");
p.put("age", "40");
// e per leggere?
// "stampa il valore della chiave age nella mappa (contenitore) p"
// anche se è un numero, lo vediamo lo stesso come una stringa
System.out.println(p.get("age"));
```

La mappa si può rappresentare come segue: {"name": "Ferdinando", "surname": "Primerano", "age": "40"}. Potrebbe ricordarvi un oggetto, ma non è esatto. In un oggetto, age sarebbe un int, qui deve necessariamente essere una stringa, perché abbiamo dichiarato V = String. Le coppie sono sempre e solo {String, String} in questo caso.

C'è anche un'altra differenza fondamentale rispetto a un oggetto: le classi dichiarano tutte le proprietà dell'oggetto in fase di creazione, e quindi sono sempre presenti, e al massimo vuote, e *non possono dichiararne altre*. L'oggetto Person ha solo quelle proprietà (name,surname,dateofbirth,profession,gender), e non posso aggiungerne, mentre posso scrivere:

```
p.put("hobbies", "java");
```

In questo modo ho arricchito di informazioni l'oggetto p, che pure da principio non era predisposto per memorizzare i miei hobbies (o qualunque altra cosa in realtà).

Un'altra grossa differenza rispetto agli oggetti è che non posso definire metodi extra all'interno delle mappe. Le mappe hanno i propri metodi - noi possiamo aggiungere coppie, non metodi.

La mappa che abbiamo visto sopra, Map < String, String >, è piuttosto comune e molto usata nella pratica, perché è adatta a rappresentare lo stato di una ampia categoria di oggetti. Molto spesso gli oggetti "viaggiano" sotto forma di Map, per essere poi ricostruiti. Noi abbiamo usato le stringhe, ma il principio è lo stesso.

Le applicazioni però sono svariate. Prendiamo il seguente esempio:

```
Map <String, Person> squadra = new HashMap ();
// K = String, V = Person. Questa mappa esprime una relazione fra una stringa e una o più persone.
// immaginiamo di avere le persone ferdinando, stella e pino (oggetti di tipo Person, non stringhe).
// ferdinando è portiere, pino è terzino sinistro e stella è terzino destro. Ferdinando fa anche da allenatore
// si tradurrebbe così:
squadra.put("portiere", ferdinando);
squadra.put("terzino sinistro", pino);
squadra.put("terzino destro", stella);
squadra.put("allenatore", ferdinando);
// la mappa squadra esprime il rapporto fra una stringa (un ruolo) e un Person
// notiamo che ferdinando è collegato sia a portiere che ad allenatore.
```

E se ora volessi sapere quanti anni ha il portiere? La sintassi sarebbe:

```
System.out.println(squadra.get("portiere").getAge());
```

squadra è la mappa (l'insieme delle coppie), get("portiere") prende il **valore** corrispondente alla **chiave** "portiere". Il valore corrispondente è l'oggetto ferdinando. Quindi squadra.get("portiere") è di tipo Person. Su quell'oggetto eseguo il metodo getAge().

In definitiva, il nostro scopo è organizzare i valori secondo le chiavi.

Come dicevamo prima, le chiavi non si possono ripetere (in effetti le chiavi sono memorizzate in un Set di K). Non potremmo avere due persone con lo stesso ruolo. Ipotizziamo che arrivi un nuovo allenatore, a fianco del precedente:

```
squadra.put("allenatore", vlad);
```

Abbiamo inserito la nuova coppia (K,V) = > (String,Person) = > ("allenatore", vlad), ma la chiave allenatore era già presente. Il risultato è che la vecchia coppia è stata sovrascritta, e ora è solo vlad l'allenatore. Di conseguenza *non possiamo utilizzare le mappe quando le chiavi si ripetono*. Perderemmo dei valori.

Di seguito riportiamo un esempio relativo all'utilizzo delle mappe. E' un programma che gestisce una mappa < String, String > di nome "facts" e che chiede all'utente, in maniera quasi psicanalitica, di parlare di sé, di aggiungere tasselli. Dove un "fatto" sia già noto il programma chiederà all'utente se vuole sovrascriverlo ("aggiornare la versione"), e a ogni iterazione stamperà tutto quello che sa dell'utente (il proprio keyset, e i rispettivi valori).

```
// Shrink.java
public static void main(String[] args)
{
    Map <String , String> facts = new HashMap <String, String>();
    Scanner keyboard = new Scanner(System.in);
    // un "factkey", una informazione sull'utente. Sarà la chiave della nostra mappa
    // ad esempio "height", o "weight", o "name"
    String factkey;
    // il valore corrispondente. Nelle mappe si lavora sempre in coppia
    String factvalues;
    // ad esempio "170", "70", "Ferdinando", o qualunque String
    System.out.println("Hello, this is a fact collector");
    // ciclo infinito
    do
    {
        System.out.println("Which fact? Type bye to exit");
        factkey = keyboard.nextLine();
        // break: terminazione anticipata del for!
        // si considera "poco elegante", ma in alcuni casi è preferibile
        if(factkey.equals("bye"))
            break;
        //me lo aveva già detto prima:
        //avevo già una coppia con quella chiave
        if(facts.containsKey(factkey))
        {
            //stampo il suo valore
            System.out.println("You said it was "+facts.get(factkey));
            System.out.println("Do you want to change that? Press enter to skip");
            String newvalue = keyboard.nextLine();
            //sovrascrivo il vecchio valore
            if(!newvalue.equals(""))
                facts.put(factkey, newvalue);
        }
        else
        {
            //non me ne aveva ancora parlato
            System.out.println("Ahh, new info. Pray tell");
            String newvalue = keyboard.nextLine();
            facts.put(factkey, newvalue);
        }
        //ora devo stampare le sue chiavi e i suoi valori: quello che so di lui
        System.out.println("What I do know about you after this:");
        // keySet -> l'insieme delle chiavi è un Set, quindi Iterable
        for(String key:facts.keySet())
            System.out.println(key+":"+facts.get(key));
        //per ogni chiave (key) dell'insieme delle chiavi(keyset)
        //della mappa fact, stampo la chiave e il valore correlato
    }
    while(true);
    keyboard.close();
}
```

7.9 Caso di studio: main() di Census multilingua con l'uso delle mappe

Un altro nome di Map è "dictionary", letteralmente dizionario. In effetti un dizionario è solo una mappa < String, String >, dove la chiave (la stringa a sinistra) è il termine in una lingua e a destra c'è una stringa con le sue possibili traduzioni. Volendosi far del male si potrebbe utilizzare una Map < String, String [] > in cui a ogni String collegheremmo un vettore di String, ma non è necessario.

Prendiamo il seguente file di testo, `italian.txt`, come esempio:

```
WELCOMEMESSAGE:Benvenuti in Census 1.0
ASKCOMMAND:Inserire un comando fra G (ricerca per genere), GT (totali per genere), A (ricerca per fascia di età), Q (per uscire)
ASKMINAGE:Inserire età minima
ASKMAXAGE:Inserire età massima
ASKGENDER:Inserire genere (M/F)
PEOPLE:Persone totali
MALES:Uomini
FEMALES:Donne
QUIT:Addio
```

Questo file è lo *stato* di una mappa < String, String >, vale a dire un insieme di coppie (6 in questo caso) che collegano una String a un'altra. La stringa a sinistra, che useremo come chiave, sarà il codice del messaggio. La stringa a destra sarà il valore stampato per l'utente. Questi dati, una volta caricati in una mappa, localizzeranno il programma in italiano.

Vedremo fra poco come, ma per ora definiamo un secondo file, `english.txt`:

```
WELCOMEMESSAGE:Welcome to Census 1.0
ASKCOMMAND:Insert a command between G (search by genre), GT (totals by genre), A (search by age range), Q (to quit)
ASKMINAGE:Insert min age
ASKMAXAGE:Insert max age
ASKGENDER:Insert genre (M/F)
PEOPLE:People
MALES:Men
FEMALES:Women
QUIT:Goodbye
```

Le chiavi sono le medesime ma sono collegate a valori diversi. Se leggeremo i dati da una mappa avremo il programma in italiano, altrimenti in inglese. Ora vediamo come:

```
// class main.java
public static void main(String[] args)
{
    Scanner keyboard = new Scanner(System.in);
    // questa mappa conterrà le traduzioni, caricate da un file
    Map <String, String> dictionary = new HashMap <String, String> ();
    // questa stringa conterrà la lingua, cioè il file da usare
    String language = "";
    //preparo un Set, un insieme, di lingue che conosciamo
    Set <String> availableLanguages = new HashSet <String> ();
    availableLanguages.add("italian");
    availableLanguages.add("english");
    // primo step: scelgo che file usare
    while(true)
    {
        System.out.println("Available languages: english, italian");
        System.out.println("Type a language");
        language = keyboard.nextLine();
        //ho questa lingua?
        if(availableLanguages.contains(language))
            break;
        else
            System.out.println("Language not recognized");
    }
    // definita la lingua, la devo caricare. Se per qualche ragione non trovo la lingua, termino
    Scanner reader = null;
    try
    {
        reader = new Scanner(new File(language+".txt"));
        while(reader.hasNextLine())
        {
            String[] parts = reader.nextLine().split(":");
            //la parte a sinistra è la chiave, la parte a destra è il valore
            dictionary.put(parts[0], parts[1]);
        }
    }
    catch(Exception e)
    {
        //non trovo il file o c'è un errore nel file. Termino
        System.out.println("Bad language file "+language+".txt. Terminating");
    }
}
```

```

        return;
    }
    finally
    {
        // in tutti i casi, chiudo il file aperto
        reader.close();
    }
    // elemento successivo: scelgo il "database"
    String filename = "census.txt";
    boolean valid = false;
    // devo dichiararlo fuori dal while, per poterlo usare fuori.
    Census census;
    while(!valid)
    {
        try
        {
            census = new Census(filename);
            valid = true;
        }
        catch(Exception e)
        {
            //Il costruttore propaga questa eccezione. Io sono costretto a gestirla
            System.out.println("File "+filename+" not found. Could you specify a different name?");
            filename = keyboard.nextLine();
        }
    }
    // ora il ciclo principale, che è completamente localizzato
    while(true)
    {
        //io stampo il valore della chiave ASKCOMMAND. Sarà in italiano o in inglese in base alla
        //scelta precedente del mio utente!
        System.out.println(dictionary.get("ASKCOMMAND"));
        String cmd = keyboard.nextLine();
        switch(cmd)
        {
            case "G":
            {
                System.out.println(dictionary.get("ASKGENDER"));
                String gender = keyboard.nextLine();
                for(Person p:census.byGender(gender))
                    System.out.println(p);
            }
            break;
            //totali per genere
            case "GT":
            {
                System.out.println(dictionary.get("PEOPLE")+":"+census.content.size());
                System.out.println(dictionary.get("MALES")+":"+census.byGender("M").size());
                System.out.println(dictionary.get("FEMALES")+":"+census.byGender("F").size());
            }
            break;
            case "A":
            {
                System.out.println(dictionary.get("MINAGE"));
                int minage = Integer.parseInt(keyboard.nextLine());
                System.out.println(dictionary.get("MAXAGE"));
                int maxage = Integer.parseInt(keyboard.nextLine());
                for(Person p:census.byAgeRange(minage,maxage))
                    System.out.println(p);
            }
            break;
            case "Q":
            {
                System.out.println(dictionary.get("QUIT"));
                return;
            }
        }
    }
}
}

```

Non facile da seguire: troppe graffe, troppa indentazione, troppi if e troppe ripetizioni. E' il momento di razionalizzare.

7.10 Caso di studio: refactoring del main di Census con metodi statici

Il main() è diventato decisamente troppo grande e sgradevole da gestire. Cerchiamo di semplificarlo scrivendo dei metodi di comodo per automatizzare i compiti più comuni, e per dividerlo in sezioni. Scrivereemo:

- un metodo per tradurre automaticamente un termine invece di scrivere ogni volta dictionary.get(). Dovrà avere accesso alla mappa "dictionary". Verrà usato diverse volte.
- un metodo per creare l'oggetto di classe Census (il "database"), chiedendolo all'utente finchè non ci da un file valido o non si arrende.
- un metodo per riempire la mappa dictionary, caricando i valori da file, come visto in precedenza.
- E già che ci siamo, un metodo per stampare direttamente i termini tradotti.

Ne approfittiamo anche per fornire il codice completo di tutte le classi, con alcune piccole aggiunge e la rimozione di tutti i commenti di dettaglio. L'esercizio migliore per lo studente è commentare le classi una per una, e anche ragionare su eventuali vulnerabilità del codice, o margini di miglioramento.

Il codice seguente fa uso delle List (per contenere le Person), dei Set (per calcolare le professioni distinte e per selezionare solo lingue esistenti), delle Map (per tradurre), dei metodi statici (per semplificare il main), dell'oggetto aggregatore (Census census) e dei principi di architettura.

Sarebbe bello aggiungere dei metodi boolean valid() e scartare gli oggetti non validi in fase di importazione da census.txt. I file forniti di seguito sono non di meno corretti.

File di supporto:

//census.txt
Ferdinando,Primerano,05/02/1980,Teacher,M
Ferdinando,Primerano,10/04/1990,Programmer,M
Betty,Smith,10/10/2000,Admin,F
//italian.txt
WELCOMEMESSAGE:Benvenuti in Census 1.0
ASKCOMMAND:Inserire un comando fra P (sommario delle professioni), G (ricerca per genere), GT (totali per genere), A (ricerca per fascia di età), Q (per uscire) ASKMINAGE:Inserire età minima
ASKMAXAGE:Inserire età massima
ASKGENDER:Inserire genere (M/F)
PEOPLE:Persone totali
MALES:Uomini
FEMALES:Donne
PRESENTPROFESSIONS:Professioni in archivio
QUIT:Addio
//english.txt
WELCOMEMESSAGE:Welcome to Census 1.0
ASKCOMMAND:Insert a command between P(professions summary), G (search by genre), GT (totals by genre), A (search by age range), Q (to quit)
ASKMINAGE:Insert min age
ASKMAXAGE:Insert max age
ASKGENDER:Insert genre (M/F)
PEOPLE:People
MALES:Men
FEMALES:Women
PRESENTPROFESSIONS:Professions in the archive
QUIT:Goodbye

E di seguito il codice:

```
// class Person in package entities
package entities;

import java.time.LocalDate;
// classe con encapsulamento
public class Person
{
    private String name, surname;
    private Date dateofbirth;
    private String profession;
    private String gender;

    public Person(String name, String surname, Date dateofbirth, String profession, String gender)
    {
        this.name = name;
        this.surname = surname;
        this.dateofbirth = dateofbirth;
        this.profession = profession;
        this.gender = gender;
    }

    public String getName()
    {
        return name;
    }
}
```

```

    }
    public void setName(String name)
    {
        this.name = name;
    }
    public String getSurname()
    {
        return surname;
    }
    public void setSurname(String surname)
    {
        this.surname = surname;
    }
    public Date getDateofbirth()
    {
        return dateofbirth;
    }
    public void setDateofbirth(Date dateofbirth)
    {
        this.dateofbirth = dateofbirth;
    }
    public String getProfession()
    {
        return profession;
    }
    public void setProfession(String profession)
    {
        this.profession = profession;
    }
    public String getGender()
    {
        return gender;
    }
    public void setGender(String gender)
    {
        this.gender = gender;
    }

    public String toString()
    {
        return name+" "+surname+" , "+dateofbirth+", "+profession+" "+gender;
    }

    public int getAge()
    {
        // approssimativa. faccio finta di avere già compiuto gli anni
        return LocalDate.now().getYear() - this.getDateofbirth().getYear();
    }
}

// class Date in package entities
package entities;

// classe con encapsulamento.
public class Date
{
    private int year,month,day;

    public Date(int day, int month, int year)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public int getYear()
    {
        return year;
    }

    public void setYear(int year)
    {
        this.year = year;
    }

    public int getMonth()
    {
        return month;
    }

    public void setMonth(int month)
    {
        this.month = month;
    }
}

```

```

}

public int getDay()
{
    return day;
}

public void setDay(int day)
{
    this.day = day;
}

public String toString()
{
    String sday = day> 9 ? day+"": "0"+day;
    String smonth = month> 9 ? month+"": "0"+month;
    return sday+"/"+smonth+"/"+year;
}

// Classe dell'oggetto aggregatore Census in Entities
package entities;

import java.io.File;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Scanner;
import java.util.Set;

public class Census
{
    private List <Person> content = new ArrayList <Person>();

    public Census(String filename) throws Exception
    {
        Scanner reader = new Scanner(new File(filename));
        while(reader.hasNextLine())
        try
        {
            Person p = rowToPerson(reader.nextLine());
            // content si dilata per accogliere il nuovo elemento, p, se siamo riusciti a crearlo
            content.add(p);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        reader.close();
    }

    public List byAgeRange(int minage, int maxage)
    {
        List <Person> res = new ArrayList <Person>();
        for(Person p:content)
            if(p.getAge()> minage && p.getAge() <maxage)
                res.add(p);
        return res;
    }

    public Set <String> getProfessions()
    {
        Set <String> res = new HashSet <String> ();
        for(Person p:content)
            res.add(p.getProfession());
        return res;
    }

    public Person rowToPerson(String row)
    {
        String[] parts = row.split(",");
        String name = parts[0];
        String surname = parts[1];
        String[] date = parts[2].split("/");
        //spezzo la data per ricavarne le parti per ricostruire
        //l'oggetto Date
        int day = Integer.parseInt(date[0]);
        int month = Integer.parseInt(date[1]);
        int year = Integer.parseInt(date[2]);
        String profession = parts[3];
        String gender = parts[4];
        // ho letto una riga dal file, e la ho trasformata in un
        //oggetto di tipo persona
    }
}

```

```

        return new Person(name, surname, new Date(day,month,year),profession, gender);
    }

    public List <Person> byGender(String gender)
    {
        List <Person> res = new ArrayList <Person>();
        for(Person p:content)
            if(p.getGender().equals(gender))
                res.add(p);
        return res;
    }

    // un getter
    public List <Person> getContent()
    {
        return content;
    }
}

// main con metodi statici, nel package main
package main;
import java.util.Scanner;
import java.util.Map;
import java.util.HashMap;
import java.io.File;
import java.util.Set;
import java.util.HashSet;
import entities.*;

public class Main
{
    //dovremo condividere la tastiera fra diversi metodi
    static Scanner keyboard = new Scanner(System.in);
    // Mappa dichiarata come "di classe", condivisa fra i metodi
    static Map <String, String> dictionary = new HashMap <String, String> ();
    //metodo di traduzione
    static String translate(String key)
    {
        //se non trovo la chiave non la traduco, la lascio as is
        return dictionary.containsKey(key) ? dictionary.get(key) : key;
    }
    //metodo di stampa con traduzione incorporata. sostituisce System.out.println
    //in alcuni casi
    //in alcuni casi
    static void print(String key)
    {
        System.out.println(translate(key));
    }
    static void pickLanguage()
    {
        //parto dall'inglese
        String language = "english";
        Set <String> availableLanguages = new HashSet <String> ();
        availableLanguages.add("italian");
        availableLanguages.add("english");
        while(true)
        {
            System.out.println("Available languages: english, italian");
            System.out.println("Type a language");
            language = keyboard.nextLine();
            //ho questa lingua?
            if(availableLanguages.contains(language))
                break;
            else
                System.out.println("Language not recognized");
        }
        //ottenuta una lingua aggiorno la mappa che contiene i termini
        Scanner reader = null;
        try
        {
            reader = new Scanner(new File(language+".txt"));
            while(reader.hasNextLine())
            {
                String[] parts = reader.nextLine().split(":");
                //la parte a sinistra è la chiave, la parte a destra è il valore
                dictionary.put(parts[0], parts[1]);
            }
        }
        catch(Exception e)
        {
            //non trovo il file o c'è un errore nel file. Termino
            System.out.println("Bad language file "+language+".txt. Terminating");
            return;
        }
    }
}

```

```

        }
    finally
    {
        // in tutti i casi, chiudo il file aperto
        reader.close();
    }
}
static Census getCensus()
{
    String filename = "census.txt";
    boolean valid = false;
    // devo dichiararlo fuori dal while, per poterlo usare fuori.
    Census census = null;
    while(!valid)
    {
        try
        {
            census = new Census(filename);
            valid = true;
        }
        catch(Exception e)
        {
            //Il costruttore propaga questa eccezione. Io sono costretto a gestirla
            System.out.println("File "+filename+" not found. Could you specify a different name?");
            filename = keyboard.nextLine();
        }
    }
    return census;
}
public static void main(String[] args)
{
    // Primo step: impostare la lingua
    pickLanguage();
    // Secondo step: creare l'oggetto census a partire dalle indicazioni dell'utente
    Census census = getCensus();
    while(true)
    {
        print("ASKCOMMAND");
        String cmd = keyboard.nextLine();
        switch(cmd)
        {
            case "G":
            {
                print("ASKGENDER");
                String gender = keyboard.nextLine();
                for(Person p:census.byGender(gender))
                    System.out.println(p);
            }
            break;
            //totali per genere
            case "GT":
            {
                System.out.println(translate("PEOPLE")+": "+census.getContent().size());
                System.out.println(translate("MALES")+": "+census.byGender("M").size());
                System.out.println(translate("FEMALES")+": "+census.byGender("F").size());
            }
            break;
            case "A":
            {
                print("MINAGE");
                int minage = Integer.parseInt(keyboard.nextLine());
                print("MAXAGE");
                int maxage = Integer.parseInt(keyboard.nextLine());
                for(Person p:census.byAgeRange(minage,maxage))
                    System.out.println(p);
            }
            break;
            case "P":
            {
                print("PRESENTPROFESSIONS");
                for(String profession:census.getProfessions())
                    System.out.println(profession);
            }
            break;
            case "Q":
            {
                print("QUIT");
                return;
            }
        }
    }
}
}

```


8 - Ereditarietà

8.1 Creare un tipo da un altro tipo: introduzione concettuale all'ereditarietà

Le classi possono essere usate come modello per la creazione di oggetti, ma si ha anche la possibilità di usarle anche come base per la creazione di altre classi, con un meccanismo noto come **ereditarietà**.

Partiamo da un caso concreto. Disponiamo già di una classe Person, ma ora vogliamo andare più in dettaglio: agli insegnanti chiederemo anche che materia insegnano, in che scuola e da quanti anni. Si tratta di altre tre proprietà che non avrebbe senso mettere in Person, perché non avrebbero significato per qualcuno che non fosse un insegnante.

Una prima soluzione sarebbe avere due tipi, Person e Teacher, indipendenti, ma noterete che la differenza fra i due è minima: tre proprietà presenti in Teacher e non in Person. Abbiamo rispettivamente: Person (name, surname, dateofbirth, profession, gender) e Teacher (name, surname, dateofbirth, profession, gender, subject,school,yearsofservice). Potremmo aggiungere che in Teacher il campo profession è superfluo - è per forza "Teacher".

Una idea migliore è *derivare* Teacher da Person, collegandoli e dichiarando Teacher come una *mutazione* o più correttamente *sottoclasse* di Person.

In gergo tecnico, Teacher sarà una sotto-classe di Person, o sotto-tipo, o classe figlia, o classe derivata. Diremo anche che Teacher estende ("extends") Person, perché parte da un tipo più generale (Person) per arrivare a uno più espressivo (Teacher). Teacher potrà riutilizzare potenzialmente tutto ciò che è in Person (metodi e proprietà) e aggiungere del proprio, secondo meccanismi anche abbastanza complessi che ora andremo a vedere. Person a sua volta è super-classe, o super-tipo, o classe-base, o classe madre, di Teacher.

In Java, questo si scrive esclusivamente così:

```
public class Teacher extends Person
{
    private String subject, school;
    private int yearsofservice;
    //Teacher sta ricevendo implicitamente quello che c'era di TRASMISSIBILE in Person, che NON è tutto
    //ma sta ricevendo il toString(), il getAge(), i getter e i setter, e quindi indirettamente
    //accesso a tutte le proprietà di Person. Teacher ha un nome, un cognome, una data di nascita e potrà accedervi
    //in quanto si tratta di una "Person" mutata, anche se in questo caso non avrà accesso diretto
}
```

A seconda dei livelli di encapsulamento di Person e dei suoi costruttori, potrebbe non servire altro per dare a Teacher tutte le proprietà e i metodi di Person, ma noi useremo il Person fornito per ultimo, per cui questa class Teacher non è adatta.

Cominciamo con l'enunciare un vincolo: la classe figlia **deve** richiamare un costruttore della classe padre. La classe Teacher appena creata ha solo il costruttore隐式. La classe Person NON ha il costruttore隐式.

La classe Teacher è costretta a implementare un costruttore che richiami l'unico costruttore di Person, e questo non è sorprendente - un Teacher dopo tutto è un oggetto di tipo Person. Se ho preso di avere nome, cognome, data di nascita, genere e professione per una persona, ne avrò bisogno anche per un Teacher. **Creando un teacher sto creando anche una persona**. La nostra classe Teacher quindi diventa come segue:

```
public class Teacher extends Person
{
    private String subject, school;
    private int yearsofservice;

    public Teacher(String name, String surname, Date dateofbirth, String gender, String subject, String school, int yearsofservice)
    {
        // super : il costruttore di Person, a cui invio i dati che mi sono arrivati
        // chiamare super in una sottoclasse significa usare il costruttore della superclasse per creare
        // un pezzo del proprio stato
        super(name,surname,dateofbirth,"Teacher",gender);
        this.subject = subject;
        this.school = school;
        this.yearsofservice = yearsofservice;
    }
}
```

Come si legge quel "super"? Ricordiamoci che creare un oggetto Teacher significa creare un oggetto Person modificato. Noi disponiamo già di un metodo che crea una Person - il costruttore della class Person. Teacher è figlio di Person, e può in qualunque momento accedere alla classe padre con la parola chiave "super". Chiamando super sto creando la parte del nuovo oggetto che è soltanto una Person. Con le righe successive, imposto le parti che invece sono di esclusiva competenza del Teacher e che non potrebbero essere di Person.

Notate che stiamo barando. Il campo profession di Person viene impostato forzosamente a "Teacher", non arriva dall'esterno. Ha senso nell'ottica di riutilizzo della classe Person per creare Teacher, ma non troppo pulito.

E ora come usiamo questo Teacher?

```
Teacher teacher = new Teacher("Ferdinando", "Primerano", new Date(05,02,1980), "M", "Java", "Tiger Den", 20);
System.out.println(teacher.getAge());
```

```
System.out.println(teacher.subject);
//ERRORE!
```

Esempio breve, ma chiarisce due aspetti. Il primo è che Teacher è **ancora** una Person, e quindi posso ancora usare il metodo getAge(), visto che getAge() è public. Un metodo public può essere usato da chiunque, quindi anche dalle mie classi figlie.
Il secondo aspetto è che valgono per Teacher le regole che valgono per tutte le classi. La proprietà subject è private, e non si può usare fuori dall'oggetto. Questo non ci sorprende, ma ora vediamo che cosa abbiamo *davvero* in Teacher: cosa è stato *ereditato* e cosa è solo suo.

8.2 Regole di trasmissione ereditaria fra classi e override

Come regola generale, posta una classe padre P e una classe figlia F, l'oggetto di classe F *eredita* (possiede in quanto derivato dalla classe padre P) **tutte le proprietà e i metodi public e protected di un oggetto di classe P e le proprietà e i metodi package se F è nello stesso package di P**. Una classe F potrà avere **un solo padre P** (Java viene detto a "ereditarietà singola"), e un numero arbitrario di antenati. La classe figlia F eredita l'ereditabile da tutti gli antenati, e conosce tutti i propri antenati. Le classi padre P **non conoscono le figlie**.

Teacher quindi non eredita *nessuna* proprietà di Person (sono tutte private), ma eredita tutti i getter e i setter (sono pubblici), eredita il metodo `getAge()` (public) e il metodo `toString()` (public, e può essere solo public come vedremo in seguito), oltre al costruttore che ha utilizzato nel proprio costruttore (la chiamata a `super()`).

Se Teacher vuole accedere, ad esempio, al proprio genere, sarà costretto a farlo tramite il getter (`getGender()`). Questo può essere desiderabile o indesiderabile, a seconda dei casi e del progettista.

Manteniamo le cose come stanno, e aggiungiamo un metodo in Teacher:

```
//Siamo sempre in Teacher.java
//voglio calcolare gli anni fino al pensionamento.
//ipotizziamo che nella scuola le donne vadano in pensione a 65 anni, gli uomini a 67
public int getServiceLeft()
{
    return getGender().equals("M") ? 67 - getAge() : 65 - getAge();
}
```

Ora Teacher si è differenziato ulteriormente da Person. Ci sono sette metodi in più (3 getter, 3 setter, e `getServiceLeft()`). Io potrò chiamare `getAge()` su qualunque oggetto Person, ma per chiamare `getServiceLeft()` l'oggetto dovrà necessariamente essere un Teacher.

Per capire lo stato delle cose ad ora è utile guardare questa tabella, da cui ho omesso per brevità getter e setter:

Classe	name	surname	dateofbirth	profession	gender	getAge()	toString()	subject	school	yearsofservice	getServiceLeft()
Person	proprio	proprio	proprio	proprio	proprio	proprio	-	-	-	-	-
Teacher	non visibile*	ereditato	ereditato	proprio	proprio	proprio	proprio				

I campi con un asterisco non sono visibili, ma abbiamo lasciato i getter e setter pubblici, che vengono ereditati, e in questo modo Teacher riesce ad accedere ai campi del padre. C'è tuttavia un problema che non abbiamo evidenziato. Proviamo il seguente codice:

```
Teacher teacher = new Teacher("Ferdinando", "Primerano", new Date(05,02,1980), "M", "Java", "Tiger Den", 20);
System.out.println(teacher);
```

Il risultato sarà il `toString()` di Person, visto che lo abbiamo ereditato. Abbiamo del codice già pronto, ma mancano le informazioni relative a materia, anni di servizio e scuola. Una soluzione "furba" potrebbe essere questa:

```
//sempre in Teacher.java
public String toString()
{
    return super.toString() + ", subject " + subject + ", school " + school + ", " + yearsofservice;
}
```

Di nuovo super. Ci stiamo sempre riferendo alla nostra classe padre, che conteneva il `toString()`. In questo caso, `super.toString()` invoca il metodo della classe padre (ottenendo uno String), equivalente a `Person.toString()`. Il metodo `Teacher.toString()` vi accoda materia, scuola e anni di servizio.

La classe figlia in questo caso sta riutilizzando un metodo della classe padre, ma non lo eredita - lo *sovrascrive*. Sarebbe andato bene anche così:

```
//sempre in Teacher.java
public String toString()
{
    return getName() + " " + getSurname() + " " + getGender() + ", teacher , " + school + " " + yearsofservice + " " + subject;
}
```

Abbiamo scelto di nascondere la data di nascita. Non abbiamo obblighi - né di richiamare il metodo della classe padre, stavolta, né di mostrare tutti i dati nel `toString()`. Anche in questo caso stiamo *sovrascrivendo* un metodo della classe padre, non eritandolo. In questo caso come nel precedente, parliamo di **override**.

Formalmente, parliamo di **override** quando un sotto-tipo sovrascrive il metodo di un super-tipo. Notate che non parliamo solo di classe padre e classe figlia. Potremmo stare facendo override di un metodo del "nonno" (la classe padre di mio padre), o di un qualunque antenato del tipo, ma non solo. Vedremo che fra i nostri super tipi non ci saranno solo classi.

Per quanto riguarda i metodi, quindi, possiamo trovarci in una di tre situazioni:

- ereditato e non modificato (ad esempio, `getAge()`), perché ci va benissimo come lo faceva "nostro padre".
- sovrascritto (`override`), usando o meno quello originale. Il `toString()` è l'esempio principale
- non ereditato (metodo `private` o metodo `package` e P ed F sono in packages diversi)

L'override è una forma di polimorfismo, vale a dire "molteplicità di forme". In questo caso, è polimorfismo di metodo : lo stesso metodo `toString()` ha forme diverse a seconda che sia nella classe padre o nella classe figlio.

C'è una differenza netta con l'overloading: nell'overloading abbiamo metodi con lo stesso nome ma con parametri diversi nello stesso tipo. Con l'overriding abbiamo lo stesso metodo all'interno di due tipi legati da un rapporto di ereditarietà.

Ricapitolando:

- Overload: polimorfismo di metodo interno allo stesso tipo, metodi con lo stesso nome ma con parametri diversi. Ad esempio, `m1()` e `m1(int n)`.
- Override: polimorfismo di metodo. Stesso metodo con gli stessi parametri implementato in maniera differente in un supertipo e in un sottotipo (padre e figlio, nonno e nipote...).

L'override è tipicamente indicato in maniera esplicita tramite una *annotation*. Le annotation sono un sistema per arricchire il codice Java di informazioni ad uso del compilatore e di altre classi, ma le approfondiremo in seguito. L'annotation per indicare che un metodo sta sovrascrivendo un altro è appunto `@Override`, da scrivere sopra il metodo. **Non** è obbligatorio, ma è utile, perché costringe il compilatore a verificare che esista un metodo con quello stesso nome nei super tipi, e dove non fosse presente segnalerà un errore al programmatore.

Per terminare il discorso, c'è un'ultima regola da ricordare: l'override non può mai ridurre la visibilità di un metodo. Al massimo può aumentarla. Questo significa che se, ad esempio, `getAge()` è definito come `public` all'interno di `Person` (come in effetti è), `Teacher` non potrà definirlo `private`, `package` o `protected` (in ordine crescente di visibilità).

Questo serve a garantire che il figlio *offra gli stessi servizi del padre*. Una classe figlia non potrà mai offrire meno servizi del padre, perché violerebbe il *principio di sostituzione di Liskov*.

Questo principio recita, con buona approssimazione, che un sottotipo deve essere sempre sostituibile al suo supertipo. Questo vuol dire che una classe F, figlia di P, deve garantire gli stessi servizi di P (gli stessi metodi, le stesse proprietà, con la stessa visibilità).

Java lo garantisce *in parte* con questa regola sulla visibilità. Un approfondimento sul principio di sostituzione esula da questo testo, ma riporto un link per chi volesse approfondire: <https://stackify.com/solid-design-liskov-substitution-principle/#:~:text=The%20Liskov%20Substitution%20Principle%20in,the%20objects%20of%20your%20superclass>.

8.3 Tipo formale e tipo concreto

Analizziamo il seguente codice:

```
Person p = new Teacher(...);
```

Potrebbe sembrare sbagliato. Person e Teacher sono tipi diversi, ma lo erano anche List ed ArrayList.

In pratica, Person è un tipo più generico di Teacher, e i due tipi sono legati da un rapporto di ereditarietà. Tutti i Teacher sono Person, ma non tutte le Person sono Teacher. Noi diremo che un Teacher è un Person (sempre), ma una Person non è necessariamente un Teacher.

I tipi Person e Teacher sono *compatibili asimmetricamente* : Teacher è compatibile con Person (posso assegnare un oggetto Teacher a una variabile di tipo Person), ma Person non è necessariamente compatibile con Teacher (non potrò, salvo conversioni, assegnare un oggetto di tipo Person a una variabile di tipo Teacher).

In Java:

```
// corretto  
Person p = new Teacher(...);  
// sbagliato  
Teacher t = new Person(...);
```

E in effetti ora bisogna capire che *variabile* e *oggetto* sono due concetti differenti e separati. Person p; dichiara una variabile, vale a dire una "scatola a forma di Person". La scatola p potrà ospitare un qualunque tipo di Person (Person, Teacher, Student...), ma ad ora è *vuota*, vale a dire *null*.

Scrivendo p = new Teacher(), io sto creando un oggetto di tipo Teacher, e lo sto "mettendo" dentro p. Ma l'oggetto non è p: è *il contenuto di p*.

Tornando al nostro primo esempio, Person p = new Teacher(), diremo che il *tipo formale* di p è Person, o che il tipo della variabile p è Person, o che il tipo del contenitore è Person. Diremo invece che il *tipo concreto* di p è Teacher, o che il tipo dell'oggetto in p è Teacher, o che il tipo del contenuto di p è Teacher.

In generale, separiamo la variabile dal suo contenitore, e ricordiamo che il tipo della variabile deve essere *compatibile* con quello dell'oggetto al suo interno. Posso sempre mettere Teacher dentro Person, perché Teacher è una Person: è una sua sottoclasse, un suo sottotipo.

8.4 Polimorfismo di oggetti, instanceof e casting

Partiamo sempre da Person p = new Teacher();, e immaginiamo di avere un'altra classe, Student, figlia di Person. Poniamo tre domande:

- l'oggetto in p è un insegnante?
- l'oggetto in p è una persona ?
- l'oggetto in p è uno studente?

La risposta alla prima è sì. Abbiamo creato un oggetto di tipo Teacher (new Teacher()), quindi è sicuramente un Teacher. La risposta alla seconda domanda è "sì", perchè Teacher è una sottoclasse di Person. Il rapporto di ereditarietà si legge anche "è": se F è sottoclasse di P, F è *anche* di tipo P.

La risposta alla terza domanda è no. L'oggetto è una Person, è un Teacher, ma non è uno Student. In effetti, nessuno potrà essere allo stesso tempo Teacher e Student.

Quindi, cosa rispondiamo alla domanda "a che tipo appartiene l'oggetto in p?".

Diremo che è un Teacher, e di conseguenza anche un Person. Nella pratica, l'oggetto in p appartiene a *due* tipi: Person e Teacher. E potrà essere usato sia come Person che come Teacher (Liskov).

Questo viene detto polimorfismo di oggetto. Un oggetto viene detto polimorfico quando appartiene a più tipi, e lo possiamo vedere in molti modi. In Java tutti gli oggetti sono polimorfici: tutti gli oggetti hanno un antenato in comune, la classe **Object**. E' lì che viene definito il metodo `toString()`, ed è per questo che `toString()` deve restare `public` in tutti i sotto-tipi (in tutti gli oggetti): non possiamo ridurre la visibilità di un metodo di `Object`, essendo tutti gli oggetti discendenti di `Object`.

Quindi in realtà l'oggetto nella variabile p appartiene a *tre* tipi: Object, Person e Teacher.

Questo ci porta a un problema: se mi arriva una variabile Person p, come posso essere sicuro del contenuto?

p potrebbe contenere una Person, così come un Teacher, così come uno Student o qualunque sottoclasse di Person. Java mette a disposizione un operatore per verificarlo, **instanceof**:

```
Person p = new Teacher(...);
System.out.println(p instanceof Person);
//true
System.out.println(p instanceof Teacher);
//true
System.out.println(p instanceof Student);
//false
```

`p instanceof Person` si legge anche "l'oggetto p appartiene alla classe Person", o "p è di tipo Person in senso stretto o lato", o "p è di tipo Person o di un sottotipo di Person". Questa relazione si dice anche "is_a": p is_a Person.

Siccome `instanceof` restituisce un boolean, posso usarlo come condizione, tipicamente negli if. Il codice successivo calcola il numero di insegnanti in una lista:

```
public int countTeachers(List <Person> people)
{
    // la lista people contiene PERSONE. Potrebbe contenere insegnanti, studenti o passanti!
    int res = 0;
    for(Person person:people)
        if(person instanceof Teacher)
            res++;
    return res;
}
```

Ma ora supponiamo di voler stampare gli anni di servizio di un Person:

```
Person p = new Teacher(...);
if(p instanceof Teacher)
{
    // QUESTO FUNZIONA
    System.out.println(p.getName());
    // QUESTO NO
    System.out.println(p.getYearsofservice());
}
```

Perché non funziona?

p contiene un Teacher, lo abbiamo anche controllato, ma Java vede p come una Person. **Java lavora sui tipi formali, sui tipi delle variabili.** p è stato dichiarato come Person. Java sa che sono garantiti solo i metodi e le proprietà di Person, e quindi non ci permetterà di accedere a elementi della sottoclasse (in questo caso, `getYearsofservice()`, il getter per gli anni di servizio). Quindi, come accedere a quei dati, quando Java vede l'oggetto come se fosse suo padre (una Person)?

Devo *forzare il tipo formale*, effettuando una operazione che viene detta di *casting*:

```

Person p = new Teacher(...);
if(p instanceof Teacher)
{
    // CASTING
    Teacher t = (Teacher) p;
    System.out.println(t.getName());
    System.out.println(t.getYearsofservice());
}

```

La prima riga all'interno del blocco if è una operazione di "casting". Letteralmente diremo che la variabile t contiene l'oggetto p ma lo vede come un Teacher. **NON** ne ho creato una copia : c'era un solo oggetto prima, p, e c'è ancora un solo oggetto. Le variabili t e p **contengono lo stesso oggetto** . In realtà, ora sarebbe il caso di spiegare cosa è davvero una variabile: è un *riferimento a un'area di memoria*. t e p si riferiscono alla stessa area di memoria, che contiene un oggetto senza nome, ma lo "vedono" in modo diverso. p vede l'oggetto come una Person, e il compilatore vi permetterà di usarlo solo coi metodi di Person. t invece vede l'oggetto, l'area di memoria, come un Teacher (come in effetti è) e vi permetterà di usarne i metodi.

Ci troveremo spesso a lavorare senza conoscere il tipo concreto (il tipo dell'oggetto) ma conoscendo il tipo della variabile. In alcuni casi dovremo forzare i tipi: in tutti questi casi prima controlleremo con instanceof (è davvero di quel tipo?), poi faremo casting. Lo faremo quando avremo bisogno di accedere a metodi o proprietà delle sottoclassi. Torniamo alla lista delle persone. Supponiamo di voler calcolare l'esperienza media dei docenti:

```

public int averageService(List <Person> people)
{
    // la lista people contiene PERSONE. Potrebbe contenere insegnanti, studenti o passanti!
    int s = 0, c=0;
    for(Person person:people)
        if(person instanceof Teacher)
    {
        Teacher t = (Teacher) person;
        s+=t.getYearsofservice();
        c++;
    }
    return s/c;
}

```

A livello teorico, diremo che il tipo formale definisce cosa una variabile è in grado di fare, mentre il tipo concreto definisce come l'oggetto eseguirà quel compito. Avendo dichiarato person come Person, non potrò invocare metodi della classe Teacher a meno di castare l'oggetto Person. Potrò invocare `toString()`, perché tutti gli oggetti lo hanno, ma ogni oggetto lo eseguirà a modo suo: ogni oggetto avrà il suo `toString()`. Anche se invoco `toString()` su Person il come verrà eseguito dipenderà dal tipo dell'oggetto, vale a dire dal tipo concreto.

Terminiamo fornendo la sintassi formale del casting: posso castare un oggetto o a una classe C scrivendo (C)o. Tipicamente salverò il risultato di questa operazione (che è solo un punto di vista diverso sulla stessa area di memoria) in un'altra variabile, quindi C c1 = (C) o;. c1 è la variabile che contiene l'oggetto o visto come appartenente alla classe C.

Il casting può dare eccezioni. o potrebbe non appartenere al tipo C, nel qual caso il programma andrà in eccezione, ma è facile da prevedere, essendo sufficiente verificare prima con `instanceof` se o appartiene a C.

8.5 La classe Object e la differenza fra oggetti e primitivi

Tutti gli oggetti (String, Scanner, File, Integer, ecc...) hanno un antenato comune nella classe **Object**.

La classe Object definisce i metodi fondamentali condivisi da tutti gli oggetti, fra cui il `toString` (che infatti è e deve restare public), e altri metodi di interesse che vedremo di seguito, assieme ad alcune differenze di base nella gestione di oggetti e primitivi.

Una prima differenza da considerare è l'operazione di copia. Vediamo il seguente esempio:

```
Person p = new Person("F","P",new Date(05,02,1980), "Teacher", "M");
Person q = p;
q.setName("Q");
System.out.println(p);
```

Potremmo credere di avere creato una copia di p, q, e di avere cambiato il nome alla copia. Non è così: **l'unico modo di creare un oggetto è tramite l'operatore new**. In questo caso, come nel casting, ho solo creato un altro nome per lo stesso oggetto. p e q puntano alla stessa area di memoria, si riferiscono alla stessa cosa. Cambiando q, ho cambiato p. E in effetti stampando p vedrò il nome cambiato, perché ho cambiato l'oggetto originale.

Come ottenere una copia di p, quindi? Java offre una soluzione, il metodo `clone()` di Object, che vedremo meglio in seguito quando ci occuperemo delle *interfacce*, ma possiamo lavorare su di una soluzione provvisoria, modificando Person in questo modo:

```
//Person.java
@Override
public Person clone()
{
    return new Person(name, surname, new Date(dateofbirth.getDay(), dateofbirth.getMonth(), dateofbirth.getYear()),profession,gender);
}

//Main.java, metodo main()
Person p = new Person("F","P",new Date(05,02,1980), "Teacher", "M");
Person q = p.clone();
q.setName("Q");
System.out.println(p);
System.out.println(q);
//output:
//F P , 05/02/1980, Teacher M
//Q P , 05/02/1980, Teacher M
```

Creo una nuova Person copiando i miei dati (notate come non copi la data di nascita, ma la ricrei), generando un oggetto nuovo e indipendente, vale a dire una nuova *area di memoria*. Questa soluzione è grossolana e artigianale, e vedremo a breve come superarla, ma per ora chiarisce l'idea. L'operatore = non crea copie di un oggetto, ma crea nomi nuovi per lo stesso oggetto.

Visto che stiamo parlando di identità e differenze, poniamoci una domanda: quand'è che un oggetto può dirsi uguale a un altro? Per i primitivi la risposta è semplice: quando hanno lo stesso valore. In quel caso utilizziamo l'operatore == (confronto):

```
int a = 5;
int b = 6;
System.out.println(a==b);
// false
a = 6;
System.out.println(a==b);
//true
System.out.println(a+" "+b);
//6 6
```

Per gli oggetti il discorso è più complesso. Non esiste una definizione automatica di uguaglianza se non quella di *indirizzo*. Due oggetti sono uguali se occupano lo stesso spazio in memoria, vale a dire se sono letteralmente lo stesso oggetto. In questo caso possiamo usare == anche per gli oggetti:

```
Date d1 = new Date(5, 2, 1980);
Date d2 = d1;
System.out.println(d1==d2);
```

Restituirà true, ma è poco interessante. Non sono solo uguali - sono la stessa cosa con due nomi diversi. Per le date, potrebbe essere più interessante ragionare in un altro modo: due date sono uguali se hanno tutte le loro caratteristiche uguali. Mentre per una Person questo potrebbe non essere vero (omonimia, stessa data di nascita e stesso percorso nella vita), due date con gli stessi giorno / mese / anno sono uguali in tutto e per tutto (al punto che potrebbe non avere senso avere due oggetti diversi per la stessa data).

Java mette a disposizione un metodo per questo confronto, `equals()`, presente in Object e con visibilità public. Lo abbiamo già usato in passato, per le stringhe. `equals()` è definito in Object come **equivalente a ==**. Restituirà true se e solo se d1 E' d2, se puntano alla stessa area di memoria:

```

Date d1 = new Date(5, 2, 1980);
Date d2 = d1;
Date d3 = new Date(5,2,1980);
System.out.println(d1==d2);
// true
System.out.println(d1.equals(d2));
// true
System.out.println(d1.equals(d3));
// false

```

Vorremmo avere tre "true", perchè d3, sebbene sia un oggetto diverso rispetto a d1 e d2 (che sono lo stesso oggetto) è in tutto e per tutto uguale. Siamo costretti a *ridefinire il metodo equals()*, vale a dire fare override.

Il metodo equals() è codificato come boolean equals(Object obj); in Object. Vuol dire che possiamo confrontare un oggetto con qualunque altro oggetto, e questo è un vantaggio e un rischio notevole, come andremo a dimostrare fra un attimo. Cominciamo a definire il metodo equals fra date in maniera intuitiva:

```

//siamo in Date.java
@Override
public boolean equals(Object obj)
{
    //se ci passano un oggetto nullo, decisamente non è uguale a noi.
    if(obj==null)
        return false;

    //se sono la stessa cosa sono anche, per forza, uguali
    if(this==obj)
        return true;

    //ci aspettavamo un oggetto di classe Date, ma a quanto pare ci hanno passato
    //qualcosa di diverso. instanceof è falso, e non prosegua
    if(!(obj instanceof Date))
        return false;

    //ok, ora sono convinto che sia una data, ma io lo vedo ancora come un oggetto
    //devo vederlo come una data, quindi castarlo
    Date other = (Date) obj;
    //e ora restituisco true se tutte le parti sono uguali, false altrimenti
    //essendo le parti primitivi, non è un problema
    return other.getDay() == day && other.getMonth() == month && other.getYear() == year;
}

```

Ora il test di prima restituirà 3 volte true, perchè d1.equals(d3) richiama il metodo equals, e confronta due oggetti che sono diversi (aree di memoria diverse) ma con lo stesso *stato*. Il metodo equals ci dirà "sono uguali almeno a livello logico", ma facciamo presente che d1==d3 è ancora false. Sono ancora oggetti diversi in memoria.

Ora cerchiamo di approfondire il metodo equals. Notiamo che non entra necessariamente una Date, ma un Object, vale a dire *qualsiasi cosa*. Volendo potremmo scrivere System.out.println(d1.equals("05/02/1980"));, come è tentazione di tutti gli studenti: il metodo non darebbe errore, ma restituirebbe false. Volendo, e non è sempre consigliabile, **potremmo insegnare a Java a confrontare una String e un oggetto Date**. Non è considerata necessariamente buona prassi, ma potremmo scriverlo così:

```

@Override
public boolean equals(Object obj)
{
    //se ci passano un oggetto nullo, decisamente non è uguale a noi.
    if(obj==null)
        return false;

    //se siamo la stessa cosa siamo anche uguali
    if(obj==this)
        return true;

    //che sia una data o una stringa!
    if(!(obj instanceof Date || obj instanceof String))
        return false;

    if(obj instanceof Date)
    {
        //ok, ora sono convinto che sia una data, ma io lo vedo ancora come un oggetto
        //devo vederlo come una data, quindi castarlo
        Date other = (Date) obj;
        //e ora restituisco true se tutte le parti sono uguali, false altrimenti
        //essendo le parti primitivi, non è un problema
        return other.getDay() == day && other.getMonth() == month && other.getYear() == year;
    }
    else
    {
        //la stringa potrebbe essere formattata male o non rappresentare una data
        try
        {

```

```

//spero che la stringa sia nel formato dd/mm/yyyy
//in caso contrario darà eccezione e restituirà false
String datestring = (String) obj;
String[] parts = datestring.split("/");
return day == Integer.parseInt(parts[0]) && month == Integer.parseInt(parts[1]) && year == Integer.parseInt(parts[2]);
}
catch(Exception e)
{
    //la stringa ha prodotto errori: restituisco false. Di sicuro non era una data
    return false;
}
}

//main.java
public static void main(String[] args)
{
    Date d1 = new Date(5, 2, 1980);
    System.out.println(d1.equals("05/02/1980"));
    //true
}

```

Ma si tratta di un estremo. Il compilatore vi avviserà che una String è un parametro "unlikely" per un confronto fra Date. In effetti sarebbe meglio evitare, quindi terremo buona la prima versione del metodo.

Per concludere, almeno per adesso, il discorso sui metodi di Object dobbiamo menzionare *public int hashCode()*.

Un hashCode() è un intero che serve a "indicizzare" l'oggetto. L'idea è che lo stato di un oggetto si possa riassumere sotto forma di un numero intero, che verrà poi utilizzato per rintracciare meglio la sua posizione in particolari strutture dati, ma la spiegazione in dettaglio è da rimandare per ora. Il nostro interesse in hashCode è relativo alla struttura dati degli insiemi (Set). Per ragioni che non andremo ad approfondire, abbiamo bisogno di equals e di hashCode per garantire il corretto funzionamento di HashSet per gli oggetti che creeremo. Il Set utilizzerà questi due metodi per capire se dispone già di quell'oggetto o meno, e quindi per evitare duplicati.

Cerchiamo di generare hashCode per fare in modo che tutti gli oggetti Date abbiano un hashCode, un "indirizzo", diverso. hashCode() diventa una "posizione naturale" dell'oggetto nella struttura dati. Per le date, è piuttosto facile generare numeri sempre diversi:

```

@Override
public int hashCode()
{
    return Integer.parseInt(year+"_"+month+"_"+day);
}

```

In questo modo HashSet ha modo di capire se due oggetti sono uguali, e anche modo di capire "dove" metterli, e dove cercarli per vedere se ci sono già. Avendo hashCode() ed equals() in Date, il seguente esempio funziona correttamente:

```

Set <Date> holidays = new HashSet <Date> ();
Date d1 = new Date(25, 12, 2020);
Date d2 = new Date(25, 12, 2020);
Date d3 = new Date(1,1,2021);

holidays.add(d1);
holidays.add(d2);
holidays.add(d3);

System.out.println(holidays);
//[25/12/2020, 1/1/2021]

```

Per quanto d1, d2 e d3 siano oggetti diversi, l'HashSet li riconosce come uguali, e non permetterà a d1 e d2 di coesistere - li riconosce come uguali e producono anche la stessa posizione all'interno dell'HashSet holidays. Sono nello stesso "ripiano" dello "scaffale", sono "uguali", e HashSet ne tiene solo uno.

Ci servirà per evitare i doppiioni, assieme ad altri accorgimenti successivi.

8.6 Caso di studio: una "fabbrica" di Date, il riutilizzo dello stesso oggetto

Abbiamo visto all'esempio precedente che d1 e d2 avevano lo stesso stato, pur essendo oggetti diversi. Questo è uno spreco di memoria:

```
Person a = new Person("F","P", new Date(5,2,1980)); Person b = new Person("A","B", new Date(5,2,1980));
```

Gli oggetti a.dateofbirth e b.dateofbirth sono due oggetti distinti (uso due volte il new), ma contengono la stessa informazione. Il consumo di memoria è doppio senza avere vantaggi apprezzabili. Avrebbe avuto più senso qualcosa di questo tipo:

```
Date d = new Date(5,2,1980);  
Person a = new Person("F","P",d);  
Person b = new Person("A","B",d);
```

Questo ha un grosso vantaggio, nel consumo di memoria, e degli svantaggi non indifferenti in fase pratica che dovremo saper gestire. Il primo è il seguente: creando una Person, come faccio a sapere se da qualche parte non esiste già la Date che mi serve?

Se sono una Person ho bisogno di una data di nascita, e la scelta più naturale è creare una, ma quella data potrebbe essere stata già creata mille altre volte. Come faccio a procurarmi la copia già esistente invece di creare una mia copia (inutile) dell'oggetto? Una soluzione è fare in modo che le Date siano prodotte in maniera controllata.

Per essere sicuri che le date non possano più essere prodotte indipendente, cambiamo la visibilità del loro costruttore a private:

```
//siamo in Date.java  
public class Date  
{  
    private Date(int day, int month, int year)  
    {  
        this.year = year;  
        this.month = month;  
        this.day = day;  
    }  
    //la visibilità di questo costruttore è private  
    //la visibilità della classe è ancora public!
```

Stiamo cominciando a "giocare" con l'incapsulamento. Stiamo dando al mondo la possibilità di vedere la classe (è public) ma non di crearla tramite il costruttore, che è esattamente quello che vogliamo: per creare una data dovremo passare da un metodo che le crea in maniera controllata. Vediamo come:

```
//siamo sempre in Date.java  
  
// questo Set appartiene alla classe, ed è unico. Conterrà tutte le date create in  
// passato durante la vita della classe. E' una soluzione ancora non ottimale ma discreta  
private static Set <Date> created = new HashSet <Date>();  
  
// questo metodo sostituirà il nostro costruttore ma NON è un costruttore! Ha un tipo di ritorno e non  
public static Set <Date> created = new HashSet <Date>();  
  
public static Date make(int day, int month, int year)  
{  
    Date d = new Date(day,month,year);  
    //se c'è, lo cerco fra i creati  
    //e se lo trovo lo restituisco  
    if(created.contains(d))  
        for(Date p:created)  
            if(p.equals(d)) return p;  
    //Se non lo abbiamo trovato  
    //vuol dire che non era ancora stato creato  
    //lo aggiungiamo alla lista dei creati  
    created.add(d);  
    //e lo restituiamo  
    return d;  
}
```

Notiamo che il metodo make() *non* è un costruttore, anche se ha le responsabilità di un costruttore. Non ha il nome della classe (si chiama make(), non Date()), e ha un return (di tipo Date), e rispetto al costruttore ha una grande libertà: può restituire un oggetto già esistente. Il costruttore ne crea sempre uno nuovo.

Cosa succede quando chiamo il metodo make()? Intanto, notiamo che è un metodo static, quindi lo posso richiamare sulla classe (Date.make(g,m,a)). Poi notiamo che è un metodo public - potrà essere richiamato da tutto il mondo. Per finire notiamo che fa uso di una

proprietà static (il Set < Date > created). Per usarlo potremmo scrivere qualcosa di simile:

```
// in qualunque pezzo di codice
Date d1 = Date.make(5, 2, 1980);
Date d2 = Date.make(5, 2, 1980);

System.out.println(d1==d2);
// true. Non sono solo uguali. Sono la stessa cosa
```

Cosa è successo? Al primo richiamo di Date.make, il set non conteneva quella data (5/2/1980). Ha creato un oggetto che aveva nome "d" e lo ha salvato dentro created. Tutto questo è stato fatto nello scope di classe, non di oggetto. Al secondo richiamo, la data esisteva ed ci è stato restituito *lo stesso oggetto creato al primo giro*. La variabile d1 e la variabile d2 puntano allo stesso oggetto.

E in effetti vale per loro non solo l'uguaglianza di stato, ma l'uguaglianza di memoria. Sono la stessa cosa.

Questo ci porta ad avere problemi in seguito. Supponiamo ad esempio di voler cambiare d2. Chiameremo d2.setYear(1990), e cambieremo anche d1, perché sono in realtà la stessa cosa. Correggendo la data di nascita di una persona, cambieremo la data di nascita di tutti quelli nati in quel giorno!

Questo ovviamente non è ammissibile, e un modo per evitarlo è di rendere le parti del tipo Date read only, di modo che chi vuole cambiare una data non possa cambiare solo giorno, mese o anno, ma sia costretto a ricreiarla da zero, *passando di nuovo per il metodo make*.

Lo otteniamo abbastanza facilmente eliminando i setter, e mantenendo le proprietà a private. In questo modo, un oggetto Date una volta creato non potrà essere cambiato nelle sue parti, e bisognerà modificare l'intero oggetto (rimpiazzarlo con un altro Date, magari già esistente).

Riportiamo di seguito il codice di Date modificato per intero, e la creazione di due Person con questo nuovo tipo Date. La data di nascita di una Person viene poi variata senza che la seconda venga modificata. Per comodità aggiungiamo anche un *secondo* metodo make (overloading di un metodo statico) che costruisce una data a partire da una String. Il metodo make(String) per altro si appoggia al metodo make(int,int,int), mostrando come i metodi possano richiamarsi a vicenda.

```
// Date.java, senza setter, con costruttore privato e parti statiche che ne fanno le veci
package entities;

import java.util.HashSet;
import java.util.Set;

public class Date
{
    public static Set <Date> created = new HashSet <Date>();

    public static Date make(int day, int month, int year)
    {
        Date d = new Date(day,month,year);
        //se c'è, lo cerco fra i creati
        //e se lo trovo lo restituisco
        if(created.contains(d))
            for(Date p:created)
                if(p.equals(d)) return p;
        //Se non lo abbiamo trovato
        //vuol dire che non era ancora stato creato
        //lo aggiungiamo alla lista dei creati
        created.add(d);
        //e lo restituiamo
        return d;
    }

    // altro metodo che si occupa della creazione di date. E' sempre static, sempre "make"
    // ma lavora su una stringa del tipo dd/mm/yyyy. La spezza, la trasforma in tre interi e usa il metodo make
    // sopra.
    public static Date make(String date)
    {
        String[] parts = date.split("/");
        //restituisco null in caso di dati assurdi
        if(parts.length!=3) return null;
        return make(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]), Integer.parseInt(parts[2]));
    }

    //proprietà private. impossibile modificare senza i setter
    private int year,month,day;

    //costruttore ad uso privato. potrà essere usato solo dentro la classe
    //quindi solo nel metodo Make
    private Date(int day, int month, int year)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }
}
```

```

public int getYear()
{
    return year;
}

public int getMonth()
{
    return month;
}

public int getDay()
{
    return day;
}

public String toString()
{
    String sday = day> 9 ? day+"": "0"+day;
    String smonth = month> 9 ? month+"": "0"+month;
    return sday+"/"+smonth+"/"+year;
}

@Override
public boolean equals(Object obj)
{
    //se ci passano un oggetto nullo, decisamente non è uguale a noi.
    if(obj==null)
        return false;
    //se siamo la stessa cosa siamo anche uguali
    if(obj==this)
        return true;
    //ci aspettavamo un oggetto di classe Date, ma a quanto pare ci hanno passato
    //qualcosa di diverso. instanceof è falso, e non proseguo
    if(!(obj instanceof Date))
        return false;
    //ok, ora sono convinto che sia una data, ma io lo vedo ancora come un oggetto
    //devo vederlo come una data, quindi castarlo
    Date other = (Date) obj;

    //e ora restituisco true se tutte le parti sono uguali, false altrimenti
    //essendo le parti primitivi, non è un problema
    return other.getDay() == day && other.getMonth() == month && other.getYear() == year;
}

public int hashCode()
{
    return Integer.parseInt(year+" "+month+" "+day);
}

}

// Main.java
public static void main(String[] args)
{
    Person a = new Person("F", "P", Date.make(5, 2, 1980), "Teacher", "M");
    Person b = new Person("C", "D", Date.make(5, 2, 1980), "Programmer", "F");
    // gli oggetti a e b sono legati alla stessa Date
    // Date.make ha prodotto lo stesso oggetto due volte
    // ora voglio cambiare la data di nascita di b, dal 5 al sei febbraio
    // QUESTO NON FUNZIONA! ho tolto i setter in Date.
    //b.getDateofbirth().setDay(6);
    //men che meno, day non è visibile.
    //b.getDateofbirth().day = 6;
    // ma posso scrivere questo:
    b.setDateofbirth(Date.make(6, 2, 1980));
    //in realtà ho cambiato l'intera data, passando sempre per il metodo "make"
    //ora ho associato l'oggetto b a un oggetto date diverso, senza cambiare a
    System.out.println(a);
    System.out.println(b);
}

```

Ci si potrebbe chiedere: come mai static make(), di classe, utilizza un costruttore, vale a dire un metodo che crea l'oggetto? In realtà make() non sta utilizzando un metodo di oggetto. Non usa getYear() o getMonth(). make() *crea* un oggetto. Dentro getYear() o getMonth() io *sono* l'oggetto. make() ha creato un oggetto di tipo Date di nome d, e lo usa come se a usarlo fosse il main(). d è una variabile temporanea del metodo make(). La proprietà che usa è created, che è static, vale a dire dello scope di classe.

8.7 Caso di studio: School e le scelta delle entities

Cominciamo col definire cosa sia una "entity": letteralmente "entity" significa "qualcosa che esiste", ed è la rappresentazione informatica di un oggetto reale. Mentre alcune classi e oggetti sono strumenti prettamente informatici (Scanner) altri servono a rappresentare concetti reali, come abbiamo visto in precedenza. Questi sono le entities.

Un altro modo di vederle potrebbe essere "oggetti di dominio". Siamo nell'ambito specifico del progetto, non nell'ambito generale di un programma. "Scanner" è una classe che ha senso in una varietà di programmi ed è uno strumento generico. Una classe (e di conseguenza un oggetto) come "Teacher" o "BankAccount" non avrebbero un grande senso fuori da un progetto School e Bank rispettivamente.

Ora dovremmo definire di cosa occuparci, quali entities creare. La scelta delle entities dipende da quello che vogliamo ottenere dal sistema. Di cosa ci occuperemo col nostro progetto per gestire una scuola? Orari dei professori, classi, voti degli studenti, stipendi del personale, statistiche di successo? Il discorso è ampio, e un programma serio richiederebbe un libro a sè. Sceglieremo di affrontare il discorso da un punto di vista in massima parte amministrativo: vogliamo gestire le anagrafiche del personale e degli studenti, produrre statistiche relative ai tempi con cui gli studenti si diploma, alla percentuale di abbandoni, ai costi degli insegnanti e del personale non docente e ovviamente rintracciare e classificare i soggetti in base a mansione, materia, anno frequentato ecc...

In particolare, vogliamo garantire i seguenti servizi:

- Possibilità di cercare una persona dato il suo badge id (identificazione)
- Possibilità di cercare il personale per occupazione (insegnanti inclusi)
- Possibilità di filtrare per fascia di età il nostro personale
- Possibilità di calcolare il tempo di percorrenza media dei nostri studenti dall'iscrizione al diploma
- Possibilità di calcolare il tasso di dropout
- Possibilità di calcolare il costo annuo del personale

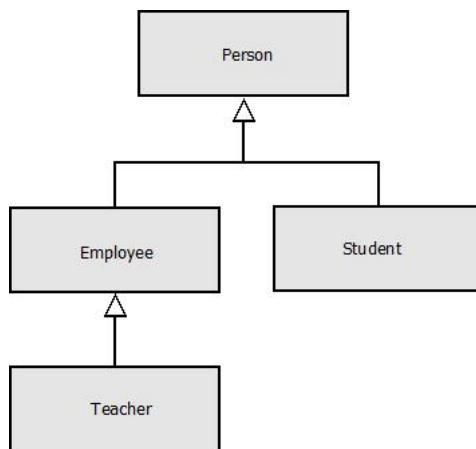
Nello stendere le entità, terremo conto dei servizi che dobbiamo implementare.

Come primo concetto, alla radice di tutto, manterremo quello di Person. Quello di cui ci occuperemo saranno sempre e solo Person e sue sottoclassi. Non ci occuperemo, per ora, di beni della scuola (palazzi, computer, libri...).

Definiamo due sottoclassi di Person: Student, che rappresenterà uno studente, iscritto o diplomato o ritirato che sia, ed Employee, un impiegato della scuola, dagli addetti alla sorveglianza agli operatori igienici agli insegnanti. Gli Employee saranno contraddistinti dal fatto di avere uno stipendio, una anzianità e una tipologia di contratto.

Negli Employee, definiamo un'altra sottoclasse di rilievo, gli insegnanti (Teacher), per cui dobbiamo sapere anche quali materie possono insegnare (potrebbero essere più di una).

Uno diagramma delle entities è riportato di seguito:



Il diagramma sopra non è disegnato a caso. Si tratta di un Class Diagram UML. UML sta per Unified Modeling Language (linguaggio di modellazione unificato) ed è un linguaggio grafico (disegni, quindi) pensato per analizzare, progettare e documentare sistemi informatici complessi. Farà parte del nostro curriculum, ma per ora ci interessano solo due concetti:

- rappresentiamo le classi con quegli eleganti rettangoli (una piccola forzatura rispetto allo standard UML, come vedremo)
- le uniamo tramite frecce che hanno un significato particolare. La linea che ho usato, continua e con freccia a punta chiusa, si legge "extends", e si legge da lato senza freccia a quello con la freccia. Da diagramma io leggo "Employee extends Person", "Student extends Person" e "Teacher extends Employee"

Come abbiamo visto, dire "A extends B" è come dire "A è una sottoclasse di B", o "A è un sotto tipo di B", o "A è un B mutato e specializzato". Da questo diagramma deduciamo che Student ed Employee sono figli di Person, e che Teacher è figlio di Employee (e nipote di Person).

Quella freccia ha anche un altro nome - si legge "is a". Uno Student is_a (è un) Person, un Employee is_a (è un) Person, un Teacher is_a (è un) Employee. La relazione is_a è transitiva: un Teacher is_a un Employee, un Employee è un Person, e di conseguenza un Teacher è un Person. In Java, lo abbiamo visto, si esprime tramite instanceof: Teacher t; t instanceof Employee = true, t instanceof Person = true

Questa struttura padre-figli-nipoti viene anche detta "gerarchia delle entities" o "gerarchia ereditaria", in cui abbiamo la classe più generica, quella che sa fare *meno* cose in cima, e via via ci specializziamo andando verso il basso. Per ora possiamo dire che ci occupiamo solo di Person nel nostro programma.

8.8 Caso di studio: la scrittura delle classi della gerarchia

Abbiamo definito di quali classi avremo bisogno per rappresentare gli oggetti concreti (in questo caso persone) di cui ci vogliamo occupare. Adesso è tempo di decidere quali informazioni memorizzare, e poi arricchire le informazioni coi metodi e coi soliti equals, hashCode e toString(). Si parte sempre necessariamente dalla radice, in questo caso Person. Modifichiamo quella di Census e ottengo quanto segue:

```
package entities;

import java.time.LocalDate;
// classe con encapsulamento
public class Person
{
    //id del cartellino. ogni persona ne avrà uno diverso
    int badgeid;
    String name, surname;
    Date dateofbirth;
    String gender;

    public Person(int badgeid, String name, String surname, Date dateofbirth, String gender)
    {
        this.badgeid = badgeid;
        this.name = name;
        this.surname = surname;
        this.dateofbirth = dateofbirth;
        this.gender = gender;
    }

    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public String getSurname()
    {
        return surname;
    }
    public void setSurname(String surname)
    {
        this.surname = surname;
    }
    public Date getDateofbirth()
    {
        return dateofbirth;
    }
    public void setDateofbirth(Date dateofbirth)
    {
        this.dateofbirth = dateofbirth;
    }

    public String getGender()
    {
        return gender;
    }
    public void setGender(String gender)
    {
        this.gender = gender;
    }

    @Override
    public String toString()
    {
        return badgeid+":"+name+" "+surname+", "+dateofbirth+" "+gender;
    }

    public int getAge()
    {
        // non posso più essere approssimativo. Devo capire se la persona può anche
        // abbandonare la scuola senza accompagnamento
        int year = LocalDate.now().getYear();
        int month = LocalDate.now().getMonthValue();
        int day = LocalDate.now().getDayOfMonth();

        int age = year - this.getDateofbirth().getYear();
        //verifico: se non ha compiuto gli anni gli tolgo un anno
        if(month < 17;
    }
}
```

```

public boolean equals(Object obj)
{
    if(obj==null) return false;

    if(!(obj instanceof Person)) return false;
    Person p = (Person) obj;
    // io e lui siamo uguali se abbiamo lo stesso badge
    return p.getBadgeid() == badgeid;
}
//semplicio: uso il mio badge come hashCode
public int hashCode()
{
    return badgeid;
}
}

```

Notiamo:

- Che le proprietà sono a visibilità package. Le classi Teacher, Employee e Student, sue discendenti, le vedranno fin tanto che stanno nello stesso package della "madre"
- che abbiamo aggiunto una proprietà "badgeid", a tutte le persone. E' il numero di cartellino, e lo porteranno studenti e insegnanti. Dovremo aggiungerlo al file, e servirà a differenziare le persone una dall'altra: siamo "uguali" se abbiamo lo stesso numero di cartellino, o di badge
- Che il metodo getAge() è finalmente "serio", calcolando correttamente l'età

Abbiamo definito la radice della gerarchia. Ora *tutte* le Person sapranno confrontarsi fra di loro ("lui è una mia copia, siamo uguali"), tutte le Person sapranno calcolare la propria età (getAge()), tutte le Person avranno quelle proprietà, e saranno anche visibili a patto di stare nello stesso package.

Ora passiamo all'elemento successivo, l'impiegato, classe Employee. Rispetto a Person, aggiunge la mansione, che abbiamo tolto da Person, lo stipendio mensile e l'anzianità:

```

package entities;

public class Employee extends Person
{
    private static final int SERVICEREQUIRED = 40;
    //proprietà package
    int salary;
    int serviceYears;
    String occupation;
    // come vedete DEVO usare un costruttore della mia classe padre
    // in questo caso ne avevo solo uno e uno uso
    public Employee(int badgeid, String name, String surname, Date dateofbirth, String gender, int salary,
    int serviceYears, String occupation)
    {
        super(badgeid, name, surname, dateofbirth, gender);
        this.salary = salary;
        this.serviceYears = serviceYears;
        this.occupation = occupation;
    }
    //per il resto, aggiungo pochi metodi:
    //costo annuo per la scuola
    public int getYearlyCost()
    {
        //un impiegato ha 13 mensilità
        return salary * 13;
    }
    //anni di servizio residuo
    public int yearsLeft()
    {
        //si va in pensione dopo 40 anni di servizio
        return SERVICEREQUIRED - serviceYears;
    }
    //toString
    //sovrascrivo il metodo della classe padre, Person
    @Override
    public String toString()
    {
        return "Employee nr. "+badgeid+": "+name+" "+surname+", "+getAge()+" years, service:>"+
               "serviceYears+ occupation:"+occupation+", salary:"+salary+ " euro";
    }
    //eredito, senza toccarli, hashCode ed equals. mi stanno benissimo come sono
    //notiamo che Employee ha ereditato implicitamente badgeid, name, surname ecc... essendo nello stesso package del padre
    //ed essendo quelle proprietà package
    //seguono getter e setter
    public int getSalary()
    {
        return salary;
    }
}

```

```

    }
    public void setSalary(int salary)
    {
        this.salary = salary;
    }
    public int getServiceYears()
    {
        return serviceYears;
    }
    public void setServiceYears(int serviceYears)
    {
        this.serviceYears = serviceYears;
    }
    public String getOccupation()
    {
        return occupation;
    }
    public void setOccupation(String occupation)
    {
        this.occupation = occupation;
    }
}

```

Ci restano da scrivere Student e Teacher, in un ordine qualunque. Sceglio di scrivere Teacher, che non sarà figlio di Person, ma di Employee:

```

package entities;
import java.util.HashSet;
import java.util.Set;

public class Teacher extends Employee
{
    private Set <String> subjects = new HashSet <String> ();
    public Teacher(int badgeid, String name, String surname, Date dateofbirth, String gender, int salary,
    int serviceYears, String[] subjects)
    {
        super(badgeid, name, surname, dateofbirth, gender, salary, serviceYears, "Teacher");
        for(String subject:subjects)
            this.subjects.add(subject);
    }
    //può insegnare questa materia?
    public boolean canTeach(String subject)
    {
        return subjects.contains(subject);
    }
    //Sovrascrivo il toString() di employee ma lo richiamo per generare il mio
    @Override
    public String toString()
    {
        return super.toString()+" , subjects:"+subjects;
    }
}

```

Notiamo una cosa: Teacher contiene un Set di String come elenco di materie, ma per comodità il costruttore riceve un vettore di String. Non è un problema: le ultime due righe del costruttore copiano dal vettore al set. Abbiamo già imparato che i parametri del costruttore non devono per forza corrispondere ai tipi delle proprietà.

E' anche importante vedere che non chiediamo occupation quando creiamo un Teacher. La forniamo noi di default - l'occupation di un Teacher è "Teacher".

Adesso terminiamo con Student, che sarà figlio di Person:

```

package entities;
public class Student extends Person
{
    //lunghezza di un corso. andrebbe in un oggetto corso ma...
    private static final int COURSELENGTH = 5;
    // data in cui si è diplomato. Potrebbe essere null
    Date diploma;
    // data di iscrizione. Deve esserci
    Date entry;
    // eventuale data di ritiro. Non può esserci se c'è una data di diploma
    Date retired;
    // anno che frequenta
    int year;
    // primo costruttore: uno studente non diplomato e non ritirato. uno studente attivo!
    public Student(int badgeid, String name, String surname, Date dateofbirth, String gender, Date entry,
    int year)
    {

```

```

super(badgeid, name, surname, dateofbirth, gender);
this.entry = entry;
this.year = year;
}
//uno studente che si è ritirato. Due date
public Student(int badgeid, String name, String surname, Date dateofbirth, String gender, Date entry, Date retired,
int year)
{
    super(badgeid, name, surname, dateofbirth, gender);
    this.retired = retired;
    this.entry = entry;
    this.year = year;
}
//uno studente che si è diplomato. Due date
public Student(int badgeid, String name, String surname, Date dateofbirth, String gender, Date entry,
int year, Date diploma)
{
    super(badgeid, name, surname, dateofbirth, gender);
    this.diploma = diploma;
    this.entry = entry;
    this.year = year;
}
//per comodità, anche uno studente completo, pur sapendo che alcune delle date saranno vuote
//ci tornerà comodo leggendo da file
public Student(int badgeid, String name, String surname, Date dateofbirth, String gender, Date entry, Date diploma,
Date retired, int year)
{
    super(badgeid, name, surname, dateofbirth, gender);
    this.diploma = diploma;
    this.entry = entry;
    this.retired = retired;
    this.year = year;
}
//e ora una serie di metodi non banali:
public boolean isActive()
{
    //Studente attivo: viene a scuola e frequenta
    return diploma==null && retired == null;
}
public boolean isRetired()
{
    //studente che si è ritirato
    return diploma==null && retired!=null;
}
public boolean isGraduate()
{
    //studente diplomato
    return diploma!=null && retired==null;
}
public int yearsLeft()
{
    //anni prima di diplomarsi, se tutto va bene
    return COURSELENGTH - year;
}
//getter e setter
public Date getDiploma()
{
    return diploma;
}
public void setDiploma(Date diploma)
{
    this.diploma = diploma;
}
public Date getEntry()
{
    return entry;
}
public void setEntry(Date entry)
{
    this.entry = entry;
}
public Date getRetired()
{
    return retired;
}
public void setRetired(Date retired)
{
    this.retired = retired;
}
public int getYear()
{
    return year;
}

```

```

    }
    public void setYear(int year)
    {
        this.year = year;
    }
    //toString(), faccio Override di quello di Person, ma lo riutilizzo
    @Override
    public String toString()
    {
        return "Student "+super.toString()+" enrolled on "+entry+", current state:"+ (isActive() ? "ACTIVE" : "NOT ACTIVE");
    }
}

```

Abbiamo un inizio di struttura, e la nostra gerarchia di classi è completa. Verifichiamo di avere capito davvero quali metodi abbiamo, e da chi li prendiamo. Partiamo da Person e seguiamo alcuni metodi e alcune proprietà nella loro evoluzione:

Classe	batchid	getAge()	toString()	getYearlyCost()	canTeach()	isActive()
Person	proprio	proprio	proprio	non presente	non presente	non presente
Employee	ereditato	ereditato	sovrascritto, quindi proprio	proprio	non presente	non presente
Teacher	ereditato	ereditato	sovrascritto quello di Employee, quindi proprio, ma richiama anche il metodo del padre	ereditato	proprio	non presente
Student	ereditato	ereditato	sovrascritto quello di Person, quindi proprio, ma richiama anche il metodo del padre	non presente	non presente	proprio

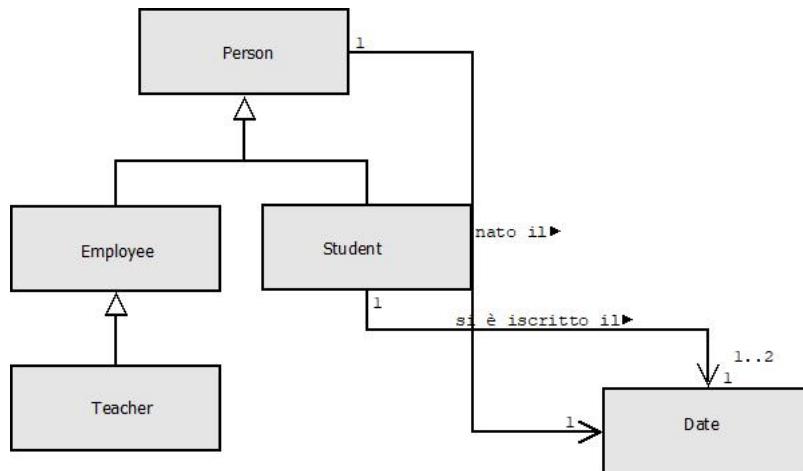
8.9 I rapporti oltre la gerarchia: il rapporto d'uso

Abbiamo visto che la freccia a punta chiusa e linea continua del diagramma UML si legge "extends", o "is_a", e indica il rapporto di padre-figlio. La freccia va dal figlio al padre, perché il figlio sa di chi è figlio, mentre il tipo padre non conosce i suoi figli. A questo punto sorge una domanda: che tipo di rapporto c'è fra Person e Date?

Una Person NON è una Date. Date NON fa parte della gerarchia di Person. Date in realtà non è neanche davvero una entity, dovrebbe essere messa in un package proprio, essendo un oggetto molto più generico e adatto a molti progetti, ma quello che ci interessa è che Date NON è una Person. Il loro rapporto viene detto di *uso*.

Diremo che Person *usa* Date, o che c'è una *relazione* fra un oggetto Person e un oggetto Date, con Person cliente e Date fornitore. Abbiamo diversi tipi di rapporto d'uso, ma questo rapporto d'uso ha il nome proprio di *associazione* e viene indicato con una freccia continua a punta aperta.

Riprendendo in mano il diagramma di prima, avremo quanto segue:



Il diagramma sta diventando dadaista, come accade spesso con UML, ma vediamo di spiegare.

Le frecce della gerarchia sono rimaste uguali. Hanno la punta chiusa e si leggono "is_a". Vanno dal basso verso il l'alto, solo per convenzione grafica. Ci sono poi due frecce nuove, che collegano Person prima e Student poi alla classe Date.

La linea che va da Person a Date ha il nome "è nato il", e c'è un n vicino a Person e un 1 vicino a Date. La freccia è aperta, e in questo caso si legge "associazione". Quella linea nel suo insieme si legge come "associazione nato-il fra n oggetti (n a sinistra, accanto a Person) di tipo Person e un oggetto (1 a destra, sul fianco sinistro di Date) di tipo Date". Per essere più chiari, Person ha bisogno di un oggetto Date: è la data di nascita. A una Person corrisponderà una data di nascita, a un oggetto Date corrisponderanno più Person. Lo abbiamo visto prima: tante persone sono nate lo stesso giorno.

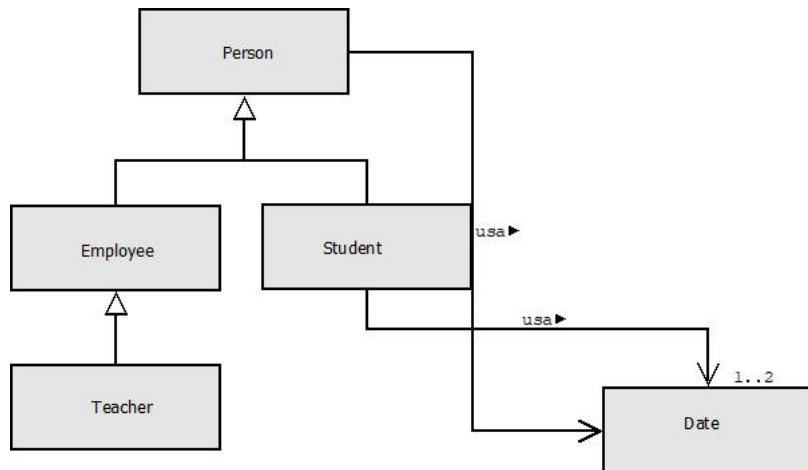
L'associazione "è nato il" è un rapporto di uso, e si può leggere anche come "has_a". Quella freccia si legge come "una Person ha un (has_a) oggetto Date nel campo dateofbirth, che esprime il rapporto nato-il". Il numero 1 (per l'oggetto Date) ed n (per la Person) vengono dette molteplicità o cardinalità dell'associazione.

Allo stesso modo, la freccia "si è iscritto il" che unisce Student a Date è un rapporto di uso, e si legge come "a Student has_a Date", o "un oggetto Student ha un oggetto Date" per la relazione "si è iscritto in quella data". In pratica, Student ha un campo entry di tipo Date, che esprime la sua data di iscrizione alla scuola, e NON è un rapporto di ereditarietà. Student NON è un oggetto Date. Student s instanceof Date è blasfemo. Semmai Student has_a Date è true.

Per la precisione, Student potrebbe averne 3 di Date. Ha una Date ereditata da Person (dateofbirth), ha sicuramente entry e potrebbe avere diploma o retired. Quindi potenzialmente Student ha altri due rapporti di uso con Date, sempre 1-1, sempre con nomi diversi.

Semplificando la teoria e cercando di andare al succo, io dirò che **A è associato a B se un oggetto A contiene nel suo stato oggetti di tipo B**. Person contiene oggetti di tipo Date, quindi Person è associato a Date, o Person has_a Date. Di associazioni poi potrebbero essercene diverse, e con molteplicità diverse, ma per ora vediamo di ricordare il funzionamento, che è poi quello che conta: **A usa B se l'oggetto di classe A usa oggetti di classe B come proprietà**.

E in effetti lo schema sopra volendo può essere anche scritto omettendo alcune molteplicità e i nomi di associazione, semplificando e astraendo, come:



"Una Person usa un Date, uno Student ne usa 1 o 2 (entry e diploma o entry e retired, mai diploma e retired assieme)". Quando si tratta di UML è possibile, anzi consigliabile, omettere pezzi dello schema, non riportare tutte le relazioni e approssimare ove necessario. Per ora dobbiamo capire che *ogni oggetto proprietà di un altro definisce una relazione di uso fra classi*. In questo, una relazione di uso fra Person e Date.

8.10 Caso di studio: l'oggetto aggregatore per School

L'oggetto aggregatore è tipicamente quello che offre i servizi che avevamo definito all'inizio del caso di studio. Non può essere l'oggetto Person a offrirli - una Person conosce se stessa, non tutta la scuola. I totali, i filter, i reduce, i map e i calcoli statistici vanno scritti in un oggetto aggregatore, un oggetto che contenga un insieme di un qualche tipo degli oggetti di cui stiamo parlando: in questo caso un insieme di Person.

Cominciamo con una prima versione della classe dell'oggetto aggregatore, che sarà una entity a sua volta e che in un impeto di fantasia chiameremo School.

```
package entities;
import java.util.List;
import java.util.ArrayList;
public class School
{
    // le mie persone sono memorizzate sotto forma di lista
    List<Person> people = new ArrayList <Person> ();
    // vuol dire che avrò solo oggetti di classe Person? Sì. Ma anche Teacher, Employee e Student sono Person

    //mi arriva un vettore: poco male. lo sposto nella lista
    public School(Person[] people)
    {
        for(Person person:people)
            this.people.add(person);
    }
}
```

Questo è un inizio di classe per l'oggetto aggregatore. Contiene una lista di Person, ma per ora non offre nessun servizio. Provvediamo a integrarli, uno per uno. Sono estremamente semplici:

```
// primo servizio richiesto: ricerca per badge
// il metodo riceverà un intero e restituirà la persona con quel badge, se c'è
// altrimenti, null
// siamo sempre in School.java
public Person searchByBadge(int badge)
{
    for(Person p:people)
        if(p.getBadgeid()==badge)
            return p;
    return null;
}
// il metodo termina in mezzo al for se trova la persona (return termina il metodo, sempre)
// se finisco le persone, esco dal for e restituisco null
// posso SEMPRE restituire null quando lavoro con gli oggetti
```

Fino a qui è stato semplice. Avevamo persone, tutte le persone avevano un badge, abbiamo eseguito uno scorrimento e restituito un solo elemento. Questo algoritmo viene detto *ricerca sequenziale*.

Punto successivo: trovare il personale per occupazione. Questo ci pone di fronte al problema di isolare il personale, visto che gli studenti **non hanno** il campo occupazione. Dobbiamo scorrere la lista, ma tenere solo gli impiegati. Dovremo anche restituire una lista di tutti gli impiegati con quella occupazione.

```
// siamo sempre in School.java. La lista people contiene sia impiegati che studenti
// devo tenere solo gli impiegati, quindi:
// restituisco una lista di IMPIEGATI. Solo loro hanno una occupazione
public List<Employee> employeesByOccupation(String occupation)
{
    List <Employee> res = new ArrayList <Employee> ();
    for(Person p:people)
    {
        if(p instanceof Employee )
        {
            //p è un impiegato. Ora devo vederlo come tale. devo CASTARLO
            Employee e = (Employee) p;
            // e potrebbe anche essere un insegnante. Va bene, gli insegnanti sono impiegati
            // se e ha l'occupazione desiderata lo aggiungo alla lista
            // è un filtro per occupazione. Occupation è l'occupazione desiderata che viene da fuori
            if(e.getOccupation().equals((occupation)))
                res.add(e);
        }
    }
}
```

```
    return res;  
}
```

Abbiamo in realtà filtrato per due criteri: prima per classe, poi per occupazione. Non avrei potuto usare il campo occupation, o il metodo getOccupation(), senza castare, perché p lo vedeva solo come una Person, e non tutte le Person hanno una occupazione.

Gli altri punti sono variazioni sul tema. Bisogna scorrere la lista e castare gli oggetti per poter eseguire i calcoli richiesti. Vengono lasciati come esercizio allo studente, e per ora concentriamo l'attenzione su un problema: i dati vengono caricati tramite un vettore, ma dovremmo caricarli tramite un file.

8.11 Caso di studio: importazione delle entities di School da file

L'importazione da file non è diversa da quanto fatto per Census, ma abbiamo un problema: potremmo dover creare oggetti di quattro classi diverse. Sarà importante avere nei dati del file un campo che ci indichi che tipo di oggetto creare. Presentiamo di seguito il file e una prima versione del costruttore di School che legge le entities (le Person) da file:

people.txt

```
Person,1,F,P,05/02/1980,M  
Employee,2,A,B,01/01/1990,M,1100,10,Secretary  
Student,3,E,F,01/01/2003,M,01/09/2015,,,5  
Teacher,4,C,D,01/01/1984,F,1200,10,Math:Geometry
```

Il primo campo indicherà che classe creare, il secondo è il badge id. Per l'insegnante abbiamo accodato tutte le materie che insegna sul fondo separate dai due punti, per poterle splittare meglio, mentre per lo studente abbiamo sempre specificato tre date aggiuntive, dando vuoto (le virgolette con niente in mezzo) dove non siano disponibili. Il signor E.F. ad esempio è uno studente iscritto nel 2015, nè ritirato nè diplomato, attualmente al quinto anno. Dove mi aspettavo di avere le date ho trovato il vuoto, quindi lascio null.

Adesso scriviamo il costruttore, rispettando i principi che ci siamo dati con Census:

```
//mi arriva il nome di un file  
//sono sempre in School.java  
public School(String filename) throws FileNotFoundException  
{  
    Scanner reader = new Scanner(new File(filename));  
    // anche in questo caso delego la creazione dell'oggetto a un metodo _rowToPerson  
    // l'underscore è una convenzione: indica un metodo privato  
    while(reader.hasNextLine())  
        people.add(_rowToPerson(reader.nextLine()));  
    reader.close();  
}  
//prima versione: immagino di avere un file perfetto  
private Person _rowToPerson(String row)  
{  
    String parts[] = row.split(",");  
    //la prima colonna è il tipo  
    switch(parts[0])  
    {  
        case "Person":  
            return new Person(Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5]);  
            // non serve il break: return termina il metodo  
        case "Employee":  
            return new Employee()  
            {  
                Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]),  
                parts[5], Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8]  
            };  
        case "Student":  
            return new Student()  
            {  
                Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],  
                Date.make(parts[6]), Date.make(parts[7]), Date.make(parts[8]), Integer.parseInt(parts[9])  
            };  
        case "Teacher":  
            return new Teacher()  
            {  
                Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],  
                Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8].split(":")  
            };  
        default:  
            //in un mondo ideale, questo non verrà mai eseguito  
            return null;  
    }  
}  
// aggiungo anche un getter per avere la lista delle Person nella sua interezza  
public List <Person> getAll()  
{  
    return people;  
}
```

Terminato questo lavoro, andiamo a provarlo in un main, dove caricheremo il file e stamperemo il risultato del metodo getAll():

```
School school = new School("people.txt");  
for(Person p:school.getAll())  
    System.out.println(p);
```

Il cui output è:

```
1:F P , 05/02/1980 M Employee nr. 2:A B, 30 years, service:10 occupation:Secretary, salary:1100 euro
Student 3:E F , 01/01/2003 M enrolled on 01/09/2015 , current state:ACTIVE
Employee nr. 4:C D, 36 years, service:10 occupation:Teacher, salary:1200 euro , subjects:[Geometry, Math]
```

Notiamo che ogni oggetto Person ha eseguito il `toString()` a modo suo: sono stati eseguiti i metodi `toString()` rispettivamente delle classi Person, Employee, Student e Teacher. Il Teacher stampa se stesso come "Employee", alla fine, ma è comunque il `toString()` di Teacher. **La lista people contiene, formalmente, Person, ma concretamente contiene una Person, un Employee, uno Student e un Teacher**. Vediamo graficamente cosa è davvero la variabile "people" contenuta nell'oggetto school di classe School:

School school:

List < Person > people:

Person people[0]:

person

Person people[1]:

employee

Person people[2]:

student

Person people[3]:

teacher

Gli elementi della variabile people, che è una List, sono *polimorfici*. Sono *formalmente* delle Person, ma potrebbero essere *concretamente* Person, Employee, Teacher o Student.

La lista people sa che potrà contenere Person, ma non sa in anticipo quali tipi di Person conterrà. Questo è detto *late binding* (scoprire alla fine su che oggetti lavoriamo), ed è detto anche polimorfismo di oggetto.

"Un oggetto Person può avere molte forme". Come abbiamo visto, vale per praticamente qualunque oggetto in Java.

Da un punto di vista grafico, people[0] è un contenitore, posizionato al primo posto della lista ("scaffale") people, che potrà contenere un qualunque tipo di Person. Quindi il suo tipo formale, il tipo della variabile people[0], è Person. Il tipo del suo *contenuto*, il tipo dell'oggetto, potrebbe essere Person o qualunque sua sottoclasse. Il contenitore people[0] potrà contenere qualunque Person, ma non sapremo quale tipo finché non riempiremo il contenitore ("late binding").

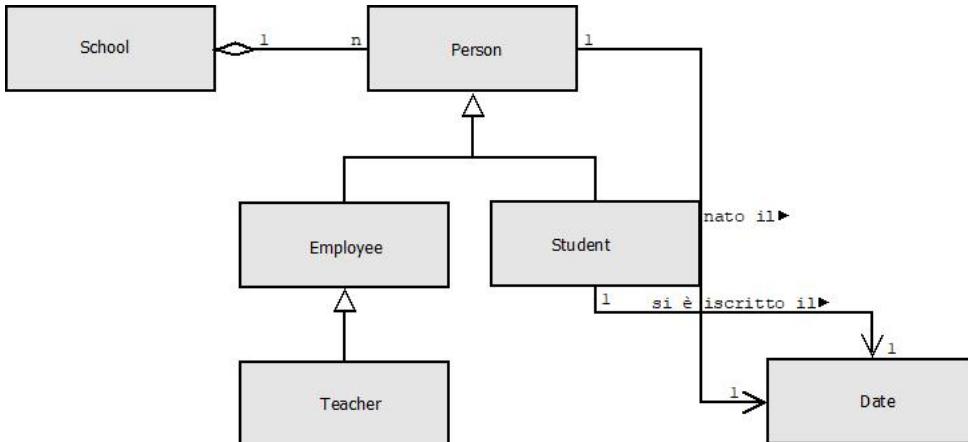
8.12 Conclusioni sul caso di studio School: rapporto di aggregazione ed esercizi

Prima di proseguire, vediamo come è diventato lo "schema del motore" per il progetto School, dopo che abbiamo inserito l'oggetto aggregatore School.

Tecnicamente, diremo che School aggrega n oggetti di tipo Person. "Aggregazione" è una relazione particolare, un tipo particolare di rapporto di uso. Dire che un oggetto school è una persona è blasfemo. Diremo che un oggetto school *has_many* Person - un oggetto school è collegato a e "possiede" molti oggetti di tipo Person.

E' il rapporto che c'è fra una squadra e un suo membro. Il membro esiste indipendentemente dalla squadra, ma la squadra associa i membri in un "contenitore".

Graficamente, abbiamo quanto segue:



Il rombo bianco dal lato di School si legge appunto "aggregazione". "Un oggetto di tipo School aggrega n oggetti di tipo Person". E' di nuovo un rapporto di uso, questa volta uno-a-molti, school *has_many* person.

Ricapitolando, in questo schema abbiamo le seguenti relazioni:

- Employee is_a Person: un impiegato è una persona, impiegato estende persona, impiegato è sottoclasse di persona. Ereditarietà.
- Teacher is_a Employee. Come sopra.
- Student is_a Person. Come sopra
- Person has_a Date. Un oggetto di tipo Person è associato a un oggetto di tipo Date. Associazione generica, che potremmo anche chiamare "nato il". Rapporto di uso. NON ereditarietà. Person NON è un Date
- Student has_a Date, come sopra, ma il nome dell'associazione è diverso. Non dovremmo chiamarla "nato il", ma "iscritto il". E in realtà ce ne sarebbero altre due: diplomato il e ritirato il. Sono tutte relazioni del tipo has_a, e sono tutte *associazioni*
- School has_many Person. E' un rapporto di uso specifico, con un nome predefinito, "aggregazione". NON è ereditarietà, significa che dentro School troverò diversi oggetti di tipo Person

Lo studente attento si chiederà che differenza ci sia fra aggregazione e associazione. La risposta è "poca". In effetti non c'è una vera differenza a livello di codice o di uso, ma si utilizza l'aggregazione (il rombo bianco) a indicare un rapporto di contenimento "liscio", mentre usiamo associazione (e diamo i nomi) per spiegare meglio la relazione fra gli oggetti. Dire che Person has_a Date può aiutarci, ma dire Person nato_il Date chiarisce la natura del legame, così come Student iscritto_il Date, o Student diplomato_il Date.

Sconsiglio, in questo momento, di preoccuparsi troppo di questa parte. La comprensione arriverà col tempo, e bisogna anche dire che progettisti diversi usano sfumature diverse nella realizzazione di questi schemi. Il programmatore impara con la pratica come interpretare questi disegni nell'ambito del team in cui si trova, ove questo team faccia uso di questo particolare formalismo (un class diagram UML) nel suo lavoro. Per adesso prendiamoli come "ausilio grafico alla comprensione" e passiamo avanti.

Per concludere, lo studente è incoraggiato a terminare l'implementazione dei servizi richiesti al punto precedente e a scrivere un main che permetta all'utente di usufruire dei servizi stessi, sul modello di quello di Census. Il meccanismo non è diverso, ma la nostra lista di oggetti ora è polimorfica, quindi in qualche caso potrebbe essere necessario castare per avere accesso ai metodi delle sottoclassi, per vedere gli oggetti come più che Person.

Una volta terminato e testato il main, lo studente dovrebbe ragionare su come implementare una gestione dell'errore, nel file e nei comandi dell'utente, per assicurarsi che il programma non vada in crash. Una volta raggiunta una ragionevole sicurezza che tutto funzioni, si esaminino le seguenti richieste aggiuntive, arrivate per e-mail dal cliente a tre giorni dalla consegna:

- Vogliamo calcolare la media degli studenti divisi per anno
- Ci serve conoscere anche la sezione per gli studenti. Le nostre sezioni vanno da A ad F
- Dobbiamo memorizzare una nuova categoria di impiegati, gli addetti alla sicurezza, per cui dobbiamo memorizzare le date in cui hanno conseguito i certificati di sicurezza A, B e C. I certificati scadono ogni dieci anni, e dobbiamo poter trovare gli addetti alla sicurezza i cui certificati stanno per scadere.

Lo studente valuti quali modifiche apportare alle classi e successivamente le realizzi per permettere l'implementazione dei servizi nella classe School.

9 - Programmazione a oggetti avanzata

9.1 Riepilogo della programmazione a oggetti: ereditarietà, encapsulamento e polimorfismo

La teoria della programmazione a oggetti, come la abbiamo vista fino ad ora, si basa su tre principi:

- **Ereditarietà:** derivare un tipo da un altro. Java è single-inheritance, vale a dire che una classe può avere una sola classe-padre, e potrà richiamarne i metodi (costruttori inclusi) tramite la parola chiave super. La classe figlia *deve* richiamare un costruttore della classe padre, ove questo sia definito. Il tipo figlio, F, *eredita* dal padre proprietà e metodi *ereditabili*. L'incapsulamento definisce ciò che viene ereditato. Tutte le classi antenate A1..An di una classe C vengono dette suoi *supertipi*.

- **Incapsulamento:** da un punto di vista teorico l'incapsulamento è l'applicazione dell'information hiding. Gli oggetti nascondono il proprio funzionamento verso l'esterno ed espongono solo dei punti di accesso definiti secondo regole di visibilità. Questo permette al sistema di ragionare in termini di servizi offerti, e non di come vengono offerti.

Nella pratica, questo si traduce in un sistema di protezione e di visibilità differenziata secondo quattro livelli di accesso: public (visibile a chiunque), protected (visibile dallo stesso package e da tutte le classi figlie, ovunque si trovino), package (visibile dallo stesso package) e private (visibile solo all'interno dello stesso oggetto / classe). Per garantire un accesso controllato a proprietà altrimenti non visibili possiamo offrire dei metodi specifici detti getter e setter, rispettivamente per leggere o per scrivere, che ci permettono di avere proprietà "di sola lettura" così come modifiche controllate che non vadano a manomettere lo stato dell'oggetto.

- **Polimorfismo:** polimorfismo significa letteralmente "pluralità di forme". Distinguiamo tre o quattro forme di polimorfismo a seconda di scelte interpretative.

Il primo è l'overload dei metodi, in cui abbiamo più metodi con lo stesso nome ma parametri diversi nella stessa classe. Abbiamo visto esempi di questo tipo con difference in Time e con static make() in Date.

Una forma di polimorfismo che viene a volte accomunata all'overload dei metodi e a volte riportata separatamente è l'overload dei costruttori, in cui possiamo avere più costruttori per una stessa classe a patto di averli con parametri diversi. Abbiamo visto overload di costruttori in Time e in Student, fra gli altri.

Abbiamo poi l'override, vale a dire la sovrascrittura di un metodo di un super tipo da parte di un sotto tipo (il figlio sovrascrive il metodo del padre, del nonno, del bisnonno...). L'esempio classico è il `toString()`, che abbiamo riscritto in tutte le entities di School.

Notiamo che la visibilità dei metodi sovrascritti non può mai essere ridotta nelle classi figlie, al massimo aumentata. Questo vuol dire che non potrò fare `protected String toString()`, perché `toString()` nasce `public` in Object, l'antenato comune di tutte le classi.

Overload (dei metodi o dei costruttori) e override sono detti polimorfismo di metodo. Esiste poi una forma di polimorfismo di oggetto, nota come late binding. Corrisponde nel non conoscere il tipo concreto di un oggetto fino al momento dell'uso. Lo abbiamo visto con la lista di Person in School - la lista people poteva contenere Person, ma anche Employee, Teacher e Student.

Segnaliamo che una domanda classica ai colloqui è: che differenza c'è fra overload e override? La risposta rapida è: in overload abbiamo metodi con lo stesso nome ma parametri diversi nella stessa classe. Nell'override abbiamo lo stesso metodo con gli stessi parametri in classi imparentate fra loro.

9.2 Classi astratte e loro uso

Analizziamo il caso School. Siamo sicuro di poter accettare oggetti che siano *solo* Person?

Idealmente, chi ha addosso un cartellino (un badge) dovrebbe essere un impiegato (quindi potenzialmente un insegnante) o uno studente. Non dovremmo avere persone "a caso" nella nostra scuola. La classe Person è *utile come base per creare altri tipi, non per creare oggetti*. E' letteralmente troppo generica, per essere utile in sè.

Una soluzione sarebbe vietarsi di creare oggetti Person, creando solo oggetti Employee o Student. In questo modo saremmo sicuri che chi entra dalla nostra porta è impiegato o studente, ma non è un limite rigido. Il resto del sistema o altri programmatore potrebbero violarlo. Java ci mette a disposizione un metodo per assicurarsi che creare un oggetto a partire da una classe sia impossibile: la parola chiave abstract.

Definendo una classe come "abstract" il compilatore ci impedirà di usarla con new, riconoscendola come troppo generica, o comunque non sufficiente da sola per istanziare un oggetto (istanziare significa creare un oggetto di una data classe). Una classe astratta è un *tipo astratto*, e i tipi astratti non sono istanziabili.

In termini di sintassi, è sufficiente scrivere quanto segue:

```
public abstract class Person
{
    //il resto è uguale
}
```

Ora qualunque tentativo di scrivere p = new Person(...) fallirà. Eclipse ci avviserà di un problema di compilazione, e il programma si rifiuterà di partire. Notiamo che Person funziona ancora per tutto il resto: eventuali metodi statici potranno ancora essere richiamati, e, soprattutto, Person farà ancora da base per Employee e Student .

In effetti, l'**uso più comune delle classi astratte è come base per creare nuove classi, non nuovi oggetti**. Una classe astratta è pensata per essere un modello per altri modelli, da estendere a seconda della bisogna. Corrisponde a un concetto troppo generico o vago per essere utile nella nostra modellizzazione del problema.

Nel caso di Person, abbiamo optato per renderla astratta. Non eravamo costretti; avremmo potuto semplicemente evitare di istanziarla (di creare oggetti con new Person()). In generale, che Person sia astratta o concreta è solo una questione di nostro gusto. Esiste tuttavia un caso in cui siamo obbligati a dichiarare una classe come astratta, ed è quando la classe contiene **metodi astratti**.

Per spiegare cosa sia un metodo astratto, valutiamo il seguente problema: non è vero che sono solo gli impiegati a costituire un costo per la scuola. Potenzialmente anche gli studenti attivi sono un costo: mensa, rimborso per i libri, ecc... potenzialmente anche borse di studio. Quindi, *tutte le persone avranno un costo da sostenere per la scuola, ma sarà calcolato in modo diverso in base alla classe della persona* .

Quando dico che tutte le Person avranno un costo, sto dicendo che il metodo int getCost(), riferito all'anno, sarà nella classe Person. E' un comportamento , cioè metodo, comune a tutte le Person, quindi deve restare in cima alla gerarchia. Sarebbe sbagliato metterlo separatamente in Employee e Student.

Tuttavia, per quanto Person debba avere il metodo getCost(), il modo di calcolarlo cambierà da impiegato (in cui sarà il costo dello stipendio) a studente (mensa, borsa di studio, libri, e SOLO per gli studenti attivi). Person sa che dovrà avere quel metodo, ma non ha modo di calcolarlo finché è solo una Person .

La soluzione in questo caso è rendere Person astratta, e sfruttare una tipologia di metodo che è disponibile solo nei tipi astratti: il metodo astratto. **Un metodo astratto è un metodo ridotto alla sua firma (nome ritorno e parametri) dichiarato in un super tipo astratto ma implementato entro il primo sotto tipo concreto** . E' un metodo che dovrà essere implementato dai discendenti del tipo che lo dichiara. I metodi astratti esistono solo nei tipi astratti e non possono essere presenti nei tipi concreti.

Un tipo che presenta dei metodi astratti deve essere dichiarato come tipo astratto. Un tipo senza metodi astratti può essere astratto o concreto.

Person fino ad ora poteva essere indifferentemente astratto (impossibile da istanziare) o concreto (istanziabile), base per oggetti e altre classi o base per sole altre classi. Ma ora cambieremo Person per rispondere alle nostre ultime esigenze:

```
public abstract class Person
{
    // Ecco il metodo astratto: un metodo che
    // dovrà essere offerto dai discendenti di Person.
    // Person lo dichiara. TUTTE le Person lo dovranno avere
    // ma ogni sotto tipo potrà implementarlo in maniera diversa
    public abstract int getCost();
    // il resto è uguale
}
```

Cosa abbiamo detto sopra? Abbiamo detto che tutte le classi che vogliono estendere Person ereditano, assieme alle proprietà e ai metodi già formati, un *obbligo* . Quello di implementare, prima di creare oggetti, il metodo getCost().

Possiamo vedere una classe astratta senza metodi astratti come "troppo generica", o "poco interessante", e quindi inadatta a essere istanziata. Possiamo vedere una classe astratta con metodi astratti come *incompleta* . I suoi discendenti dovranno completarla, prima di poter creare oggetti.

A seguito di quanto ho scritto sopra, ora devo implementare getCost() in Employee e Student. Perchè non necessariamente in Teacher? Employee è un tipo concreto. Posso usarlo per creare oggetti, e di conseguenza deve saper calcolare il proprio costo. Teacher è figlio di un tipo

concreto, quindi non eredita *debiti*. Teacher potrà fare override di getCost(), ma non è richiesto. Employee e Student invece dovranno farlo, in quanto figli concreti di un tipo astratto e incompleto, a cui manca getCost(), pur avendolo dichiarato.
 Procediamo quindi a modificare Student ed Employee. Per Employee, getCost() è solo un alias per il metodo, già esiste, che calcola il costo annuo. Per lo studente, considero come costo annuo un fisso di 1000 euro per libri e mensa (solo pranzo), per i soli studenti attivi.

```
// Student.java
@Override
public int getCost()
{
    return isActive() ? 1000 : 0;
}
// Employee.java, e verrà ereditato da Teacher
public int getCost()
{
    return yearlyCost();
}
```

Adesso tutte le Person che *posso creare (istanziare)* hanno il metodo getCost(). Non ha importanza che Person stesso non lo abbia, perché Person è astratto: *noi non creeremo mai Person*.

In compenso, tutti i Person concreti sono completi. Tutti i tipi Person concreti, non astratti, hanno tutti i metodi di cui hanno bisogno per lavorare. Che una Person sia un Employee, un Teacher o uno Student, avrà un getCost().
 E siccome tutte le Person hanno un getCost() (per quanto ciascuna lo calcoli a modo suo), posso scrivere quanto segue:

```
int sum = 0;
for(Person p:people)
    sum+=p.getCost();
```

Le Person all'interno di people ora *non saranno mai solo Person*. Saranno sempre o Teacher, o Employee o Student. E ciascun oggetto potrà calcolare il proprio costo a modo suo (in realtà, ci sono solo due varianti ad ora), ma *tutti* gli oggetti Person avranno il metodo getCost(), perché tutti gli oggetti Person sono stati creati a partire da tipi concreti, che sono completi per definizione.

Questa è una applicazione, simultanea, di late binding e override, quindi di due forme diverse di polimorfismo. L'oggetto p è polimorfico e il metodo getCost() è polimorfico.

Per terminare il lavoro, devo modificare il metodo _rowToPerson di School:

```
private Person _rowToPerson(String row)
{
    String parts[] = row.split(",");
    //la prima colonna è il tipo
    switch(parts[0])
    {
        //il caso "person" non è più contemplato: Person è astratta. Non posso usarla
        //per creare oggetti
        // case "Person":
        // return new Person(Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5]);
        //non serve il break: return termina il metodo
        case "Employee":
            return new Employee
            (
                Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8]
            );
        case "Student":
            return new Student
            (
                Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                Date.make(parts[6]), Date.make(parts[7]), Date.make(parts[8]), Integer.parseInt(parts[9])
            );
        case "Teacher":
            return new Teacher
            (
                Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8].split(":")
            );
        default:
            //in un mondo ideale, questo non verrà mai eseguito
            return null;
    }
}
```

Notiamo che gli oggetti che creeremo saranno ancora di classe Person. Un Teacher è un Person, uno Student è un Person - ma non saranno mai *solo Person*.

Come regola generale, *se fra i miei antenati ci sono delle classi astratte e io sono concreto, io dovrò implementare o ereditare implementato ogni metodo definito come astratto fra i miei antenati*.

Vediamolo con un esempio. Poniamo A astratto con metodo c1() concreto ed a1() astratto. Poniamo B astratto figlio di A, con metodo astratto a2() e con metodo concreto c2(). B non è obbligato a implementare a1(), perché anche B è astratto: anche B non sarà usato per creare oggetti e può restare "incompleto".

Ora, poniamo C concreto e figlio di B. C erediterà due metodi concreti (c1() da A e c2() da B), e dovrà implementare due metodi astratti (a1() e a2()), a causa dell'obbligo ereditato rispettivamente da nonno e padre.

Erediterà inoltre tutte le eventuali proprietà ereditabili (non per forza tutte) di A e B.

Potevamo avere anche un altro caso. Poniamo A astratto con metodo c1() concreto ed a1() astratto. Poniamo B astratto figlio di A, con metodo astratto a2() e concreto c2(), ma B implementa anche il metodo a1() di suo padre A. C concreto, figlio di B, riceverà concreti a1(), c1() e c2(), e dovrà implementare solo a2(), visto che suo padre B ha estinto l'obbligo verso il nonno A.

C potrà *comunque* sovrascrivere (override) il metodo a1(), ma non ne avrà l'obbligo. Nella sua gerarchia esiste già una versione di a1() implementata, concreta.

Ricapitolando usiamo le classi astratte quando un tipo è "troppo generico" per essere di interesse, e vogliamo che venga esteso (in questo caso è opzionale), e/o quando un tipo è *incompleto* (e in questo caso è obbligatorio).

Sappiamo che il tipo *dovrà* avere un metodo (Person.getCost()), ma non possiamo calcolarlo nella classe padre: dovrà essere presente in tutte le classi figlie *concrete*, dove avremo le informazioni per calcolarlo correttamente. Le classi astratte sono uno dei due tipi astratti, vale a dire dei tipi di cui non possiamo istanziare direttamente oggetti. Il secondo tipo sono le interfacce, che vedremo in seguito.

Una classe astratta serve come base per creare classi figlie (nella maggior parte dei casi), quindi come base per l'ereditarietà, e può contenere tutto quello che può esserci in una classe concreta (proprietà, metodi, proprietà di classe, metodi di classe) con l'aggiunta dei metodi astratti, il cui obbligo di implementazione ricade al massimo sui discendenti concreti.

Le classi astratte sono molto usate nella pratica, come anche le interfacce, perché ci troviamo spesso nelle condizioni di sapere *cosa* vorremo fare con un dato tipo ma di non sapere il *come farlo*. Il cosa è *definito nel supertipo*. Il come è *definito nei sottotipi*.

9.3 Interfacce: il tipo come contratto

Abbiamo definito un programma come un "motore". Gli oggetti sono pezzi che interagiscono fra di loro fornendosi a vicenda dei servizi: il singolo Person offre memorizzazione e calcolo per una Person, mentre School o Census offrono la gestione di un gruppo di Person, e main() si occupa di interagire con l'utente.

Abbiamo visto questo principio in passato, ed è la separation of concerns, o separazione dei compiti. Ogni pezzo ha una responsabilità e offre i propri servizi. Person ad esempio offre il calcolo dell'età su di una Person.

In certi casi, vogliamo formalizzare i servizi offerti creando quello che è a tutti gli effetti un "contratto", un insieme di metodi che vogliamo siano forniti da un dato oggetto di una data classe. In questo caso parliamo di *interfaccia*. Siamo di nuovo nella condizione di sapere cosa vogliamo, ma non sappiamo o non vogliamo sapere come verrà realizzato.

Partiamo da un esempio, e riflettiamo su School e Census. In entrambi i casi abbiamo caricato le Person da file: il servizio di caricamento ad ora è stato scritto dentro i rispettivi oggetti (dentro le classi in realtà), ma è *sbagliato*. Potremmo voler caricare Person da file in altri mille progetti, e potremmo avere bisogno di usare files di volta in volta diversi. Di volta in volta, i dettagli dell'operazione cambieranno, ma non cambierà il *cosa*.

Il caricamento di Person da file è una operazione che ripeteremo spesso, e che posso definire in maniera precisa in termini di "cosa voglio ottenere". Diremo che posso definire un contratto, una *interfaccia*, in questo modo:

```
// PersonFileReader.java
public interface PersonFileReader
{
    List <Person> readFrom(String filename) throws Exception;
}
```

Questo codice è ingannevolmente semplice. Ci sono molti elementi impliciti da notare:

- Questa non è una classe. Questa è una *interfaccia*, vale a dire un *contratto* per altre classi.
- Una interfaccia è un *tipo astratto* e potrà essere usato come tipo formale di una variabile. Vedremo fra poco come.
- Il metodo readFrom(String) è implicitamente public, ed è implicitamente abstract (come i metodi visti con le classi astratte). NON SERVE scrivere abstract. Se è in un tipo interface, è astratto, cioè dichiarato ma non implementato
- Per le interfacce come per le classi, il nome del file deve essere uguale al nome del tipo. Il tipo si chiama PersonFileReader, il file è PersonFileReader.java
- L'interfaccia sarà il contratto, l'obbligo, per una o più classi che *dovranno* avere il metodo readFrom(string) throws Exception. Dovranno, cioè, onorare il contratto

Notiamo che non abbiamo implementato niente. Non è l'interfaccia a fornire il servizio - *l'interfaccia definisce il contratto*. Saranno una o più *classi* a implementare il servizio nell'oggetto.

Formalmente, se una classe C implementa (onora) una interfaccia I, C deve implementare i metodi astratti di I. Il rapporto fra interfaccia e classe è il rapporto fra contratto e fornitore, o se preferite, fra *contratto* e *componente*.

Una classe implementa un interfaccia *se implementa i metodi astratti contenuti nell'interfaccia*. Un componente onora un contratto se implementa i servizi richiesti dal contratto.

Definito il contratto, definisco la classe (il componente) che lo onori. Supponendo di lavorare sugli stessi identici file di prima, potrei scrivere la seguente classe:

```
// la classe BasicCSVSchoolPersonFileReader implementa l'interfaccia PersonFileReader
// vale a dire, implementa i suoi metodi
// vale a dire, onora il suo contratto
// "implements PersonFileReader"
// è BasicCSVSchoolPersonFileReader perchè i file da aprire sono CSV: Comma Separated Values, valori separati da virgole
public class BasicCSVSchoolPersonFileReader implements PersonFileReader
{
    public List<Person> readFrom (String filename) throws Exception
    {
        List <Person> res = new ArrayList <Person> ();
        Scanner reader = new Scanner(new File(filename));
        while(reader.hasNextLine())
            res.add(_rowToPerson(reader.nextLine()));
        reader.close();
        return res;
    }

    private Person _rowToPerson(String row)
    {
        String parts[] = row.split(",");
        switch(parts[0])
        {
            case "Employee":
                return new Employee
                (
                    Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],

```

```
        Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8])
    );
case "Student":
    return new Student
    (
        Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
        Date.make(parts[6]), Date.make(parts[7]), Date.make(parts[8]), Integer.parseInt(parts[9])
    );
case "Teacher":
    return new Teacher
    (
        Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
        Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8].split(":")
    );
default:
    return null;
}
}
```

Ora disponiamo di una classe che genera un oggetto contenente il metodo che ci interessava, vale a dire `readFrom(String)`: disponiamo di un *componente* che onora (implementa) quel *contratto*. Adesso possiamo impiegarlo dentro `School`, *separando* il lavoro di `School` da quello di importazione del file:

```
public School(String filename) throws Exception
{
    PersonFileReader reader = new BasicCSVSchoolPersonFileReader();
    //leggo direttamente Person ora, non righe
    people = reader.readFrom(filename);
}
```

Notiamo che la variabile locale reader è di *tipo formale* PersonFileReader, e di tipo concreto BasicCSVSchoolPersonFileReader. Vi ricorderà qualcosa: PersonFileReader è un *super-tipo* di BasicCSVSchoolPersonFileReader, ma *non* è la sua classe padre. BasicCSVSchoolPersonFileReader è una delle tante forme possibili di PersonFileReader, è uno dei suoi casi particolari, è uno dei tanti *componenti* che onorano il contratto che è PersonFileReader. Una delle possibilità, ma possono averne tante, fra cui ad esempio:

```

// "safe" perché non restituirà nessuna Person null e non andrà in crash
// il componente precedente sarebbe andato in crash incontrando un file mal formato
// e avrebbe restituito null dove non avesse riconosciuto il tipo della person
// questo gestisce meglio gli errori
public class SafeCSVSchoolPersonFileReader implements PersonFileReader
{
    public List<Person> readFrom (String filename) throws Exception
    {
        Scanner reader = new Scanner(new File(filename));
        List <Person> res = new ArrayList();
        while(reader.hasNextLine())
        {
            try
            {
                // questa riga può dare errore
                Person p = _rowToPerson(reader.nextLine());
                //se non ha dato errore e ha prodotto una Person
                //lo aggiungo alla lista
                if(p!=null) res.add(p);
            }
            catch(Exception e)
            {
                //salto la persona.
            }
        }
        reader.close();
        return res;
    }

    private Person _rowToPerson(String row)
    {
        String parts[] = row.split(" ");
        switch(parts[0])
        {
            case "Employee":
                return new Employee
                (
                    Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                    Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8]
                );
            case "Student":

```

```
        return new Student
    (
        Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
        Date.make(parts[6]), Date.make(parts[7]), Date.make(parts[8]), Integer.parseInt(parts[9])
    );
case "Teacher":
    return new Teacher
    (
        Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
        Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8].split(":")
    );
default:
    return null;
}
}
```

Ora abbiamo *due* forme diverse per PersonFileReader, una Basic e una Safe. La versione Basic funzionerà solo su un file perfetto, altrimenti andrà in crash. Il secondo maschererà tutti gli errori e non farà mai andare in crash il programma, fin tanto che c'è un file. Ci si potrebbe chiedere il senso di averne due quando il secondo "funziona meglio", ma non è strettamente vero. In molti casi noi *vogliamo* sapere che un file contiene errori. Il primo componente, Basic, offre un approccio *strict*, vale a dire, tutto deve essere perfetto o generiamo errori (Exception) che arrivano fino al main(). Il secondo componente offre un approccio *loose*, vale a dire permissivo. Un file potrà avere righe di commento, righe vuote, righe con errori, e il programma continuerà a funzionare.

Supponiamo di voler usare School in ambo i modi, strict o loose. C'è una soluzione molto elegante che usa le interfacce:

```
public School(String filename, String importRules) throws Exception
{
    // late binding! Posso creare un componente o l'altro indifferente
    // da un punto di vista formale sono entrambi PersonFileReader
    // in termini di tipo concreto, sarà o Basic o Safe
    // ma NON ci importa, perchè entrambi hanno il metodo readFrom
    // e non possono fare a meno di averlo, perchè onorano l'interfaccia. DEVONO avere quel metodo
    PersonFileReader reader = importRules.equals("strict") ?
        new BasicCSVSchoolPersonFileReader() :
        new SafeCSVSchoolPersonFileReader() ;
    people = reader.readFrom(filename);
}
```

Diremo che `BasicCSVSchoolPersonFileReader` *is_a* `PersonFileReader`, e che `SafeCSVSchoolPersonFileReader` *is_a* `PersonFileReader`, e che `PersonFileReader` è un super-tipo di `BasicCSVSchoolPersonFileReader` e di `SafeCSVSchoolPersonFileReader`, ma *non* loro padre. Diremo che `BasicCSVSchoolPersonFileReader` e `SafeCSVSchoolPersonFileReader` **implementano** `PersonFileReader`.

E qualcosa di simile all'ereditarietà, ma viene detta implementazione. *Non* è ereditarietà diretta. *Non* posso ereditare proprietà di oggetto dall'interfaccia, *non* è mio padre, ma in effetti l'interfaccia aggiunge un tipo all'oggetto generato dalla classe:

```
PersonFileReader reader = new BasicCSVSchoolPersonFileReader();
System.out.println(reader instanceof PersonFileReader);
//true
System.out.println(reader instanceof BasicCSVSchoolPersonFileReader);
//true
```

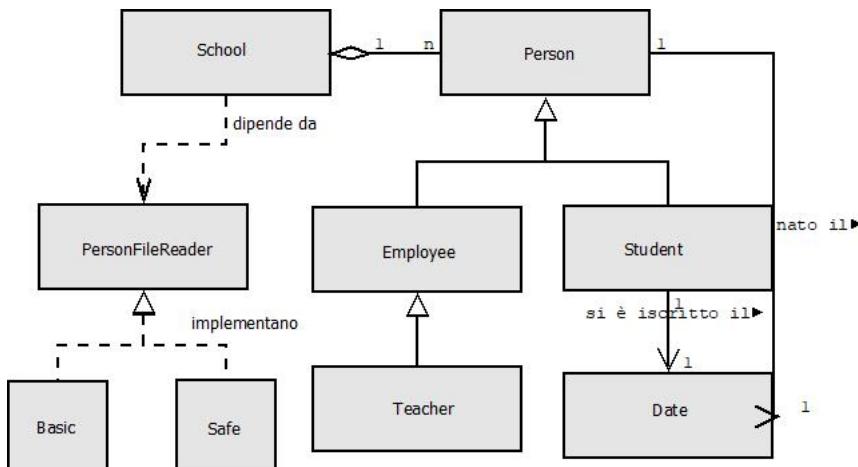
Per chi usa l'interfaccia non c'è distinzione fra Basic e Safe. Il cliente (in questo caso l'oggetto di classe School) ha bisogno di un oggetto che legga per lui Person da file. Ha bisogno di un PersonFileReader, ed entrambe queste classi producono PersonFileReader. Il *cosa* è chiaro all'oggetto School, il *come* non lo riguarda.

Stiamo usando le interfacce da diverso tempo senza averlo ancora realizzato: List, Set e Map sono interfacce. Definiscono il funzionamento rispettivo di liste, insiemi e mappe, lasciando a delle classi la loro implementazione (ArrayList, HashSet e HashMap rispettivamente). Queste classi forniscono i metodi richiesti dai rispettivi contratti, ma non offrono garanzie oltre a quelle definite nei contratti. Map, ad esempio, non offre garanzie sull'ordine di memorizzazione degli elementi, ed HashMap non ne aggiunge. Se vogliamo mantenere l'ordine degli elementi (l'ordine di inserimento di solito), in fase di scorrimento, useremo un tipo di Map diverso: LinkedHashMap.

```
Map <String, String> map = new HashMap <String, String>();  
Map <String, String> orderedmap = new LinkedHashMap <String, String>();
```

Sono entrambe mappe. Entrambe implementano `put()`, `get()`, `keySet()`, ma lo fanno in maniera diversa, e potranno essere usate a seconda della necessità, sempre nel rispetto al contratto Map.

Tornando all'esempio School, vediamo come è diventato il nostro class diagram UML:



Ci sono frecce nuove con significati specifici. Ne approfittiamo anche per ripassare le altre:

- Linea continua, freccia chiusa, come quella che va da Teacher ad Employee. Si legge "extends". Teacher extends Employee, Teacher is_a Employee.
- Linea continua, freccia aperta. Da Person a Date, o da Student a Date. Viene detta "associazione", ed è un rapporto d'uso. Spesso ha un nome specifico: Person nato_il Date, o se preferite Person has_a Date. E' anche detta "composizione sensu latu", composizione in senso lato. Person è composto ANCHE da un Date
- Rombo bianco e linea continua: aggregazione. L'oggetto dal lato del rombo "aggrega", contiene, gli oggetti della classe dal lato della linea. School aggrega n oggetti di tipo Person. Il rapporto è di uso, has_many. E' il rapporto fra una squadra e un membro della squadra, e anche questa è composizione sensu latu. C'è una scarsa differenza semantica fra aggregazione e associazione. In ambo i casi, se A aggrega B, alcuni oggetti di tipo B fanno parte dello stato di A.
- Rombo nero e linea continua: composizione sensu strictu. L'oggetto A possiede e usa in maniera esclusiva uno o più oggetti di tipo B. Se A muore, anche i B collegati vengono cancellati. Non presente in questo schema, lo vedremo in seguito. E' composizione sensu strictu, o "aggregazione forte".
- Linea tratteggiata, freccia chiusa: implementazione. E' il rapporto fra Basic e PersonFileReader, o fra Safe e PersonFileReader, e si legge implements. Basic (BasicCSVSchoolPersonFileReader) implements PersonFileReader, Basic is_a PersonFileReader, Basic instanceof PersonFileReader. Lo stesso vale per SafeCSVSchoolPersonFileReader. Somiglia in molte cose all'ereditarietà, ma è tecnicamente considerata "implementazione". In pratica, è identità di tipo. Anche in questo caso possiamo parlare di una gerarchia, ma non di ereditarietà sensu strictu, ma di implementazione.
- Linea tratteggiata, freccia aperta: dipendenza. School dipende da PersonFileReader. NON è associato a PersonFileReader, perché l'associazione prevede che se A è associato a B, B faccia parte dello stato di A. PersonFileReader non fa parte dello stato di School: viene usato come variabile locale. In questo caso il termine corretto è "dipendenza".

9.4 Implementazione di interfacce multiple e aggiunta di funzionalità tramite interfacce

Java è un linguaggio single-inheritance, multiple implementations. Questo è quanto recita la teoria e la definizione formale. In pratica vedremo che i contorni si fanno sfumati, ma è vero quanto segue: **una classe può avere un solo padre, ma implementare quante interfacce vuole**. Formalmente, una classe C può essere figlia di una sola classe P e implementare I1..In.

Ne vediamo una applicazione andando a modificare Person. Desideriamo che le persone del nostro archivio si possano ordinare, ma dobbiamo scegliere *come*. Possiamo sempre impostare un qualunque ordine su un insieme, e nel caso delle persone gli ordini naturali sono diversi: per età, per mansione, per anzianità lavorativa, alfabetico per nome o per cognome... scegliamo di impostare un ordine basato sul badge. Java offre una interfaccia predefinita per l'ordinamento degli oggetti, ed è Comparable. Effettuiamo questa modifica su Person:

```
public abstract class Person implements Comparable<Person>
{
    // il resto rimane uguale
    @Override
    public int compareTo(Person other)
    {
        return badgeid - other.getBadgeid();
    }
}
```

Comparable è una interfaccia generica (vedremo a breve), che richiede un tipo (Person) per funzionare a dovere, e richiede un metodo di nome `compareTo(Person other)`, che scelgo di implementare in Person, per fare in modo che venga ereditato da tutte le altre Person (Employee, Teacher, Student). Per definizione, A `compareTo` B deve restituire 0 se i due oggetti sono "uguali" per ordine, un positivo se A > B e un negativo se A < B. In questo modo mi è sufficiente restituire la differenza fra i badge. Se il mio badge è maggiore del suo, io vengo dopo di lui.

Avendo aggiunto Comparable, e avendo implementato `compareTo`, un oggetto di tipo Person è ora in grado di dire se è "maggiore" o "minore" di un altro oggetto dello stesso tipo. Di per sé questo potrebbe sembrare poco utile, ma Comparable fa parte delle interfacce standard di Java, ed è integrata nella libreria predefinita del sistema. Una volta implementato, posso usarlo con un altro strumento, Collections.sort(), che userà Comparable per capire come ordinare i dati.

Supponiamo di avere i seguenti dati:

```
Employee,6,A,B,01/01/1990,M,1100,10,Secretary
Student,4,E,F,01/01/2003,M,01/09/2015,,,5
Teacher,3,C,D,01/01/1984,F,1200,10,Math:Geometry
```

L'ordine in cui li troveremmo normalmente nella lista è l'ordine di inserimento, quindi prima il 6, poi il 4, poi il 3. Ciò non è desiderabile, e mi è sufficiente modificare così School:

```
public School(String filename, String importRules) throws Exception
{
    PersonFileReader reader = importRules.equals("strict") ?
        new BasicCSVSchoolPersonFileReader() :
        new SafeCSVSchoolPersonFileReader() ;
    people = reader.readFrom(filename);
    // userà per ordinare gli oggetti l'ordine di comparable
    // sort è un metodo statico della classe Collections
    Collections.sort(people);
}
//main()
School school = new School("people.txt");
for(Person p:school.getAll())
    System.out.println(p);
//output
//Employee nr. 3:C D, 37 years, service:10 occupation:Teacher, salary:1200 euro , subjects:[Geometry, Math]
//Student 4:E F , 01/01/2003 M enrolled on 01/09/2015 , age: 18 current state:ACTIVE
//Employee nr. 6:A B, 31 years, service:10 occupation:Secretary, salary:1100 euro
```

Il metodo `compareTo()` definisce quello che possiamo chiamare "ordinamento naturale". C'è modo di utilizzare ordinamenti diversi, ma lo vedremo in seguito. Per ora, notiamo che implementare un'interfaccia (Comparable) ha permesso a Person di essere ordinato automaticamente da un altro componente. Questo dare-e-avere è comune nella pratica. Molto spesso ci troveremo a implementare interfacce per avere accesso a servizi già implementati. E' il meccanismo dei *framework*, di cui parleremo bene in seguito.

Ora aggiungiamo un altro tassello. Vogliamo che un oggetto di tipo Person possa essere clonato, come abbiamo già fatto in passato, scrivendo un metodo `clone()` apposito. In realtà Java offre già un metodo `clone()`, all'interno di Object, ma è "disabilitato" di default. In generale, non vogliamo che un oggetto possa essere clonato se il programmatore non lo dichiara esplicitamente.

Per rendere possibile clonare un oggetto devo seguire due step. Il primo è creare un metodo `clone()` del tipo dell'oggetto, che richiami il metodo `super.clone()` di Object. Notiamo che `super.clone()` è *protected*, quindi posso accedere dentro le classi figlie, ma non da fuori. Notiamo anche che

`super.clone()` restituisce una copia vista come `Object`, vale a dire come oggetto generico. Dovremo castarlo al tipo che ci interessa. Da ultimo, l'operazione di cloning potrebbe non essere concessa, essere disabilitata, e dare un'eccezione (`CloneNotSupportedException`):

```
// in Person.java
public Person clone()
{
    try
    {
        return (Person) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        e.printStackTrace();
        return null;
    }
}
```

Potremmo aspettarci che questo funzioni, ma in effetti non abbiamo ancora "abilitato" l'operazione di clone:

```
School school = new School("people.txt");
// prendo la prima persona
Person p2 = school.getAll().get(0);
Person p3 = p2.clone();
//L'ultima riga da eccezione. Non abbiamo dichiarato che Person è clonabile.
//l'output è il seguente:
//java.lang.CloneNotSupportedException: entities.Teacher
//at java.base/java.lang.Object.clone(Native Method)
//at entities.Person.clone(Person.java:120)
//at entities.Test.main(Test.java:11)
```

Tutto funziona se aggiungiamo l'interfaccia `Cloneable` a `Person`. In questo caso `Cloneable` non ci obbliga a implementare davvero `clone()`, ma serve al metodo `clone` per essere certo che il tipo è pensato per essere clonabile. Questa interfaccia funziona da **marker**, o segnalatore, per aggiungere un tipo a un oggetto e/o comunicare al resto del sistema che un determinato servizio è disponibile, a volte senza neanche richiederne l'implementazione.

Person quindi diventa:

```
public abstract class Person implements Comparable<Person>, Cloneable{
```

La classe `Person` implementa due interfacce (`Comparable < Person >` e `Cloneable`) ed è figlia di `Object`. Utilizza il metodo `clone()` di `Object` per realizzare il proprio metodo `clone`. In questo modo, potremo scrivere qualcosa di questo tipo:

```
School school = new School("people.txt");
// prendo la prima persona
Person p2 = school.getAll().get(0);
Person p3 = p2.clone();
p3.setName("X");
System.out.println(p2);
System.out.println(p3);
```

Ricapitolando, abbiamo usato due interfacce (`Comparable` e `Cloneable`) per aggiungere due funzioni (comparazione / ordinamento e clonazione) a un tipo già esistente. Queste funzioni sono generiche e si possono applicare a oggetti di molti tipi diversi, in gerarchie ereditarie diverse. Potremmo scegliere, ad esempio, di rendere clonabili solo gli studenti e non gli insegnanti, o di implementare `Comparable` anche per i beni della scuola. Soprattutto, **avremo Comparable radicalmente diversi a seconda che siano persone o beni**, vale a dire polimorfismo di oggetto. Possiamo anche dire che le interfacce definiscono dei "tratti" o delle "funzioni" che possiamo inserire in qualunque altro tipo, e che poi entrano a far parte dell'eredità di quel tipo.

Per terminare, poniamoci una domanda: l'oggetto `t` di classe `Teacher`, a quanti tipi appartiene?

L'oggetto `t` appartiene a `Teacher` (la propria classe), `Employee` (la propria classe padre), `Person` (il nonno), `Object` (il bisnonno), ma anche a `Comparable < Person >` e a `Cloneable`, perché uno dei suoi antenati (`Person`) le implementava, e le ha *trasmesse*. `Teacher` potrà essere castato a uno qualunque di questi tipi.

Un oggetto appartiene alla propria classe e a tutti i super-tipi di quella classe. I super tipi di una classe sono tutte le classi antenato in ascendenza diretta (padre, nonno, bisnonno...), tutte le interfacce che implementa direttamente e tutte le interfacce che implementa indirettamente in quanto ereditate dalle classi antenato.

Per terminare, è possibile creare una interfaccia *estendendo* una o più interfacce. L'ereditarietà fra interfacce è multipla - una interfaccia può estendere quante interfacce vuole, finché non si creano condizioni di conflitto. Prima abbiamo utilizzato assieme `Comparable < Person >` e

Cloneable : volendo posso combinare in una interfaccia "CommonType":

```
package entities;
public interface CommonType extends Cloneable, Comparable
{
    // non definisco nessun nuovo metodo, ma combino i metodi di Cloneable e Comparable
    // ora mi basterebbe scrivere Person implements CommonType
}
```

Ovviamente posso aggiungere altri metodi. Ad esempio potrei scrivere:

```
package entities;
public interface CommonType extends Cloneable, Comparable
{
    boolean valid();
}
```

Se adesso scrivessi *class Person implements CommonType* , ogni oggetto Person sarebbe clonabile, dovrebbe avere il metodo int compareTo e il metodo valid(), a indicare la validità o meno dello stato. Questo meccanismo è flessibile e potente, e offre una grande libertà agli architetti di sistema, ma è anche complesso da gestire, e rischia di creare conflitti fra i metodi, come vedremo in seguito.

9.5 Metodi concreti nelle interfacce: default e static

Fino a Java versione 7, le interfacce potevano dichiarare solo metodi astratti. Da Java 8 in avanti è possibile dichiarare metodi concreti di due tipologie: static e default.

Un metodo static è analogo in tutto e per tutto a quello che troviamo nelle classi, ma è situato in una interfaccia, vale a dire in un tipo astratto. Per il resto, può essere usato allo stesso modo, invocandolo sulla interfaccia e non su classe o oggetto. Ad esempio:

```
// in PersonFileReader
static PersonFileReader make(String filename, String mode)
{
    return mode.equals("strict") ?
        new BasicCSVSchoolPersonFileReader() :
        new SafeCSVSchoolPersonFileReader();
}
```

Questo metodo si occupa di scegliere quale tipo di PersonFileReader creare. E' una "factory", qualcosa che vedremo meglio in seguito, è ottenuto come metodo statico dell'interfaccia PersonFileReader stessa. Ora possiamo modificare School in questo modo:

```
public School(String filename, String importRules) throws Exception
{
    //usando il metodo statico, già esistente, dell'interfaccia
    PersonFileReader reader = PersonFileReader.make(filename,importRules);
    people = reader.readFrom(filename);
    Collections.sort(people);
}
```

Non è nulla di nuovo, è un metodo statico, vale a dire un sottoprogramma che lavora solo sullo stato della classe. In questo caso, sullo "stato" dell'interfaccia, il che è una differenza importante: *le interfacce possono contenere solo variabili final static, vale a dire, costanti*. Una interfaccia *non definisce* uno stato di oggetto (non ha proprietà) e ha uno stato di interfaccia che è *immutabile*. Quindi potremmo avere:

```
public interface PersonFileReader
{
    String STRICTTYPE = "strict";

    static PersonFileReader make(String filename, String mode)
    {
        return mode.equals(STRICTTYPE) ?
            new BasicCSVSchoolPersonFileReader() :
            new SafeCSVSchoolPersonFileReader();
    }

    List <Person> readFrom(String filename) throws Exception;
}
```

Con STRICTTYPE public final static. L'interfaccia non contiene davvero uno stato, cioè qualcosa di mutabile: solo metodi e costanti. Il discorso diventa più complicato con i metodi di default. I metodi di default sono *metodi già implementati ma sovrascritti*, già presenti nell'interfaccia. L'interfaccia quindi non "regala" solo degli obblighi alle classi che la implementano, ma può regalare dei metodi già fatti, può comportarsi quasi come un "padre" per la classe.

Torniamo su PersonFileReader, e notiamo una cosa: di solito leggiamo una riga per volta, la convertiamo e cerchiamo di trarne una Person. Questo meccanismo è sempre uguale - leggo una riga, la trasformo in una persona, la aggiungo a una lista. Cambia semmai la logica di trasformazione, quindi volendo potrei riscrivere il tutto in questo modo:

```
public interface PersonFileReader
{
    String STRICTTYPE = "strict";

    static PersonFileReader make(String filename, String mode)
    {
        return mode.equals(STRICTTYPE) ?
            new BasicCSVSchoolPersonFileReader() :
            new SafeCSVSchoolPersonFileReader();
    }

    // lascio QUESTO ASTRATTO
    Person rowToPerson(String row);

    //mentre questo diventa concreto
}
```

```

default List <Person> readFrom(String filename) throws Exception
{
    Scanner reader = new Scanner(new File(filename));
    List <Person> res = new ArrayList();
    while(reader.hasNextLine())
        res.add(rowToPerson(reader.nextLine()));
    reader.close();
    return res;
}
}

```

Il metodo `readFrom(String filename)` ora è concreto. Vuol dire che sia `BasicCSVSchoolPersonFileReader` che `SafeCSVSchoolPersonFileReader` lo erediteranno. E in effetti per `Basic` andrà benissimo così, non avrà bisogno di riscriverlo: gli basterà implementare un pezzetto molto più piccolo, il solo `rowToPerson`, che ora DEVE essere public:

```

public class BasicCSVSchoolPersonFileReader implements PersonFileReader
{
    //readFrom lo eredito. mi basta specificare questo
    public Person rowToPerson(String row)
    {
        String parts[] = row.split(" ");
        switch(parts[0])
        {
            case "Employee":
                return new Employee
                (
                    Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                    Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8]
                );
            case "Student":
                return new Student
                (
                    Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                    Date.make(parts[6]), Date.make(parts[7]), Date.make(parts[8]), Integer.parseInt(parts[9])
                );
            case "Teacher":
                return new Teacher
                (
                    Integer.parseInt(parts[1]), parts[2], parts[3], Date.make(parts[4]), parts[5],
                    Integer.parseInt(parts[6]), Integer.parseInt(parts[7]), parts[8].split(":")
                );
            default:
                return null;
        }
    }
}

```

Questo meccanismo di dare-e-avere è tipico di interfacce e classi astratte. La classe `BasicCSVSchoolPersonFileReader` fornisce `rowToPerson`, e in cambio ha `readFrom()` già fatto, così come prima fornivamo `compareTo` di modo da poter usare `Collections.sort()`. Questo meccanismo, ricevere `readFrom()` "in cambio" di `rowToPerson()`, funzionerà per tutti i file che mantengono "un oggetto per riga", e sarà una base (un default, appunto) su cui costruire.

L'altra classe, `SafeCSVSchoolPersonFileReader`, resta quasi uguale, perché per quella classe il metodo `readFrom` di default non è sufficiente - non esegue i controlli sugli errori che invece sono l'essenza di `SafeCSVSchoolPersonFileReader`. Mi limito a rendere pubblico il metodo `rowToPerson`, che prima era privato, perché *devo* soddisfare l'interfaccia `PersonFileReader`, che contiene costanti, metodi astratti, metodi statici e metodi di default, cioè di oggetto, ma resta comunque una interfaccia e quindi sempre un contratto da onorare.

9.6 Interfacce funzionali e lambda

Esiste una categoria specifica di interfacce, nota come "funzionale". Una interfaccia funzionale è una interfaccia che dispone di *un solo metodo astratto*. Può avere metodi statici, metodi di default, costanti, ma al massimo *un* metodo astratto.

Un esempio di interfaccia funzionale è l'interfaccia standard Comparator. Comparator somiglia molto a Comparable, ma mentre Comparable confronta se stesso con qualcos'altro (ferdinando.comparareTo(carcarlo));, Comparator ha un metodo int compare(oggetto1, oggetto2). Quindi ci sarebbe qualcosa di questo tipo: comparator.compare(ferdinando,carcarlo);. L'interfaccia Comparable indica che *io* sono comparabile. L'interfaccia Comparator indica che *io* sono un *comparatore*, un oggetto che nella vita confronta altri oggetti e li ordina. Comparable definisce un solo ordine, quello "naturale". Person può avere un solo compareTo(). Un Comparator invece è *esterno* a Person, e posso definire tutti i Comparator che voglio:

```
package entities;
import java.util.Comparator;

public class PersonAgeComparator implements Comparator <Person>
{
    @Override
    public int compare(Person a, Person b)
    {
        //la logica è la stessa. Se restituisco un numero positivo, è più grande a
        //se è 0 sono uguali
        //se è negativo, è più grande b
        return a.getAge()-b.getAge();
    }

    public class PersonSurnameComparator implements Comparator <Person>
    {
        @Override
        public int compare(Person a, Person b)
        {
            return a.getSurname().compareTo(b.getSurname());
        }
    }
}
```

Ho creato due classi COMPARATORE. Uno confronta per età, l'altro per cognome. Ora posso usarli all'interno di Collections.sort(), come secondo argomento:

```
School school = new School("people.txt");
//ordino per età:
Collections.sort(school.getAll(), new PersonAgeComparator());
System.out.println(school.getAll());
Collections.sort(school.getAll(), new PersonSurnameComparator());
System.out.println(school.getAll());
// output:
// stampa ordinata per età
//[Student 4:E F , 01/01/2003 M enrolled on 01/09/2015 , age: 18 current state:ACTIVE, Employee nr. 6:A B, 31 years, service:10
occupation:Secretary, salary:1100 euro, Employee nr. 3:C D, 37 years, service:10 occupation:Teacher, salary:1200 euro , subjects:[Geometry,
Math]]
// stampa ordinata per cognome
//[Employee nr. 6:A B, 31 years, service:10 occupation:Secretary, salary:1100 euro, Employee nr. 3:C D, 37 years, service:10
occupation:Teacher, salary:1200 euro , subjects:[Geometry, Math], Student 4:E F , 01/01/2003 M enrolled on 01/09/2015 , age: 18 current
state:ACTIVE]
```

Ho creato due oggetti (un PersonAgeComparator anonimo e un PersonSurnameComparator anonimo), da fornire al metodo static sort di Collections, per farglielo usare al posto dell'ordinamento naturale (compareTo di Comparable). Per farlo ho dovuto creare due classi e implementare due volte la stessa interfaccia (Comparator < Person >).

C'è un modo diverso, per quanto orribile, ed è la scrittura di *classi anonime*.

Non stiamo parlando di **oggetti** anonimi, ma di classi anonime, usa e getta, di tipi che creo per poterli usare una sola volta:

```
School school = new School("people.txt");
// voglio ordinare per età:
Collections.sort(school.getAll(), new Comparator<Person>() {public int compare(Person a, Person b) { return a.getAge() - b.getAge();}});
System.out.println(school.getAll());
```

Questo orrore senz'anima ma perversamente affascinante è la creazione di una *classe anonima* che implementa l'interfaccia Comparator < Person > e il suo istanziamento istantaneo per creare un oggetto usa-e-getta che verrà usato da sort.
Un modo meno brutale di vederlo sarebbe stato:

```
School school = new School("people.txt");
// creo allo stesso tempo la classe anonima e l'oggetto!
Comparator <Person> byAge = new Comparator<Person>() {public int compare(Person a, Person b) { return a.getAge() - b.getAge();}};
Collections.sort(school.getAll(), byAge);
System.out.println(school.getAll());
```

Ricapitoliamo, perchè la stranezza del tutto può essere dura da digerire: sto creando all'interno del codice una classe anonima e un oggetto. L'oggetto sarà di tipo Comparator < Person >, e l'ho fatto solo per poter passare a Collections.sort() qualcosa da usare per ordinare i dati.

E' già meno prolioso di prima. Non ho dovuto creare un file a parte con una classe (PersonAgeComparator) che userò una volta sola in tutta la vita. Ho creato contestualmente classe e oggetto, rispettando la concordanza dei tipi e implementando questa interfaccia. Questo posso farlo per *qualunque* interfaccia, ma per le interfacce funzionali posso beneficiare di una sintassi migliore. C'è UN solo metodo astratto, e quindi posso *creare classe e oggetto specificando solo il metodo*. In questo caso il metodo è compare, prende in ingresso due persone e restituisce un intero. Posso riscrivere il programma di sopra in questo modo:

```
School school = new School("people.txt");
//questa riga 3 è l'equivalente della riga 3 dell'esempio precedente
Comparator <Person> byAge = (a,b) -> (a.getAge() - b.getAge());
Collections.sort(school.getAll(), byAge);
System.out.println(school.getAll());
```

Adesso siamo nell'ambito di una soluzione elegante, ma difficile da seguire. Cercheremo di spiegarla.

- l'unico metodo astratto di Comparator è compare
- compare prende due ingressi (a,b), che sono in questo caso Person e presto vedremo perchè
- compare produce un intero
- matematicamente potremmo dire che compare è una funzione f(a,b) -> (int)
- e in effetti abbiamo scritto (a,b) -> (int)

Quella sintassi viene detta *lambda*, ed è un modo per scrivere in maniera sintetica l'unico metodo astratto di una interfaccia funzionale. La lambda produce un oggetto del tipo dell'interfaccia - e in effetti la nostra lambda produce un Comparator < Person >. Quando scrivo (a,b) -> (int), sto implementando compare, e creando a partire da compare() una classe anonima che implementa Comparator, e dalla classe un oggetto, che in questo caso viene salvato in byAge, per poi essere utilizzato in Collections.sort().

Le lambda sono estremamente comuni nei programmi Java moderni. Vediamo un'altra applicazione: supponiamo di voler estrarre tutti i docenti con almeno dieci anni di esperienza dall'archivio. Una soluzione classica sarebbe la seguente:

```
List <Teacher> res = new ArrayList <Teacher> ();
for(Person p: people)
    if(p instanceof Teacher && ((Teacher)p).getServiceYears() >= 10) res.add(p);
//ho fatto casting-in-linea e usato la lazy evaluation. La seconda condizione non viene
//valutata se la prima è falsa (lazy evaluation), e di conseguenza non ho eccezioni di casting
```

E' una operazione di filter standard - tengo alcuni elementi, scarto altri. E in effetti, a ogni elemento sto attribuendo un valore, vero o falso, e tengo solo quelli per cui la condizione è vera. C'è un modo migliore di farlo:

```
//concetto GENERICO di filtro su Person
//qualunque tipo di filtro. Entra una Person, esce un booleano
// PersonFilter.java
public interface PersonFilter
{
    boolean test(Person p);
}
//e ora di nuovo nell'oggetto School...
public List <Person> filter( PersonFilter filter )
{
    //Mi arriva un oggetto di tipo PersonFilter! Un oggetto che conterrà un metodo keep ("tieni")
    //se il metodo keep mi restituisce "vero", tengo la persona, altrimenti la scarto
    //in questo metodo io NON SO che filtro sarà. Arriva da fuori.
    List <Teacher> res = new ArrayList <Teacher> ();
    for(Person p: people)
        if(filter.keep(p)) res.add(p);
    return res;
}
//e ora main()
```

```

School school = new School("people.txt");
// tutte le persone sopra i 30 anni
System.out.println(school.filter((p)->(p.getAge()>30));
// quello che ho fra parentesi è il metodo keep dell'interfaccia PersonFilter.
// da p di tipo Person a un boolean, come specificato dal metodo

// tutti gli insegnanti sopra i 10 anni di esperienza
System.out.println(school.filter((p)->(p instanceof Teacher && (((Teacher)p).getServiceYears()>=10));
// idem

```

Molti dei metodi richiesti dall'utente si possono ridurre ad applicazione del metodo filter con un PersonFilter come parametro. Si può lavorare in maniera simile anche per le operazioni di mappatura e di riduzione, per quanto sia meno immediato.

Le lambda hanno tuttavia una serie di limiti per cui non è sempre possibile utilizzarle. Li vedremo meglio con la pratica. Per adesso, questa sintassi è qualcosa a cui ci dobbiamo abituare. Ricordiamoci che le lambda sono un modo rapido per creare, con la stessa scrittura, un metodo, una classe e un oggetto, che verrà poi usato in un qualche momento.

Avrei anche potuto scrivere la prima lambda in questo modo:

```
PersonFilter filter = (p)->{ return p.getAge()>30};
```

Questa è la forma estesa: non sempre potremo scrivere semplicemente il valore di ritorno, come abbiamo fatto nel primo caso. In molti casi dovremo scrivere un metodo composto di più righe, e come sappiamo quando abbiamo più righe abbiamo un blocco di codice e uno scope, ed essendo un metodo deve avere un return, quando non è void. Nel caso di un metodo void, avremo un blocco di codice senza return (se non per interrompere il metodo anzitempo).

Termineremo il discorso sulle lambda in seguito. Ora cerchiamo di ricapitolare quanto visto in questo capitolo.

9.7 Riepilogo della programmazione a oggetti avanzata

- Dividiamo i tipi in tipi concreti e tipi astratti. I tipi concreti sono le classi. I tipi astratti sono interfacce e classi astratte. I tipi astratti non possono essere istanziate (non posso creare oggetti direttamente)
- Un metodo si dice astratto quando è ridotto alla sua dichiarazione, senza corpo. Ad esempio boolean valid();. I metodi astratti possono esistere solo nei tipi astratti.
- Una classe può estendere una sola classe padre, ma implementare quante interfacce vuole. Una interfaccia può estendere quante interfacce vuole. Il rapporto fra classe figlia e classe padre è detto extends. Il rapporto fra classe implementante e interfaccia è detto implements. I due rapporti sono casi particolari di "is_a" o "instanceof". Dire che Bar implements PuntoDiRistorazione significa che un oggetto di classe di tipo Bar è un punto di ristorazione.
- Una classe può contenere proprietà di oggetto, metodi di oggetto, proprietà di classe e metodi di classe.
Una classe astratta può contenere proprietà di oggetto, metodi di oggetto, proprietà di classe, metodi di classe e metodi astratti (di oggetto). Una classe astratta può contenere metodi astratti, ma non è costretta a contenerne. Se una classe contiene metodi astratti, invece, deve essere astratta. I metodi astratti non sono obbligatori, ma se ci sono la classe è astratta.
Una interfaccia Java 7 può contenere solo metodi astratti. Una interfaccia Java 8 può contenere proprietà final static (costanti), metodi statici (di tipo) e metodi di default (di oggetto, già implementati ma sovrascrivibili).
- Una interfaccia con un solo metodo astratto (potrebbe averne altri concreti) è detta "funzionale", o "FunctionalInterface". Per una interfaccia del genere possiamo creare oggetti di una classe anonima che soddisfa il contratto dell'interfaccia utilizzando la sintassi delle *lambda*, la cui sintassi generale è (parametro1, parametro2, ... parametron del metodo astratto) -> { blocco di codice con return del tipo del metodo da implementare }. Laddove il metodo si possa scrivere con una sola riga, è sufficiente la forma: (p1...pn) -> (r1), con p1..p2 i parametri ed r1 il ritorno, che sarà un'espressione del tipo corretto.
- Interfacce notevoli con cui abbiamo lavorato: List (che definisce il comportamento di una lista), Set (che definisce il comportamento di un insieme), Map (che definisce il comportamento di una mappa), Cloneable (che indica a Java che l'oggetto sarà clonabile), Comparable (che obbliga l'oggetto a fornire il metodo int compareTo(), l'ordine naturale) e Comparator, che serve per creare oggetti "comparatori" che abbiano il metodo compare. I Comparator possono essere usati per ordinare i dati con Collections.sort()

9.8 Caso di studio: il progetto School con uso di lambda e interfacce

Portiamo a termine il progetto School, partendo dal rivedere quelli che erano i requisiti, vale a dire i servizi richiesti, ripescandoli dal paragrafo apposito:

- Possibilità di cercare una persona dato il suo badge id (identificazione)
- Possibilità di cercare il personale per occupazione (insegnanti inclusi)
- Possibilità di filtrare per fascia di età il nostro personale
- Possibilità di calcolare il tempo di percorrenza media dei nostri studenti dall'iscrizione al diploma
- Possibilità di calcolare il tasso di dropout
- Possibilità di calcolare il costo annuo del personale

E' buona prassi separare ciò che ci viene richiesto (il cosa) da come lo otterremo (il come). Di conseguenza, prima di scrivere la classe per l'oggetto aggregatore School, scriverò la sua *interfaccia*.

```
package entities;
import java.util.List;

<"/>"i servizi che dovremo offrire col componente School"
public interface SchoolService
{
    // Possibilità di cercare una persona dato il suo badge id (identificazione)
    Person person(int badgeid);
    // Possibilità di cercare il personale per occupazione (insegnanti inclusi)
    List <Person> employeesByOccupation(String occupation);

    // Possibilità di filtrare per fascia di età il nostro personale
    List <Person> employeesByAgeRange(int minage, int maxage);

    // Possibilità di calcolare il tempo di percorrenza media dei nostri studenti dall'iscrizione al diploma
    double averageStudentStay();

    // Possibilità di calcolare il tasso di dropout
    double dropoutRate();

    // Possibilità di calcolare il costo annuo del personale
    int totalCost();
    // aggiungo anche la possibilità di vedere tutti gli oggetti:
    List <Person> getAll();
}
```

Definita l'interfaccia, farò in modo che School la implementi ("onorò il contratto").

Questo potrebbe sembrare superfluo, ma non lo è: ad ora School pescherà i dati da un file, ma presto leggerà da un database. In quel caso noi vorremo mantenere lo stesso contratto, lo stesso "cosa", ma il "come" cambierà radicalmente, dovendoci appoggiare a una tecnologia diversa sotto.

Le interfacce servono per isolare i nostri clienti dai nostri cambiamenti che per loro non sono rilevanti. Il main() vorrà da School quei servizi e solo quei servizi, e non gli importerà come vengono memorizzati gli studenti, o come vengono convertiti o ricaricati. Quello che ci interessa sono le interfacce, e questo si chiama *programmare per contratto*.

Ora passiamo all'implementazione di School:

```
package entities;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class School implements SchoolService
{
    // le mie persone sono memorizzate sotto forma di lista
    List<Person> people = new ArrayList <Person> ();
    // vuol dire che avrò solo oggetti di classe Person? Sì. Ma anche Teacher, Employee e Student sono Person

    // mi arriva il nome di un file
    public School(String filename) throws Exception
    {
        PersonFileReader reader = new BasicCSVSchoolPersonFileReader();
        people = reader.readFrom(filename);
        Collections.sort(people);
    }

    // Mi arriva il nom di un file e una parola chiave fra "strict" e "loose"
    public School(String filename, String importRules) throws Exception
    {
        // late binding! Posso creare un componente o l'altro indifferente
        // da un punto di vista formale sono entrambi PersonFileReader
        // in termini di tipo concreto, sarà o Basic o Safe
    }
}
```

```

// ma NON ci importa, perchè entrambi hanno il metodo readFrom
// e non possono fare a meno di averlo, perchè onorano l'interfaccia. DEVONO avere quel metodo
PersonFileReader reader = PersonFileReader.make(filename,importRules);
people = reader.readFrom(filename);
//userà per ordinare gli oggetti l'ordine di comparable
Collections.sort(people);
}

//mi arriva un vettore: poco male. lo sposto nella lista
public School(Person[] people)
{
    for(Person person:people)
        this.people.add(person);
}

// ora arrivano i metodi interessanti: preparo un metodo generico per filtrare le persone:
// mi arriva un filtro, vale a dire un oggetto di una classe (magari anonima) che implementa
// PersonFilter, e lo uso per decidere chi tenere e chi scartare (metodo keep)
private List <Person> _filter(PersonFilter filter)
{
    List <Person> res = new ArrayList <Person> ();
    // mi basta usare l'interfaccia funzionale PersonFilter che abbiamo visto prima
    // sotto forma di lambda
    for(Person p:people)
        if(filter.keep(p))
            res.add(p);
    return res;
}

// mi basta richiamare il metodo sopra
public List<Person> employeesByOccupation(String occupation)
{
    return _filter((p)->(p instanceof Employee && ((Employee)p).getOccupation().equals(occupation)));
}

@Override
public Person person(int badgeid)
{
    //posso usare filter anche qui:
    List <Person> res = _filter((p)->(p.getBadgeid()==badgeid));
    // se la lista contiene un elemento, lo restituisco, altrimenti non lo ho trovato. Restituisco 0
    return res.size()>0 ? res.get(0) : null;
}

@Override
public List employeesByAgeRange(int minage, int maxage)
{
    // è sempre un filtro...
    return _filter((p)->(p.getAge()>=minage && p.getAge()<=maxage));
}

@Override
public double averageStudentStay()
{
    // anche qui è comodo usare _filter
    List <Person> graduates = _filter((p)->(p instanceof Student && ((Student)p).getDiploma()!=null));
    double sum = 0;
    for(Person p:graduates)
    {
        Student s = (Student) p;
        sum+=(s.getDiploma().getYear() - s.getEntry().getYear());
    }
    return sum / graduates.size();
}

@Override
public double dropoutRate()
{
    //calcolo il numero di ritirati e lo divido per il numero totale di non attivi (i non attivi sono diplomati o ritirati)
    return (_filter((p)->((Student)p).getRetired()==null)).size()*100.00 / _filter((p)->(!((Student)p).isActive())).size();
}

@Override
public int totalCost()
{
    int sum = 0;
    for (Person p:people)
        sum+=p.getCost();
    return sum;
}

public List <Person> getAll()

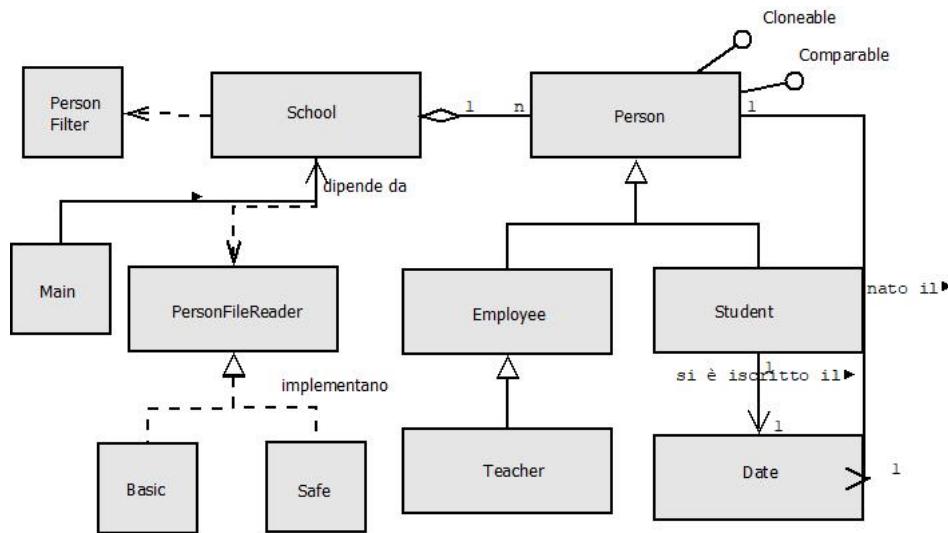
```

```

    {
        return people;
    }
}

```

La scrittura del main() è lasciata allo studente come esercizio, ma noi riportiamo il Class Diagram UML, inclusivo di main. Noterete dei pallini bianchi accanto a Person: sono una forma alternativa di indicare l'implementazione. Ogni pallino bianco è una interfaccia, e infatti Person implementa Cloneable e Comparable.



9.9 Approfondimenti: definizione di classi e interfacce generiche

Ragioniamo brevemente su List: notiamo che abbiamo sempre scritto List < Person >, o List < Integer > o List < String >. Di volta in volta abbiamo definito un *tipo* diverso per il suo contenuto ma *il suo funzionamento non è cambiato*. Richiamando List.add() inserivamo un elemento del tipo che avevamo scelto. Richiamando size() ottenevamo sempre un intero, il numero degli elementi presenti. Non ci interessava, per il suo funzionamento, di che tipo si trattasse.

List è in effetti una interfaccia *generica*, o meglio, che *usa il sistema dei generics in Java*. Si tratta di interfacce e classi pensate per lavorare non su tipi specifici ma su *tipi più o meno arbitrari*. Una List infatti funziona su String, Integer, Person... su qualunque oggetto in realtà.

Scriviamo una classe o una interfaccia generica quando abbiamo individuato un *comportamento che resta valido a prescindere dal tipo*, o almeno nell'ambito di alcuni tipi. Le operazioni di aggiunta, rimozione, conteggio, scorrimento, sono valide a prescindere dal tipo, e possono essere riunite in una interfaccia di nome "List".

Il *tipo* che trattiamo può variare di volta in volta, e viene dichiarato fra parentesi angolari (< >). Quindi List < Integer > è una List di interi. Map < K, V > è una mappa con chiave di tipo K e valori di tipo V.

Comparator < Person > è un oggetto che saprà confrontare fra di loro due Person: il tipo Person viene usato per entrambi i parametri del metodo compare(). Il tipo fra parentesi angolari viene anche detto "parametro tipo", e può essere più di uno, come abbiamo visto per le mappe.

In effetti, Comparator, List, Map e Set sono tutte interfacce generiche, cioè pensate per definire operazioni su tipi che non conosciamo a priori.

Ora, riflettiamo sulla interfaccia PersonFilter. E' una *interfaccia funzionale* che richiede un metodo boolean keep(Person p);, e restituirà true se p rispetta una condizione, false altrimenti. E' un sistema estremamente generico, quando si tratta di filtrare Person. Ma se volessimo scrivere un filtro universale, in grado di lavorare su qualunque oggetto? Dovremmo usare il sistema dei generics:

```
public interface Filter <X>
{
    boolean keep(X x);
}
```

Questa è la forma in assoluto più generica del concetto di controllo su un elemento. Entra un elemento di un tipo X che non conosciamo, ed esce un boolean. Lo ripetiamo: noi non conosciamo il tipo X. X è un tipo che verrà fornito in fase di creazione dell'oggetto.

Supponiamo di avere due tipi diversi di entità nel nostro progetto School: le Person e i beni ("Asset"). Gli Asset hanno un costo storico di acquisto, getPrice(), che sarà importante per calcolare l'ammortamento, ma per ora facciamo qualcosa di diverso: scriviamo un filtro per le Person e uno per gli Asset:

```
Filter <Person> ageFilter = (p)->(p.getAge()>30);
Filter <Asset> costFilter = (a)->(a.getPrice()>1000);
```

Non abbiamo avuto bisogno di creare un tipo PersonFilter e un tipo AssetFilter. Abbiamo utilizzato il concetto generico di filtro, Filter < X >, *parametrizzandolo*, esattamente come abbiamo parametrizzato List, Set, Map, Comparator. Il primo oggetto ha come type parameter la classe Person, il secondo ha come type parameter la classe Asset. **In questo caso il parametro è un tipo, non un oggetto**.

Notiamo che Java in realtà ha già questi concetti e queste interfacce, e li useremo diffusamente nella pratica nei prossimi capitoli. Per adesso studiamo la versione "fatta in casa", per capire cosa c'è sotto.

9.10 Un caso di studio: la struttura dati Deck < X >

Scegiamo di rappresentare un mazzo di carte da gioco. Potrebbero essere napoletane, francesi, carte di qualche gioco collezionabile... non ci interessa. Accetteremo come carte perfino degli Integer, dei numeri interi comuni.

Vogliamo che il mazzo di carte offra le seguenti funzioni:

- mischiare il mazzo
- estrarre una "mano" di n carte
- dirci tutte le carte del mazzo

Per buona prassi, definiremo prima il contratto, l'interfaccia Deck, e poi lo implemetneremo:

```
package entities;
import java.util.List;

public interface Deck <X>
{
    // la lista di tutte le carte
    // una List, non un set, perchè potremmo avere più copie della stessa carta nel mazzo
    List <X> cards();

    // mischiare il mazzo e quindi rimettere tutto dentro
    void shuffle();

    // estrarre una mano di n carte
    List <X> hand(int n);
}
```

Questa è una interfaccia generica, che lavorerà su oggetti qualunque. Ora dobbiamo andare a implementarla:

```
package entities;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

// una implementazione generica dell'interfaccia generica Deck
// X è sempre il nostro parameter type
public class SimpleDeck implements Deck
{
    // carte presenti. Ci arrivano dal costruttore. Gradualmente si svuoterà
    List <X> content = new ArrayList <X> ();
    // carte estratte. Inizialmente è vuoto
    List <X> drawn = new ArrayList <X> ();

    public SimpleDeck(X[] v)
    {
        for(X x:v)
            content.add(x);
    }

    @Override
    public List <X> cards()
    {
        // anche se non conosco X, so che content sarà List <X>
        return content;
    }

    @Override
    public void shuffle()
    {
        // mischiare le carte vuol dire rimettere tutto dentro :
        content.addAll(drawn);
        // e ovviamente svuotare le estratte
        drawn.removeAll(drawn);
        //questa è perversa: ordino gli elementi completamente a caso
        //generando due numeri casuali con Math.random() e sottraendoli uno all'altro
        Collections.sort(content, (x,y)->((int)(Math.random()*100 - Math.random()*100)));
    }

    @Override
    public List <X> hand(int n)
    {
        // estraggo n carte
        List <X> res = new ArrayList <X>();
        // devo vedere quante carte mi rimangono
        // left = rimaste
        if(content.size()<=n)
```

```

        return content;
    else
        for(int i=0;i<n;i++)
            res.add(content.get(i));
    //aggiungo tutte le estratte all'elenco delle estratte
    drawn.addAll(res);
    content.removeAll(res);
    return res;
}
}

```

E ora il main(), che estre 4 volte di fila due mani da due carte (String, in questo caso) ciascuna:

```

// main()
Deck <String> deck = new SimpleDeck <String>(new String[]{ "A", "B", "C", "D" , "E"});
for(int i = 0;i <4;i++)
{
    deck.shuffle();
    System.out.println(deck.hand(2));
    System.out.println(deck.hand(2));
}

```

Lo studente è incoraggiato a studiare il codice di SimpleDeck e a capirne il funzionamento insiemistico. *Non* è rigoroso: c'è modo di mandarlo in crisi, e più di uno, quindi fare hardening di questo codice sarebbe un buon esercizio.

Quello che abbiamo visto in pratica è stata la definizione di una interfaccia generica (Deck < X >) che definisce il contratto per una classe generica (SimpleDeck < X >), ed entrambe *lavorano senza conoscere il tipo su cui lavorano*. Trattano qualunque tipo X come un oggetto, senza preoccuparsi di cosa sia davvero. Al punto che posso modificare il programma sopra in questo modo:

```

// main()
Deck <Integer> deck = new SimpleDeck <Integer>(new Integer[]{ 5, 6, 1, 10, 11, 9, 8, 3});
for(int i = 0;i <4;i++)
{
    deck.shuffle();
    System.out.println(deck.hand(2));
    System.out.println(deck.hand(2));
}

```

E tutto funzionerà allo stesso modo, questa volta che le mie carte sono interi. Ma... e se fossero vere Card?

```

// Card.java
class Card
{
    String seed;
    int number;
    // getter e setter e costruttori standard
}
// main
public class TestDeck
{
    public static void main(String[] args)
    {
        Deck <Card> deck = new SimpleDeck <Card>
        (
            new Card[]{ new Card("P", 1), new Card("P", 3), new Card("C", 1 ), new Card("Q", 3)}
        );
        for(int i = 0;i <4;i++)
        {
            deck.shuffle();
            System.out.println(deck.hand(2));
            System.out.println(deck.hand(2));
        }
    }
}

```

Di nuovo, il funzionamento non cambia. Sono passato da String a Integer, da Integer a Card, e tutto continua a funzionare, e tutto per via della potenza dei generics, e perchè sto usando tutto come "oggetto". Notate che non chiamo alcun metodo sugli elementi di tipo X, perché in effetti non ho idea di che metodi abbiano. Potrò contare sul fatto che abbiano `toString()` (perchè tutto ha `toString()`), `equals` e poco altro.

In alcuni casi questo è limitante, e rinuncerò a un po' di flessibilità per ottenere maggiore espressività. E' il caso dei generics che lavorano su *parameter types vincolati*. Li approfondiremo a breve, ma per chi volesse portarsi avanti, questa è un'ottima fonte: <https://docs.oracle.com/javase/tutorial/java/generics/index.html>

10 - I Database e MySQL

10.1 Introduzione

Un database ("base di dati") è una forma evoluta di persistenza.

La persistenza del dato non è un concetto nuovo. Perdiamo le nostre variabili al riavvio del programma, e in generale allo spegnimento della macchina, essendo probabilmente mantenute in RAM. Per rendere l'informazione persistente la salviamo su memoria di massa (i files che abbiamo usato fino a ora ne sono un esempio). I database sono una evoluzione di questi strumenti.

Un database non si limita a memorizzare i dati, ma li organizza secondo dei concetti che vadano oltre la brutale "sequenza". La maggior parte dei database dispone anche di un linguaggio interno di interrogazione (ricerca) e modifica degli stessi, che rende più semplice l'aggiornamento, l'inserimento e soprattutto la lettura dei dati ivi memorizzati. La tecnologia per la realizzazione di database attualmente più diffusa è quella dei database "relazionali", o SQL (Structure Query Language), dal nome del linguaggio che usiamo per "parlare" col database.

Si tratta di strutture in cui il concetto fondamentale è la tabella, e in cui possiamo definire relazioni fra tabelle diverse, ma anche fra righe della stessa tabella. Nella prossima sezione definiremo i concetti fondamentali di un database MySQL. Per adesso limitiamoci a vederli come versioni molto evolute dei nostri file archivio, in cui ogni database conterrà l'equivalente di diversi files, e di conseguenza lo *stato* di molti oggetti. Java sarà "cliente" del database, utilizzandolo per leggere e scrivere gli oggetti e renderli permanenti, ma anche per eseguire ricerche avanzate e calcoli bypassando il lavoro algoritmico in Java.

10.2 Elementi di un database SQL

Il modello relazionale, o modello SQL, memorizza i dati in una o più *tabelle*.

Una tabella è composta di righe e colonne. Non esiste un vincolo sul numero di righe, ma tutte le righe hanno le stesse colonne. Le colonne prendono anche il nome di "campi", e sono tipizzate. Questo vuol dire che tutte le righe hanno le stesse colonne, con valori diversi ma sempre dello stesso tipo. Un esempio potrebbe essere la tabella Person con campi name, surname e dateofbirth, di tipo rispettivamente varchar(100), varchar(100) e date. varchar è l'equivalente di String in Java, con delle differenze marginali (possiamo specificare la lunghezza massima), mentre date è un tipo specializzato per memorizzare le date (e fonte di notevoli grattacapi con i formati, come tutte le date).

La tabella Person potrebbe contenere due righe, con valori:

Person

Name	Surname	Date of birth
James	Watson	1990-01-01
Jill	Watson	1985-01-01

Potrebbe non sembrare un grosso vantaggio rispetto ai files che abbiamo usato fino a ora, ma ci sono dei grossi vantaggi: abbiamo la sicurezza che il terzo valore sarà una data valida. La tipizzazione è, almeno nella maggior parte dei casi, vincolante. Una riga che non presenta una data valida non verrà salvata nel database.

Inoltre, saremo in grado di modificare il valore di un singolo campo o di una singola riga senza eccessive difficoltà, a differenza di quanto avremmo dovuto fare modificando le righe di un file.

Questo è possibile perché il database ci offre un linguaggio specifico per interagire coi dati, il già citato SQL. Le successive tre istruzioni (vedremo dopo dove e come inserirle) aggiungono, modificano e cancellano righe:

```
INSERT INTO PERSON VALUES('SCOTT', 'KIRK', '1990-02-01');
UPDATE PERSON SET DATEOFBIRTH='1986-01-01' WHERE NAME='JILL';
DELETE FROM PERSON WHERE NAME='JAMES';
```

Queste tre righe sono i tre comandi fondamentali di modifica del dato in SQL. Abbiamo inserito una nuova riga (viene messa automaticamente in fondo), modificato una riga esistente (la seconda riga si legge "modifica la tabella Person mettendo data di nascita pari al primo gennaio 1986 in tutte le righe in cui la colonna nome sia Jill"), e cancellato una riga esistente ("cancella tutte le righe in cui la colonna name ha valore James"). Di contro, potremo leggere solo quello che ci interessa, selezionando potenzialmente solo alcune righe o solo alcuni campi:

```
SELECT NAME,SURNAME FROM PERSON WHERE YEAR(DATEOFBIRTH)=1986;
```

Selezionerà solo i nomi e i cognomi delle persone nate nel 1986. Questa operazione è di selezione (filter) e proiezione (mappatura), espressa in maniera estremamente sintetica tramite un query, vale a dire una interrogazione. Approfondiremo in dettaglio questo aspetto.

Inoltre, lo stesso database potrebbe avere tabelle diverse con campi diversi. Potremmo avere, ad esempio, Person e Book, e possiamo sempre aggiungere, e rimuovere, tabelle e righe. Possiamo anche modificare la struttura di tabelle già esistenti, per quanto questo ci esponga a problemi che vedremo in seguito.

Ci accorgiamo rapidamente che la tabella Person corrisponde in qualche modo alla classe Person. In effetti, la tabella Person memorizza lo stato di quelli che in Java potrebbero essere oggetti di tipo Person. Parte del nostro lavoro sarà trasformare queste righe in oggetti funzionanti, di modo da poterli usare in Java, e viceversa trasformare gli oggetti in righe.

Formalmente, diremo che un database d, nella sua forma più essenziale, è un insieme di tabelle {t1...tn}. Ogni tabella è un insieme di righe {r1...rn}. Ciascuna riga è composta di campi {c1...c2}, e all'interno della tabella tutte le righe hanno gli stessi campi e questi ultimi sono tipizzati come abbiamo visto in precedenza.

Un database SQL può contenere inoltre altri oggetti che utilizzano tipicamente i dati contenuti nelle tabelle. In gergo, diciamo che un database SQL è programmabile, ma il come lo vedremo bene in seguito.

10.3 SQL, le sue estensioni e i DBMS

Così come per creare un programma abbiamo bisogno di un compilatore e magari di un IDE, per creare un database dobbiamo utilizzare un DBMS, vale a dire un Database Management System, letteralmente un "gestore di database", un tool per la creazione e la gestione di database secondo un certo standard.

I DBMS relazioni sono legati a doppio filo a SQL. In termini generali, tutti i DBMS relazioni "parlano" SQL standard, ma molti hanno un "dialetto" che lo estende permettendo cose non previste da SQL standard. Ogni DBMS offre circa gli stessi servizi di base (creazione di un database, creazione delle tabelle, inserimento, modifica e lettura dei dati memorizzati), ma le istruzioni da scrivere per ottenerli potrebbero variare, seppur di molto poco, in alcuni casi.

Inoltre, alcuni DBMS potrebbero non garantire alcuni dei vantaggi che abbiamo enunciato sopra. SQLite, un DBMS estremamente popolare in ambito mobile e Android per la sua leggerezza, non effettua controllo sui tipi dei campi, per cui possiamo trovare valori non compatibili in colonne teoricamente controllate. Altri DBMS non avranno tool equivalenti a quelli che vedremo.

In ambito industriale hanno forte diffusione SQLite (per la sua leggerezza), Oracle (per progetti di grandi rilievo, di natura enterprise, molto spesso basati su Java), SQL Server (la soluzione di Microsoft) e MySQL, che sarà il nostro strumento di sviluppo di riferimento in ambito database. Citiamo anche, per completezza, un DBMS non relazionale, non SQL, di nome MongoDB, usatissimo per i progetti Big-Data, per cui il modello relazionale fatica ad avere prestazioni accettabili.

10.4 MySQL Server

MySQL supporta tutto SQL standard e aggiunge delle proprie estensioni, un proprio dialetto, ed è una soluzione molto richiesta per progetti piccoli e medi. E' gratuito e può essere installato facilmente anche su macchine non eccessivamente prestanti. Per i dettagli dell'installazione rimandiamo alla guida in appendice.

MySQL è una applicazione client - server. Nella pratica, installare MySQL significa prima di tutto installare un server, vale a dire un programma che eroga un servizio su una rete tramite una porta. Vediamo di chiarire i concetti di base dell'architettura di sistema client-server e di un DBMS client-server in particolare:

- **Server hardware, o fisico:** la macchina su cui viene installato un programma che offre un servizio, vale a dire il server software. Un server fisico può ospitare diversi server software, a volte anche dello stesso tipo.
- **Server software:** il programma che offre un servizio. Viene installato su una macchina (il server fisico), e prende possesso di una "porta". Una porta è un numero intero che serve a indicare la "posizione" del programma server sul server fisico. Ad esempio, sulla stessa macchina, che potremmo chiamare SERVER1, potremmo avere due server, un server web (che produrrà siti web, e che vedremo in seguito) alla porta 80 e un server MySQL (che permetterà di immagazzinare e leggere dati) alla porta 3306. 3306 è la porta standard di MySQL, ma è possibile cambiarla, e in effetti potremmo avere più server software MySQL sullo stesso server fisico, a patto di avivarli su porte diverse. In questo caso parleremo di istanze diverse di MySQL. Potremmo avere, ad esempio, una istanza sulla porta 3306 e una sulla porta 3307, con database diversi.
- **Istanza:** una istanza di MySQL è una installazione di MySQL Server legata a una porta. Può essere attiva o meno (il server può o meno accettare richieste), e ogni istanza ha i suoi database (una istanza, n database) e i suoi utenti.
- **Database:** una collezione di tabelle e di altri oggetti che dalle tabelle prendono i dati, ma non solo. I database MySQL possono contenere sottoprogrammi e altra logica. In generale, possiamo ragionare in questi termini: una istanza, n database, un database, m tabelle, una tabella, k righe, ciascuna con x campi tipizzati.
- **Client fisico, o hardware:** la macchina su cui è installato il programma che userà il server. A volte è la stessa macchina su cui è installato il server. Un client fisico può eseguire diversi client software allo stesso tempo.
- **Client software:** il programma che utilizza il servizio offerto da un server software. Un esempio di client software è il browser (un "client web", letteralmente).

Installato il MySQL server (il programma che fornirà i dati) ci verrà chiesto di specificare un utente "root" (radice, l'admin supremo dell'istanza, con accesso a tutti i database) e una password. In fase di sviluppo, e per semplificarcici la vita, useremo l'utente root per lavorare ma è una facilitazione brutale. Nella pratica si definiscono degli utenti con dei permessi limitati a singoli database e a volte a singoli oggetti del database, ma questo esula dagli scopi del corso e dal tempo a nostra disposizione. Nome utente e password ci serviranno in fase di connessione al database. Non potremo connetterci senza.

La connessione avverrà fra un client MySQL (un programma in grado di "parlare" MySQL, di inviare comandi al server e di leggerne le risposte) e il server. Il client potrebbe essere su qualunque macchina collegata alla stessa rete, ma non è insolito che il client sia sulla stessa macchina del server.

Nell'uso pratico, Java farà da client per il server MySQL, tramite apposite librerie (JDBC), ma creare i database direttamente in Java è scomodo. Ci appoggiamo a un altro client specializzato nella creazione e nell'amministrazione di database (e istanze) MySQL - MySQLWorkbench, la cui installazione è dettagliata nella guida allegata.

Il codice che riportiamo di seguito è pensato per essere eseguito nelle finestre di query di MySQLWorkbench.

10.5 Creazione di un database

L'icona nuova finestra di query in MySQLWorkbench ci mette di fronte a un "foglio bianco". Possiamo scrivere i nostri comandi in questa finestra e attivarli col tasto play nella parte superiore della barra.

Notiamo che i comandi SQL sono permanenti per definizione. Una volta eseguito il comando per la creazione del database non sarà necessario rieseguirlo, il database sarà già presente sull'istanza e rieseguire il comando produrrà solo un errore.

C'è una ambiguità nel significato di query. Sensu strictu, query significa "interrogazione", ed è una operazione di lettura (principalmente il comando SELECT), ma sensu latu query è un qualunque comando inviato al database. Quando parliamo di "nuova finestra di query" lo intendiamo in questo senso.

Vediamo di seguito i passaggi per la creazione di un nuovo database, rifacendoci al progetto Census visto in precedenza:

```
CREATE DATABASE CENSUS;
```

Una volta eseguito l'istanza avrà un database di nome Census. E' tutto qui. Census è ancora vuoto, non dispone di alcuna tabella, tanto meno di dati, ma è già pronto a essere usato. Ora dobbiamo "spostarci" dentro il database.

```
USE CENSUS;
```

Da ora in avanti la finestra di query sa di dover lavorare dentro quella particolare scatola. A questo punto creiamo la nostra prima tabella:

```
CREATE TABLE PERSON
(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(100),
    SURNAME VARCHAR(100),
    CITY VARCHAR(100),
    ADDRESS VARCHAR(100),
    DATEOFBIRTH DATE,
    JOB VARCHAR(100),
    SALARY NUMERIC(7,2),
    GENDER VARCHAR(1),
);
```

E' abbastanza immediato. La tabella Person sarà composta di righe di nove campi ciascuna, contenenti nome, cognome, città, indirizzo, data di nascita, lavoro, salario e genere. Il salario sarà un valore numerico lungo sette cifre, di cui due dopo la virgola (7,2). Questo ci permette una grande precisione nello specificare la "struttura" di un numero. Disponiamo di un tipo dedicato per le date (come visto prima), e in realtà anche per tempo e data-tempo, ma dobbiamo spiegare almeno quella prima riga:

```
ID INT PRIMARY KEY AUTO_INCREMENT
```

Non è formalmente obbligatorio, ma tutte le tabelle dovrebbero disporre di una chiave primaria. Una chiave primaria è definita come "campo o insiemi di campi che identificano in maniera univoca una riga". Due righe nella stessa tabella non possono avere la stessa chiave primaria. In questo caso la chiave primaria è un intero auto_increment, vale a dire, un numero progressivo che viene inserito dal sistema (per quanto si possa inserirlo anche noi). E' una scelta estremamente comune ma non universale.

Quali altri campi avrei potuto usare per distinguere fra di loro le righe, vale a dire le persone? Il codice fiscale, ad esempio. L'email di registrazione. E così via... Questo problema viene detto problema di scelta della chiave primaria fra più chiavi candidate. Per praticità, noi useremo sempre e solo id numerici auto_increment, che è poi la soluzione preferita dall'industria. Avremo quindi la persona 1,2,3... nella tabella Person.

Notiamo che nessuno ci impedirà di avere lo stesso id in un'altra tabella. Le chiavi primarie sono uniche all'interno della tabella, ma possiamo trovare lo stesso campo con lo stesso nome e lo stesso valore in altre tabelle senza che questo crei problemi. Creata la tabella, procediamo a inserire dei dati al suo interno.

10.6 Manipolare i dati: inserimento, modifica, cancellazione

Scrivere nel file era abbastanza semplice, ma SQL pretende una maggiore precisione per garantirci i suoi servizi. Andiamo a inserire dei dati nella tabella Person utilizzando il comando SQL apposito, INSERT:

```
INSERT INTO PERSON
  (NAME,SURNAME,CITY,ADDRESS,DATEOFBIRTH,JOB,SALARY,GENDER)
VALUES
  ('JILL', 'RED', 'RACCOON VILLE', '2000-01-02', 'RANGER', 2500.00, 'F'),
  ('JACK', 'RED', 'RACCOON VILLE', '1998-01-02', 'RESEARCHER', 2700.00, 'M'),
  ('CHRIS', 'WHITE', 'NEW YORK', '1995-05-02', 'RANGER', 2500.00, 'M'),
  ('KEITH', 'WHITE', 'LISBON', '1980-06-02', 'RESEARCHER', 3500.00, 'M');
```

Questo comando inserirà quattro persone nel database. La sintassi è `INSERT (campi) INTO tabella (valoririga1), (valori riga2)...` Ed è una di diverse sintassi possibili. In questo caso ho specificato i campi che volevo inserire, ma avrei potuto usare la sintassi `INSERT INTO table (valori riga 1),(...),(...)`. In questo caso avrei dovuto specificare tutti i campi, mentre con la prima forma posso definire solo i campi che voglio inserire, e questo è molto comodo come quando, in questo caso, abbiamo un id auto_increment.

I campi non specificati (id, in questo caso) prendono il valore di default. Il valore di default è tipicamente "NULL", ma il valore di default di un id auto_increment è un valore calcolato, un progressivo. In questo modo il database crea per noi un identificativo senza che noi si abbia bisogno di specificarlo esplicitamente.

Ci accorgiamo di avere sbagliato. Il signor Keith è di Siviglia (Seville) invece che di Lisbona, e il suo anno di nascita è il 1982. Possiamo rimediare in questo modo:

```
UPDATE Person SET city='Seville', dateofbirth = '1982-06-02' WHERE id =4;
```

Questa è una operazione di modifica. Il where è un predicato (una condizione) che serve a definire quali righe verranno modificate. In questo caso verrà modificata una sola riga, quella con id=4 (ipotizziamo che il signor Keith White abbia id=4, ma una scorsa alla tabella ci darà la risposta finale).

Se volessimo cancellare tutte le persone di Raccoon Ville, avremmo una sintassi simile:

```
DELETE FROM PERSON WHERE CITY = 'RACCOON VILLE';
```

Ma non è detto che MySQL ce lo permetterebbe. Le operazioni di cancellazione, come tutto il resto, sono definitive e di conseguenza si rischia di uccidere una bella porzione dei nostri dati con una istruzione affrettata o scritta male. MySQL in generale parte in "safe mode", e vi permette di cancellare solo con prediciati che contengano la chiave primaria, di modo da indicare precisamente chi volete rimuovere.

E' possibile, ma sconsigliato, disabilitare il safe mode. Una soluzione per fare quello che volevamo è la seguente:

```
DELETE FROM PERSON WHERE ID in (1,2);
```

Queste operazioni sono abbastanza banali e meccaniche e sono spesso automatizzate graficamente (è possibile far tutto tramite l'interfaccia tabellare di MySQL, senza una riga di codice), ma dovremo scriverle in Java per "comunicare" col DB, quindi bisogna imparare la sintassi. Una tipologia di operazione nettamente meno banale è l'interrogazione di un database.

10.7 Interrogazioni su una tabella

Il comando per le interrogazioni in SQL si chiama SELECT ed è uno strumento che meriterebbe un libro a parte.

Il comando SELECT produce risultati selezionando righe e colonne a partire dalle tabelle del database, ma non solo. E' anche in grado di produrre risultati a partire da altre SELECT e di produrre risultati calcolati.

Cominciamo con una serie di esempi di complessità crescente:

```
SELECT * FROM PERSON;
```

Seleziona tutti i campi (*, "ALL") dalla tabella Person, prendendo tutte le righe.

```
SELECT NAME,SURNAME FROM PERSON;
```

Seleziona solo nome e cognome (solo due colonne) dalla tabella Person, prendendo tutte le righe. Questa viene detta "proiezione".

```
SELECT * FROM PERSON WHERE GENDER = 'F';
```

Seleziona tutte le colonne dalla tabella Person, ma mantenendo solo le righe con gender = 'F'. gender = 'F' è un predicato, vale a dire una condizione, e serve per l'operazione di filter. In gergo questa viene detta selezione. Possiamo combinare le cose:

```
SELECT NAME,SURNAME WHERE GENDER = 'F';
```

Proietta nome e cognome per le righe della tabella Person con gender = 'F';

Valgono i consueti connettori logici:

```
SELECT NAME,SURNAME FROM PERSON WHERE GENDER='F' AND CITY='RACCOON VILLE';
SELECT NAME,SURNAME FROM PERSON WHERE GENDER='F' OR CITY='RACCOON VILLE';
SELECT NAME,SURNAME FROM PERSON WHERE GENDER='F' OR NOT CITY='RACCOON VILLE';
```

SQL dispone di una sintassi e di un lessico più simili all'inglese, essendo pensato per essere facile da leggere anche da parte dei non specialisti. Dispone anche di una serie di operatori specializzati:

```
SELECT NAME FROM PERSON WHERE CITY IN ('SEVILLE', 'LONDON');
```

equivalente a SELECT name FROM Person WHERE city = 'Seville' or city = 'London';

```
SELECT NAME FROM PERSON WHERE YEAR(DATEOFBIRTH) BETWEEN 1980 AND 1990
```

equivalente a SELECT name from Person WHERE YEAR(dateofbirth)>=1980 and YEAR(dateofbirth)<=1990;

YEAR è una delle tante funzioni predefinite di MySQL, e ricava l'anno a partire da una data. Capiterà lavorando su progetti reali di avere un campo con lo stesso nome di una funzione o di una parola chiave, e di fare confusione.

Sempre restando in tema date, queste sono first class citizens nel sistema dei tipi di MySQL, e possiamo usarle con i consueti operatori di confronto:

```
SELECT NAME FROM PERSON WHERE DATEOFBIRTH > '1980-01-01';
```

Notiamo che abbiamo proiettato name, ma selezionato per dateofbirth. Questo non è insolito. Magari vorremo visualizzare solo alcuni campi come risultato della query, ma escludere delle righe sulla base di campi non visibili.

Possiamo anche eseguire ricerche approssimate:

```
SELECT * FROM PERSON WHERE NAME LIKE 'N%';
```

L'operatore LIKE permette di eseguire ricerche approssimate sulle stringhe secondo una sintassi che fa uso di simboli speciali. In questo caso stiamo cercando "tutti i nomi che cominciano con N, seguiti da qualunque altra sequenza di caratteri, potenzialmente anche nulla". Troveremo tutte le persone di nome N, Nino, Nina, Nathan, NonNeHoIdea... Tutti gli altri verranno esclusi. Per approfondire consigliamo questo link: https://www.w3schools.com/sql/sql_like.asp

Notiamo che possiamo proiettare anche dei campi derivati, vale a dire dei calcoli:

```
SELECT NAME, SURNAME, YEAR(NOW()) - YEAR(DATEOFBIRTH) FROM PERSON;
```

Questa query produrrà tre colonne, la cui terza sarà "anonima" e riceverà un nome assegnato automaticamente da MySQL. La funzione NOW() restituisce il momento attuale (un valore di tipo datetime) e viene passato alla funzione YEAR(), per ricavarne solo l'anno, al quale viene sottratto l'anno di nascita della persona che stiamo trattando (formalmente, il valore del campo dateofbirth della riga della tabella Person su cui stiamo lavorando), per ottenere una età approssimata, supponendo che abbia compiuto gli anni.

Per evitare di avere colonne con nomi astrusi possiamo rinominare le colonne (memorizzate o calcolate) usando l'operatore "AS" (alias):

```
SELECT NAME, SURNAME, YEAR(NOW()) - YEAR(DATEOFBIRTH) AS AGE FROM PERSON;
```

La colonna age viene calcolata di volta in volta, non memorizzata. Potremmo vederla come una funzione anonima, o un "metodo". Il processo di aliasing, come abbiamo detto, non riguarda per forza solo le colonne calcolate:

```
SELECT NAME AS NOME FROM PERSON;
```

Siamo anche in grado di ordinare i dati e di limitare il numero di righe restituite. Vediamo un esempio reale:

```
SELECT CONCAT(NAME, ' ', SURNAME) AS CITIZEN, YEAR(NOW()) - YEAR(DATEOFBIRTH) AS AGE FROM PERSON WHERE CITY = 'RACCOON  
VILLE'  
ORDER BY YEAR(NOW()) - YEAR(DATEOFBIRTH) DESC LIMIT 10;
```

La clausola ORDER permette di ordinare per una o più colonne (in questo caso, per una colonna derivata, l'età), in questo caso in ordine decrescente (opzione DESC), e riporta solo i primi 10 risultati. In sostanza, abbiamo stampato nominativo ed età dei dieci cittadini più anziani di Raccoon Ville. A scopo dimostrativo, abbiamo anche fuso nome e cognome in un campo unico, separato da uno spazio in mezzo, di nome "citizen". Un'altra applicazione immediata:

```
SELECT NAME, SURNAME, NATIONALITY FROM RUNNINGCONTEST WHERE TITLE='MARATHON' ORDER BY TRACKTIME LIMIT 3;
```

In mancanza dell'opzione DESC l'ordinamento avviene dal basso verso l'alto, quindi prenderemo i tre atleti col tempo più basso ad aver partecipato alla maratona, vale a dire gli atleti sul podio.

In Java useremo quasi sempre la forma SELECT *, per ricavare tutti i campi, ma imposteremo spesso dei filtri per caricare dal database solo i dati che ci interessano, e tipicamente vorremo anche specificare un ordine. Possiamo ordinare e filtrare anche in Java, ma è meno efficiente che ricevere i dati già manipolati dal DB.

10.8 Raggruppamento (GROUP BY) e funzioni di gruppo

Abbiamo visto come SQL offra funzioni equivalenti a map e filter, ma non ancora a reduce. Una volta selezionate m righe a partire da n (con $m \leq n$), possiamo mapparle a un insieme diverso (ad esempio aggiungendo o togliendo colonne), ma non abbiamo ancora visto come ricavare risultati che riguardino più di una riga. Diciamo che la nostra visibilità è "sulla singola riga". YEAR(), NOW() e CONCAT(), così come i normali operatori (+, -, ecc..) sono "scalari", vale a dire che prendono in ingresso i campi di una riga.

SQL offre anche delle funzioni di gruppo che eseguono le riduzioni più comuni. Vediamo qualche esempio:

```
SELECT SUM(SALARY) FROM PERSON WHERE CITY = 'RACCOON VILLE';
-- SOMMA DEI SALARI DI RACCOON VILLE, IN SOSTANZA IL SUO PIL.
SELECT AVG(SALARY) FROM PERSON WHERE JOB = 'RESEARCHER';
-- SALARIO MEDIO DI UN RICERCATORE.
SELECT COUNT(*) FROM PERSON;
-- NUMERO DELLE PERSONE IN ARCHIVIO.
SELECT MAX(SALARY), MIN(SALARY) FROM PERSON WHERE YEAR(DATEOFBIRTH) = 1980;
-- SALARIO MASSIMO E MINIMO DEI NATI NEL 1980.
```

SUM, AVG, COUNT, MAX e MIN sono funzioni di gruppo, lavorando non su una riga ma su un gruppo di righe. Entrano insieme di righe ed escono dei valori singoli, che è il funzionamento di base del reduce.

Possiamo usare queste funzioni anche come parte di una espressione numerica:

```
SELECT MAX(SALARY)-MIN(SALARY) FROM PERSON WHERE YEAR(DATEOFBIRTH) = 1980;
-- SALARY GAP FRA IL PIÙ PAGATO E IL MENO PAGATO DEI NATI NEL 1980.
-- ANCHE IN QUESTO CASO CI CONVIENE USARE DEGLI ALIAS:
SELECT MAX(SALARY)-MIN(SALARY) AS DELTA FROM PERSON WHERE YEAR(DATEOFBIRTH) = 1980;
```

Analizziamo quest'ultimo caso: abbiamo operato una selezione (solo i nati nel 1980), e sulle righe selezionate abbiamo eseguito due funzioni di gruppo (max e min). Questo ci ha restituito una sola riga a partire dalle n righe dei nati nel 1980. La funzione ha ridotto n righe a una sola riga, con due valori (immaginiamo il massimo sia 2.500, il minimo 1.200):

Max	Min
2.500	1.200

Su questa riga abbia poi eseguito una trasformazione aritmetica scalare (una proiezione sui dati di una sola riga):

Delta
1.300

In questo caso abbiamo lavorato su un gruppo. Il gruppo era composto da una selezione di righe (tutti i nati nel 1980), ed è poi stato compresso in una sola per fornire alla fine un unico risultato. È una operazione di riduzione, o matematicamente parlando una funzione che trasforma un insieme di righe in un valore.

Ora, supponiamo di volere lo stipendio medio per professione. Una possibilità è la seguente:

```
SELECT AVG(SALARY) AS AVERAGE FROM PERSON WHERE JOB = 'RESEARCHER';
SELECT AVG(SALARY) AS AVERAGE FROM PERSON WHERE JOB = 'CLERK';
...
...
```

Ma una soluzione perdente. Dobbiamo conoscere a priori tutte le professioni da tracciare, ed eseguire n queries, ciascuna delle quali restituirà una sola riga.

La soluzione corretta è quella di ordinare a SQL di raggruppare in base ai valori presenti nella tabella, e di applicare a ogni gruppo una o più funzioni di gruppo. Questa funzione si ottiene tramite la clausola GROUP BY. Vediamo la soluzione corretta e spieghiamo cosa avviene:

```
SELECT JOB, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY JOB;
```

GROUP BY job significa "dividi la tabella in gruppi di righe con lo stesso valore per la colonna job". Immaginiamo di avere 3 Researcher, 2 Clerk e un Officer. Avremo tre gruppi, di tre, due e una riga rispettivamente. Questo processo viene anche detto partizionamento. Su ognuno di questi gruppi viene applicato il meccanismo di riduzione visto in precedenza. Al primo gruppo (gruppo job = 'Researcher') viene applicata la funzione avg(salary), e quindi calcoliamo la media dei salary, e produciamo una riga che conterrà due colonne: job (la chiave di raggruppamento), e il risultato della funzione, con alias average. Il gruppo è collasato a una riga, ed è la regola generale: ogni gruppo viene ridotto a una riga di risultati (reduce).

Lo stesso avverrà per gli altri due gruppi. Il risultato potrebbe essere il seguente:

Job	Average
Researcher	2500
Clerk	1700
Officer	2000

Il gruppo Officer aveva una sola riga, ma poco male, conta comunque come gruppo.
E se volessi verificare gli stipendi differenziandoli per professione e genere?

```
SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE FROM PERSON GROUP BY JOB, GENDER
```

Questo è un raggruppamento doppio. Supponendo di avere gli stessi dati di prima, avremo non n, ma n x m gruppi, dove n sono i valori possibili per il primo campo di raggruppamento (job) ed m i valori possibili per il secondo (gender). Avendo tre job possibili e due gender, avremo sei gruppi:

Researcher	M
Researcher	F
Clerk	M
Clerk	F
Officer	M
Officer	F

Alcuni si saranno già accorti di una cosa: abbiamo un solo Officer, quindi uno dei gruppi sarà vuoto. Nel caso in cui l'Office dovesse essere una donna, il gruppo Officer M sarà vuoto e NON comparirà nel risultato. Questo ci insegna una nuova regola: i gruppi vuoti non producono righe di risultato. Ipotizzando di avere uomini e donne nelle altre categorie, ci troveremo ad avere questi risultati:

Job	Gender	Average
Researcher	M	x1
Researcher	F	x2
Clerk	M	x3
Clerk	F	x4
Officer	F	x5

Dove le varie x sono i risultati del calcolo eseguito sul gruppo. Come visto sopra possiamo usare diverse funzioni di gruppo anche in questo caso:

```
SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE, MAX(SALARY) AS MAXSALARY, MIN(YEAR(DATEOFBIRTH)) AS OLDEST FROM PERSON  
GROUP BY JOB, GENDER
```

Ora, supponiamo di tornare alla query precedente, volendo escludere chi guadagna meno di 1000 euro dal calcolo. Ipotizzando che ci sia un ricercatore senza borsa di studio, avremo gruppi formati da due, due e una riga (dovremo escludere il ricercatore che prende solo un rimborso, magari di 500 euro). Lo possiamo ottenere col consueto processo di selezione:

```
SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE FROM PERSON WHERE SALARY>=1000 GROUP BY JOB, GENDER
```

L'ordine di esecuzione è selezione - partizionamento - riduzione - proiezione. Prima vengono eliminate le righe non desiderate e successivamente si creano i gruppi, si eseguono i calcoli e si proiettano i risultati desiderati. Notiamo che si tratta sempre di selezione sulla riga, almeno per ora, ma se rimuoviamo tutte le righe rimuoviamo anche il gruppo ("un gruppo senza righe non produce risultato").
Un'altra regola da ricordare è che, se usiamo le funzioni di gruppo, possiamo proiettare solo i campi per cui abbiamo raggruppato e funzioni di gruppo o calcoli su questi:

```
SELECT C1, C2, AVG(C3), SUM(C4), AVG(C1+C2), MIN(C3)+MAX(C2) FROM P  
GROUP BY C1, C2;
```

E' corretto. Invece:

```
SELECT C1, C2,C3, AVG(C3), SUM(C4), AVG(C1+C2), MIN(C3)+MAX(C2) FROM P  
GROUP BY C1, C2;
```

E' errato. MySQL lo accetterà ma altri DBMS, più rigorosi, non lo permettono. La ragione è semplice: che senso ha proiettare un campo che potrebbe cambiare da una riga all'altra dello stesso gruppo? Vediamo un esempio:

```
SELECT JOB, GENDER, AVERAGE(SALARY) AS AVERAGE FROM PERSON GROUP BY JOB;
```

Ogni gruppo deve produrre una riga, ma noi stiamo raggruppando solo per job. I gruppi saranno 3: Researcher, Clerk e Officer. In Researcher e Clerk abbiamo sia uomini e donne, ma l'intero gruppo deve essere compreso in una sola riga. Quindi, quale valore avrà gender, M o F? MySQL prenderà il primo valore disponibile trovato, ma ha poco senso. In generale, se voglio il valore di gender, devo raggruppare per gender, e quindi spezzare il gruppo job in due sottogruppi, che produrranno due righe. Ricordiamoci la regola "così in alto, così in basso". Se ho dei campi "liberi" nella SELECT e c'è un GROUP BY, devo averli anche nella GROUP BY. Se ho SELECT c1, sum(c2) from t, devo avere group by c1. Non c2, perché c2 non compare "libero". Non potrei neanche usare una funzione scalare su c2, perché c2 sarebbe fuori da una funzione di gruppo.

Resta un ultimo punto da vedere prima di passare avanti.

Supponiamo di avere molti più dati di quelli presi per esempio adesso. Magari tutti i dati nazionali italiani, quindi circa sessanta milioni di righe, o magari anche solo trenta milioni non considerando bambini e persone fuori dal mercato del lavoro.

Ipotizziamo di voler calcolare il salario medio per città. E' abbastanza semplice:

```
SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY;
```

Ma l'Italia è la nazione delle tante città. Potremmo non essere interessati a nuclei troppo piccoli, soprattutto considerando che magari le persone indicate lavorano poi in una città vicina. Supponiamo di voler considerare solo le città con almeno 30.000 abitanti (che è già una buona soglia, escludendo anche alcuni capoluoghi di provincia). Notiamo che noi non possiamo desumere dalla singola riga quanti abitanti abbiamo in una data città. E' una proprietà del gruppo (una funzione di gruppo, la funzione count), non del singolo. Dovremmo tenere quei gruppi (quelle città) per cui COUNT(*) ("conta le righe") è ≥ 30.000 .

La maniera corretta di farlo è tramite l'utilizzo della clausola HAVING:

```
SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY  
HAVING COUNT(*)>30000;
```

In having abbiamo un predicato applicato a un gruppo, non alla singola riga, ed è il punto ideale per imporre condizioni (predicati) su funzioni di gruppo. In questo caso, abbiamo escluso di gruppi con meno di 30.000 righe. Ma potremmo fare di meglio:

```
SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY  
HAVING COUNT(*)>30000 AND AVG(SALARY)<1000;
```

Abbiamo escluso anche le città con un reddito medio troppo basso (possiamo ipotizzare un problema nel dato per una città di 30.000 persone con salario medio inferiore a 1000 euro, ma accettiamolo per amor di esempio).

Per finire, possiamo ordinare il risultato:

```
SELECT CITY, AVG(SALARY) AS AVERAGE FROM PERSON GROUP BY CITY  
HAVING COUNT(*)>30000 AND AVG(SALARY)<1000 ORDER BY AVG(SALARY) DESC;
```

In queste query tipicamente Milano risulta prima seguita da Roma, ma varrebbe la pena di rapportare il salario al costo della vita. Volendolo fare, disponendo di questi dati, vedremo che la qualità della vita non rispetta per forza questa distribuzione.

10.9 Relazioni e query su più tabelle: il concetto di JOIN

Fino ad ora abbiamo lavorato su singole tabelle. Ogni riga ha rappresentato una entità singola, lo "stato di un oggetto", nel nostro caso una persona. Adesso proviamo ad ampliare il campo. Supponiamo di avere, oltre alla nostra tabella precedente, una tabella city.

City

Name	AverageRent	BasicExpenses
Milano	800	200
Roma	650	150

Con name chiave primaria (pessima idea, ma lo vedremo dopo).

Ora abbiamo i dati relativi al costo della vita per città, e i dati degli stipendi, ma sono in due tabelle diverse. Questo è desiderabile, perché altrimenti dovremmo riportare i dati della città (gli affitti e le spese) in ogni riga di Person, creando una enorme ridondanza.

Supponiamo di voler stimare quanto risparmia ogni cittadino partendo dai costi della sua città e dal suo salario. Abbiamo bisogno di due tabelle. Andiamo per gradi:

```
SELECT SURNAME, CITY, SALARY - AVERAGERENT - BASICEXPENSES FROM PERSON, CITY;
```

Siamo riusciti a prendere due tabelle, specificandole separate con una virgola nel from, ma il risultato è sgradevole. E' vero che abbiamo preso le due tabelle. E' vero che abbiamo tutti i campi necessari ma tutte le persone sono state associate a tutte le città. Abbiamo ottenuto un mischiume per cui tutti sono stati associati a ogni città: per la precisione, abbiamo ottenuto quello che viene detto in gergo un prodotto cartesiano, o "cross join".

In pratica, avremo un risultato di questo tipo, ipotizzando di avere solo tre persone e due città: P1 C1

P1 C2
P2 C1
P2 C2
P3 C1
P3 C2

Dove Pn e Cn sono righe. Questa sintassi produce "super righe" unendo una riga di Person a una riga di City.

Se la tabella person aveva 30.000.000 di righe, e la tabella city ne aveva 10.000, il risultato sarà di 300.000.000.000 di righe. Decisamente scomodo da consultare, oltre al fatto che letteralmente 9999 righe su 10.000 sono spazzatura, visto che associano una persona con una città che non le compete.

Dobbiamo collegare le righe correttamente. Questo si ottiene tramite un predicato, una condizione anche abbastanza semplice:

```
SELECT PERSON.NAME, PERSON.SURNAME, PERSON.CITY, PERSON.SALARY - PERSON.AVERAGERENT - PERSON.BASICEXPENSES FROM PERSON, CITY WHERE PERSON.CITY = CITY.NAME;
```

Questa sintassi richiede una maggiore attenzione. Prima di tutto, abbiamo voluto specificare da quale delle due tabelle stiamo prendendo i campi. Non ci limitiamo più a dire "name", perché il campo name è presente sia in city che in person, e MySQL non saprebbe quale dei due proiettare. Specifico che voglio proiettare (stampare) il campo della tabella person, di modo da evitare l'errore "ambiguous field name" in fase di esecuzione. Per gli altri campi non ci sono problemi di ambiguità, ma è buona prassi specificare da dove prendo i dati quando uso più di una tabella (e tipicamente una query "seria" tocca tante tabelle...). Da ora in avanti prenderemo come abitudine quella di indicare i campi proiettati come nomeTabella.nomecampo.

La seconda novità è quel predicato nel where. E' un predicato che confronta due campi, la colonna name in city (chiave primaria) e la colonna city in person. Stiamo dicendo "tieni solo le righe per cui questa condizione è vera", e siccome le nostre righe sono nella forma PkCj (righe costituite da una persona e una città) gli stiamo dicendo di tenere solo determinati accoppiamenti, quelli che hanno un senso.

Torniamo all'esempio sopra. Supponiamo che P1 abiti in C2 (p1.city = 'C2'), P2 abiti in C2 e P3 abiti in C1.

Possibili combinazioni Predicato

P1(C2) C1	c2 = c1 -> falso
P1(C2) C2	c2 = c2 -> vero
P2(C2) C1	c2 = c1 -> falso
P2(C2) C2	c2 = c2 -> vero
P3(C1) C1	c1 = c1 -> vero
P3(C1) C2	c1 = c2 -> falso

Quindi terremo come risultato solo le righe P1 unita a C2, P2 unita a C1 e P3 unita a C1. Un totale di tre righe, non sei, per cui vale la relazione "la persona P vive nella città C".

A questo applichiamo le nostre consuete trasformazioni e calcoli, e possiamo calcolare per ogni persona il risparmio teorico, prendendo il salario (dalla riga di Person) e sottraendogli affitto e spese (dalla tabella city).

La condizione person.city = city.name prende il nome di predicato di join. Il predicato di join può essere una qualunque condizione, e può anche mancare (in mancanza di predicato di join abbiamo il prodotto cartesiano delle due tabelle). Una operazione di JOIN propriamente detta,

comunque, è definita come l'applicazione di un predicato al prodotto cartesiano, o "una selezione sul prodotto cartesiano". Informalmente, significa capire quali combinazioni di righe tenere e quali scartare. La sua forma generale è questa:

```
SELECT * FROM T1, T2, T3 WHERE COND
```

Il predicato di Join in questo caso è il modo di SQL per esprimere a una relazione fra due entità. In Java avremmo avuto la entity Person e la entity City. Una città può ospitare n persone, mentre una persona può risiedere in una sola città.

Questo tipo di rapporto in Java prende il nome di associazione, e sappiamo darle una cardinalità: una città - n persone. In SQL questo concetto prende il nome di relazione fra tabelle, e il predicato di join ci dirà per quali righe vale la relazione "vive a" fra Person e City.

Ricapitolando:

- fra Person e City c'è una relazione, per ora implicita, che potremmo leggere "vive in" (andando da Person a City) oppure "ospita", andando da City a Person.
- potremmo anche scriverla come "una persona vive in una città" o "una città ospita n persone"
- in ambo i casi la relazione è 1-n. A una persona associeremo una città, mentre a una città più persone.
- ricostruiamo la relazione mantenendo unite solo le righe per cui questa è valida, vale a dire le righe per cui person.city = city.name

In questo caso la relazione è stata ricavata per logica. Le due tabelle non erano al corrente di essere in relazione, ma in SQL è preferibile dichiarare le relazioni che prevediamo di usare, perché questo ci permetterà di ottimizzare le ricerche (velocizzarle) e di aggiungere dei controlli ulteriori.

Prima però è opportuno migliorare le nostre tabelle. Cominciamo da City: City ha name come chiave primaria. Per quanto questo possa non essere un problema, ci espone al rischio di una ominimia e quindi di ripetizione della chiave primaria (vietatissimo). Non potremo avere due città con lo stesso nome, e questo può andare bene a livello statale, ma potrebbe non andare bene a livello globale (pensiamo alle varie Alessandria o Springfield sparse per il mondo).

Conviene rifattorizzare City aggiungendo un ID per identificare univocamente la città. Quindi potremmo avere :

1 Milano

2 Roma

3 Napoli

...

Il tracciato record della città sarà (id, name, averagerent, basicexpenses).

A questo punto ragioniamo su Person. Se in person mettiamo il nome della città non avremo un riferimento alla vera chiave primaria. Posto il fatto che il signor James vive a "Springfield" noi potremmo educatamente chiedere: quale? E' un problema che gli appassionati di un noto cartone conoscono bene.

Occorre che la Person conosca l'id della città, non il nome. Il nome non è univoco, l'id sì. Quindi il tracciato di Person potrebbe essere il seguente:

```
CREATE TABLE PERSON
(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    NAME VARCHAR(100),
    SURNAME VARCHAR(100),
    CITYID INT,
    ADDRESS VARCHAR(100),
    DATEOFBIRTH DATE,
    JOB VARCHAR(100),
    SALARY NUMERIC(7,2),
    GENDER VARCHAR(1),
);

```

Non sappiamo più il nome della città in cui viviamo, ma possiamo ricavarlo collegando la tabella Person a City, tramite il suo id. La colonna cityid viene detta chiave esterna da Person verso City, ed esplicita la relazione "vive in".

Il join ora ha questa forma:

```
SELECT * FROM PERSON, CITY WHERE PERSON.CITYID = CITY.ID;
```

Notiamo che cityid si può ripetere! Cityid NON è chiave primaria in Person, altrimenti ogni Person avrebbe una città per sè. City.id è chiave primaria in city, ma cityid è un campo di Person, e non è chiave primaria.

Un esempio:

ID Name Surname CityID

1 John Watson 1

2 Jill Watson 1

3 James Wick 2

Chiave esterna è un modo carino di dire "riferimento". cityid è un campo che fa riferimento a una riga in un'altra tabella.

A questo punto possiamo enunciare una procedura generale per le relazioni 1-n. Poste due tabella legate da un rapporto 1-n (una riga di t1 è legata a n righe di t2, una riga di t2 è legata a una sola riga di t1), ci procede in questo modo:

- la tabella lato uno (city in questo caso) deve avere una chiave primaria
- la tabella lato n (person in questo caso) deve avere una chiave esterna, dello stesso tipo della chiave primaria nella tabella lato 1 (int in questo caso), come riferimento alla riga a cui è collegata
- Il predicato di join a questo punto è la condizione t1.chiaveprimaria = t2.chiaveesterna. In forma abbreviata, t1.pk = t2.fk (fk, foreign key).

Un altro esempio, per chiarire: una persona può disporre di diverse auto. Il rapporto è una Person - n Car. La tabella Car potrebbe avere come tracciato (id, platenumber, model, cost, ownerid) e il join fra le tabelle Person e Car per ricavare il proprietario:

```
SELECT * FROM PERSON, CAR WHERE PERSON.ID = CAR.OWNERID;
```

Questo nel caso in cui non ci fossero multiproprietà o auto di proprietà aziendale. Questa query è piuttosto utile per comminare multe e altre sanzioni, per inciso. Il campo ownerid esprime, tramite riferimento, un rapporto fra person (1) e car (n).

Le relazioni fra entità possono essere molte. Supponiamo ad esempio che un'auto sia di proprietà di una sola persona, ma che sia stata revisionata da un'altra. Allora avremmo qualcosa di questo tipo in Car: (id, platenumber, model, cost, ownerid, revisorid). A questo punto avremmo due modi di unire Person e Car: per trovare il proprietario, come prima, o per trovare il revisionatore:

```
SELECT * FROM PERSON, CAR WHERE PERSON.ID = CAR.REVISORID;
```

Ogni chiave esterna rappresenta quindi una relazione fra la tabella in cui si trova e un'altra, o potenzialmente anche la tabella stessa (in questo caso parliamo di self join, ma vedremo in seguito).

10.10 L'integrità referenziale e il problema degli orfani

E se avessimo una riga di questo tipo in Person?

```
id name surname cityid
1 A1   B1      -5
2 A2   B2      NULL
```

Visto che gli id tipicamente partono da 1, direi che nè A1 nè A2 sanno dove vivono. I due cityid dovrebbero riferirsi a una riga di City, ma probabilmente quelle righe non esisteranno.

In questo caso diremo che le righe 1 e 2 di Person sono orfane rispetto alla relazione "vive in" con City. "Orfane" intese come "senza padre", o "senza un elemento di riferimento". Possiamo usare le informazioni di quella riga, ma non possiamo collegarla in maniera sensata a nessuna città. Lo stesso accadrebbe se avessimo cityid = 1, e la riga corrispondente mancasse in city, o fosse stata modificata cambiando id.

Questo ci pone di fronte al problema della politica di gestione degli orfani. Vogliamo accettarli o meno? E possiamo impedirli?

La risposta alla prima domanda è "dipende". Dobbiamo decidere se la relazione fra le tabelle è semanticamente fondamentale per il nostro lavoro (e quindi renderebbe inutile il programma) o se è opzionale. In questo caso dobbiamo decidere se vogliamo sapere per forza la città di appartenenza di una persona.

La risposta alla seconda domanda è "sì, ma c'è un prezzo".

La prassi comune è di evitare gli orfani. Questo si ottiene con due step:

- si enunciano in maniera esplicita le relazioni, in fase di creazione delle tabelle
- si impostano delle politiche di gestione degli orfani quando la tabella "padre" o "riferimento" cambia

Vediamo in pratica. Supponiamo di volere assolutamente conoscere revisore e proprietario di un'auto prima di venderla. Lo otteniamo in questo modo:

```
CREATE TABLE CAR
(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    PLATENUMBER VARCHAR(500),
    MODEL VARCHAR(500),
    COST NUMERIC(10,2),
    OWNERID INT,
    REVISORID INT,
    FOREIGN KEY(OWNERID) REFERENCES PERSON(ID)
    ON UPDATE CASCADE
    ON DELETE RESTRICT,
    FOREIGN KEY(REVISORID) REFERENCES PERSON(ID)
    ON UPDATE CASCADE
    ON DELETE RESTRICT
);
```

Le ultime righe specificano in maniera esplicita le foreign key, che prima veniva usate come tali ma non dichiarate, in sostanza dichiarando la relazione. La tabella Car sta dichiarando una relazione prima fra il proprio campo ownerid e il campo id di person e poi fra il proprio campo revisorid e il campo id di person. La riga di Car è legata a due righe diverse di Person, come abbiamo visto prima, con due rapporti diversi.

Anche solo questo basterebbe a ottenere un primo risultato: non potremmo inserire valori nulli in ownerid e revisorid, in quanto le chiavi esterne (come del resto le primarie) non possono essere nulle. In fase di creazione la riga dovrebbe sapermi dire di chi è l'auto e chi l'ha revisionata.

Le due righe successive (on update cascade e on delete restrict) specificano una politica di gestione del rapporto. Cosa succede se qualcuno cambia l'id di una persona che possiede tre macchine, portandolo da 3 a 6?

Si tratta di una operazione di update. Sto aggiornando la tabella padre, e per gli aggiornamenti abbiamo optato di effettuare un "cascade", vale a dire una propagazione, verso l'entità figlia. Le auto della persona 3, che è diventata 6, adesso hanno ownerid = 6 anche loro. Il cambiamento è stato eseguito automaticamente dal DBMS.

E se provassi a cancellare la persona 3? In questo caso la politica che abbiamo scelto era "restrict", come accade tipicamente in questi casi. Restrict significa che non posso cancellare la persona finché ci saranno auto di sua proprietà, o anche auto che ha revisionato. Se voglio uccidere il "padre" devo prima cancellare tutti i suoi "figli". Se voglio eliminare una riga della tabella lato 1 devo cancellare tutte le righe delle tabelle lato 2 a cui è legata.

In generale, con restrict, non ho l'autorizzazione a cancellare una riga finché esistono righe legate a lei da quella relazione.

E se avessimo scelto on delete cascade per la chiave esterna ownerid? Cancellando la persona, avremmo cancellato tutte le auto a lei collegate - la cancellazione si sarebbe propagata ai "figli". Potrebbe avere senso per un cliente che desidera cancellare il proprio storico. Invece potrebbe non volerlo e mantenere restrict per il revisore (in generale, il revisore non dovrebbe poter cancellare i propri dati o i dati del proprio lavoro o dei propri errori, quindi è più corretto restrict).

Questa tecnica viene detta "integrità referenziale", e potete ricordarla agevolmente come "niente orfani". Non è supportata da tutti i DBMS, ma è praticamente universale in tutti quelli di livello enterprise. Il "prezzo" è che spesso ci troveremo ad avere errori nelle operazioni di modifica o di cancellazione perché ci eravamo scordati dell'integrità referenziale, e non avevamo agito correttamente prima.

Sarebbe anche opportuno conoscere le relazioni prima di creare le tabelle, per poterle dichiarare in fase di creazione, ma non è troppo arduo modificarle in seguito.

10.11 Il rapporto 1-1 e l'ereditarietà in MySQL. Forma implicita ed esplicita del Join e concetto di Inner Join

Le relazioni fra tabelle rispecchiano quelle fra entità (in effetti, le tabelle rappresentano lo stato di entità), ma mentre in UML abbiamo rapporti specifici e in Java abbiamo oggetti collegati in un DBMS SQL abbiamo "solo" una tipologia di rapporto: il collegamento fra righe. Vediamo come tradurre quello a cui siamo abituati in tabelle, partendo dal rapporto di ereditarietà.

Il rapporto di ereditarietà è in realtà un rapporto di specializzazione. Un oggetto di un tipo A può anche essere di un tipo B, dove B è una sottocategoria di A. Supponiamo di voler tracciare, solo per gli insegnanti, la scuola di riferimento ("scuola polo"), gli anni di anzianità e il titolo di studio, oltre alle materie per cui è abilitato.

Una prima soluzione sarebbe modificare Person, aggiungendo le colonne relative (anzianità, titolo di studio, scuola polo), ma resterebbero vuote per tutte le righe che non sono relative a degli insegnanti. Quello che vogliamo fare invece è simulare l'ereditarietà (che non esiste nei DBMS SQL standard) con un rapporto 1-1 opzionale.

La categoria più generale, la classe "padre", viene "mappata" sulla tabella Person che abbiamo visto in precedenza. A questa aggiungiamo una tabella Teacher con questa forma:

```
CREATE TABLE TEACHER
(
    ID INT PRIMARY KEY,
    SCHOOL VARCHAR(100),
    DEGREE VARCHAR(100),
    SUBJECTS VARCHAR(100),
    YEARS INT,
    FOREIGN KEY(ID) REFERENCES PERSON(ID)
    ON CASCADE UPDATE
    ON DELETE RESTRICT
);
```

La colonna id ha una duplice funzione. Internamente a Teacher è chiave primaria ma anche chiave esterna verso Person. Questo vuol dire che una riga di Teacher deve avere una riga di Person di riferimento per completarsi. Teacher in effetti non conosce il proprio nome, la propria città, la propria data di nascita. Di contro, Person non è costretto ad avere chiavi esterne, perché in effetti per Person questa relazione non è obbligatoria, ma opzionale. Una riga di Person potrebbe essere collegata a una riga di Teacher, ma potrebbe anche fare un altro lavoro, nel quel caso non troveremo l'id di quella Person nella tabella Teacher.

Una situazione tipica potrebbe essere la seguente:

Person
ID Name
1 A
2 B
3 C

Teacher
ID School
1 S1
3 S2

Perchè manca l'id 2 in Teacher? E' la stessa ragione per cui l'id di Teacher non è auto_increment. I signori A e C sono insegnanti. Il signor B si occupa di altro.

Il join in questo caso non è troppo diverso da quello visto prima, ma la chiave esterna stavolta è anche chiave primaria per la tabella "figlia":

```
SELECT * FROM PERSON, TEACHER WHERE PERSON.ID = TEACHER.ID;
```

E il join si riduce a t1.pk = t2.pk.

Ne approfittiamo per introdurre una nuova particella, l'inner join esplicito, e per capire il perchè lo definiamo "inner", e perchè in realtà i join che abbiamo usato fino a ora sono sempre stati inner.

La query di sopra si può riscrivere come

```
SELECT * FROM PERSON INNER JOIN TEACHER ON PERSON.ID = TEACHER.ID;
```

In questa forma il predicato di join viene messo nel from, non nel where, ed è considerata più elegante. La sintassi generica è:

```
SELECT * FROM T1 INNER JOIN T2 ON (PREDICATO DI JOIN)
```

Questo permette di separare i predicati di join dalle altre condizioni della query (che chiamiamo per convenzione "condizioni di dominio"). Ci permetterà anche di scegliere meglio il tipo di join, come vedremo a breve.

Il risultato è esattamente lo stesso che ottenevamo prima, ma in questa forma parliamo di inner join esplicito, mentre nella versione precedente

(quella col where) si parla di inner join implicito.

Ma cosa vuol dire inner? inner è una parola chiave che serve a indicare che dobbiamo prendere solo le righe per cui quella condizione è vera, vale a dire per cui vale la relazione. Ricordiamolo come "prendi le righe collegate e solo quelle". Infatti, applicando quella query ai dati di esempio di sopra, otteniamo questo risultato:

ID	Name	School
1	A	S1
3	C	S2

Abbiamo effettivamente ricomposto la tabella, e assegnato a ogni insegnante la sua scuola (ho omesso gli altri campi per brevità), ma dov'è il signor B? B è stato tralasciato, perché non c'è nessuna riga che gli corrisponda in Teacher. B è una Person, ma non è un Teacher, e non può risultare in una interrogazione in cui si voglia usare l'inner join,隐式或显式都可以。 Ricordiamocelo: inner join significa "prendi le righe collegate".

Ma se volessimo vedere anche B? Chiaramente non potremmo associargli una scuola, non essendo lui un insegnante, ma magari vorremo vederlo nella lista comunque. Come fare a prenderlo, e che valore avrà il campo School per lui?

Possiamo prendere anche B utilizzando un join diverso, usato tipicamente per queste eventualità, che prende il nome di LEFT JOIN. La sintassi è la seguente:

```
SELECT * FROM T1 LEFT JOIN T2 ON (PREDICATO DI JOIN)
```

E si legge "prendi le righe di t1 e collegale alle righe di t2 per cui vale la relazione espressa dal predicato di join. Se non trovi righe di t2, presenta comunque la riga di t1 e combinala con una riga di valori null per i campi di t2". La possiamo ricordare come "preserva tutte le righe di t1" o "preserva tutte le righe a sinistra", che la relazione valga oppure no.

La query di sopra avrà questo risultato:

ID	Name	School
1	A	S1
2	B	NULL
3	C	S2

La riga 2, il signor B, che non partecipa alla relazione con Teacher, è comunque presente, ed è stato associato a una riga composta di valori null. Il left join non escluderà mai le righe a sinistra, che sono tipicamente quelle della tabella "padre", ma possono anche essere quelle della tabella lato 1 in un rapporto 1-n.

Niente vieta che una riga venga esclusa da una condizione di dominio. Supponiamo di voler trovare tutte le persone con un lavoro della città con id =1. Per gli insegnanti vogliamo visualizzare anche la scuola. La soluzione sarebbe la seguente:

```
SELECT PERSON.NAME, PERSON.SURNAME, PERSON.JOB, TEACHER.SCHOOL  
FROM PERSON LEFT JOIN TEACHER ON PERSON.ID = TEACHER.ID  
WHERE PERSON.CITYID = 1;
```

Anche se il left join mantiene tutte le righe della tabella Person, molte di quelle righe verranno eliminate dalla condizione di dominio, che pretende che cityid sia uguale a 1.

Abbiamo usato cityid di proposito. Usando solo le tabelle person e teacher non abbiamo accesso al nome della città, ma se avessimo voluto lo stesso risultato per gli abitanti di Milano? Ci sarebbe servita una terza tabella:

```
SELECT PERSON.NAME, PERSON.SURNAME, PERSON.JOB, TEACHER.SCHOOL  
FROM PERSON INNER JOIN CITY ON CITY.ID = PERSON.CITYID LEFT JOIN TEACHER ON PERSON.ID = TEACHER.ID  
WHERE CITY.NAME = 'MILANO';
```

Adesso comincia a diventare più complesso. Si traduce in questo modo: "A ogni persona associa la sua città, e pretendi di avere la città o scarta la persona (inner join, solo le righe collegate). A questa combinazione aggiungi, se c'è, una riga di Teacher collegata a Person, altrimenti una riga di valori nulli".

Il risultato di questa query, prima della proiezione di nome, cognome, lavoro e scuola, è una lista di righe del tipo {P(i), C(k), T(i)}, dove P(i) è la riga di person con id i, C(k) è la riga di City con id k collegata a P(i) e T(i) è l'eventuale Teacher collegato a P(i). Notiamo che Person e Teacher hanno lo stesso id (i), essendo il join eseguito sulla chiave primaria. T(i) sarà nullo nel caso in cui la persona non sia un insegnante (campi vuoti).

Come regola generale, se abbiamo n tabelle nel nostro from avremo n-1 join (left o inner, opzionali o obbligatori) il nostro risultato sarà composto da righe divise in n parti, una da ogni tabella. Su quella maxiriga eseguiremo la proiezione per ricavare quello che ci interessa. Vedremo questo meccanismo meglio in seguito.

10.12 I rapporti 1-n

I rapporti 1-n sono quelli che in Java abbiamo chiamato variamente uso, associazione, aggregazione o composizione, e che abbiamo già visto applicare in pratica da MySQL nella prima sezione sul join.

In termini generici, a una entità (riga) di una tabella t1 corrispondono n entità (righe) di una tabella t2. Queste vengono legate tipicamente tramite l'uso di una chiave esterna, per quanto questo non sia sempre il caso, e come abbiamo visto prima due tabelle possono essere legate da più di una relazione (o rapporto che dir si voglia).

Nel caso in cui vogliamo esprimere in SQL un rapporto di associazione o aggregazione, la maniera naturale di operare è tramite integrità referenziale con le clausole on update cascade e on delete restrict. Nel caso in cui si tratti di composizione, la clausola più naturale potrebbe essere on delete cascade (gli oggetti dipendenti nel rapporto di composizione non dispongono di solito di una esistenza indipendente rispetto al contenitore).

Il rapporto fra una squadra e i suoi membri potrebbe essere un rapporto 1-n di associazione / uso / aggregazione in Java (le differenze sono flessibili), ma cancellare una squadra non dovrebbe portare alla cancellazione dei giocatori. Il rapporto fra un prodotto e le sue recensioni dovrebbe essere un rapporto 1-n di composizione - cancellare il prodotto rende insensate le righe delle recensioni, e di conseguenza potrebbe essere il caso di cancellarle contestualmente.

In termini pratici, di select e di join, non ci sono differenze:

```
SELECT PRODUCT.*, REVIEW.* FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID
```

Con product.id chiave primaria in product e productid chiave esterna di review verso product, product lato 1, review lato n. La query di sopra riporterà tutti i prodotti con recensioni, ciascuno legato alla propria recensione. Le righe avranno la forma P(i)R(k), con una riga di prodotto (il prodotto con id i) legata a n righe di review con id k diversi fra loro:

```
P R  
1 1  
1 2  
1 3  
2 1
```

Il prodotto 1 ha tre recensioni (con id 1,2 e 3 rispettivamente) mentre il prodotto 2 ne ha solo una. Un eventuale prodotto 3 non compare. Per vedere anche i prodotti senza recensioni dovremo usare la left join:

```
SELECT PRODUCT.*, REVIEW.* FROM PRODUCT LEFT JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID
```

A questo punto avremo come risultato:

```
P R  
1 1  
1 2  
1 3  
2 1  
3 NULL
```

Possiamo applicare le funzioni di gruppo anche ai join, e questo è spesso utile nei rapporti 1-n. Immaginiamo di dover calcolare la votazione media per ogni prodotto. Il punteggio (score) sarà contenuto nella recensione, mentre il nome del prodotto e il suo id saranno nella tabella prodotti. Procediamo per gradi:

```
SELECT PRODUCT.ID, PRODUCT.NAME, REVIEW.SCORE FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID;
```

Abbiamo i dati che ci servono. Ora dobbiamo ridurli a un risultato di nostro gusto:

```
SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE FROM PRODUCT INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME;
```

Volevamo mantenere product.id e product.name, di conseguenza abbiamo dovuto raggruppare (product.id e product.name sono campi liberi), mentre non abbiamo raggruppato per score, ovviamente. Di score ci interessava la media. Possiamo anche trovare la valutazione migliore e la peggiore:

```
SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE, MIN(SCORE) AS WORST, MAX(SCORE) AS BEST FROM PRODUCT
```

```
INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME;
```

Potremmo voler escludere le recensioni troppo vecchie:

```
SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE, MIN(SCORE) AS WORST, MAX(SCORE) AS BEST FROM PRODUCT
INNER JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME WHERE
YEAR(REVIEW.REVIEWDATE)>=2020;
```

E così via, ma resta un "problema". E' sempre un inner join, quindi non vedremo i prodotti senza recensioni:

```
SELECT PRODUCT.ID, PRODUCT.NAME, AVG(REVIEW.SCORE) AS AVERAGE, MIN(SCORE) AS WORST, MAX(SCORE) AS BEST FROM PRODUCT
LEFT JOIN REVIEW ON PRODUCT.ID = REVIEW.PRODUCTID GROUP BY PRODUCT.ID, PRODUCT.NAME WHERE
YEAR(REVIEW.REVIEWDATE)>=2020;
```

Creare le tabelle product e review, impostare le chiavi esterne e l'integrità referenziale, inserire dei dati di prova e poi provare a eseguire questa query (e verificarne i risultati) è lasciato come esercizio per lo studente.

Per terminare, non tutti i rapporti sono dichiarati e non tutti i join richiedono chiavi esterne. Supponiamo di avere una tabella buyer con una colonna budget, rappresentante quanto vuole spendere per un regalo, e di avere una tabella present (regali). Supponiamo di voler associare a ogni buyer tutti i regali che può permettersi:

```
SELECT BUYER.NAME, PRESENT.NAME FROM BUYER, PRESENT WHERE BUYER.BUDGET>= PRESENT.COST;
```

Anche questa è una relazione uno-a-molti, espressa non tramite una uguaglianza ma tramite una disuguaglianza, su cui non possiamo ovviamente applicare l'integrità referenziale. Il predicato di join non è per forza un uguaglianza, per quanto sia il caso più comune.

10.13 I self join e gli alias di tabella

In alcuni casi vorremo accoppiare i dati di una tabella con righe provenienti dalla stessa tabella. In questo caso diremo che prendiamo la tabella due volte, con due ruoli diversi.

Vediamo un esempio pratico. Supponiamo di avere una tabella employee con un rapporto 1-n fra impiegati: un responsabile coordina n risorse. In questo caso una riga della tabella impiegati avrà il ruolo di responsabile e sarà accoppiata a n righe col ruolo di risorse.

Il tracciato della tabella potrebbe essere il seguente:

```
CREATE TABLE EMPLOYEE
(
    ID INT PRIMARY KEY,
    NAME VARCHAR(100),
    SURNAME VARCHAR(100),
    COORDINATORID INT
);
```

coordinatorid è chiave esterna di employee verso employee, ed esprime il rapporto fra la riga e il suo coordinatore. Non esprimo integrità referenziale in quanto mi aspetto che ci siano impiegati senza responsabili, per cui coordinatorid dovrà essere NULL.

Ora, supponiamo di dover riportare il cognome della risorsa assieme al cognome del suo coordinatore. Cominciamo col prodotto cartesiano:

```
SELECT * FROM EMPLOYEE, EMPLOYEE;
```

Questa query darà errore! MySQL ha bisogno di distinguere le due tabelle per poter capire da quale delle due sta prendendo i campi, o meglio, da quale riga. La soluzione è fornire un alias di tabella:

```
SELECT * FROM EMPLOYEE COORDINATOR, EMPLOYEE RESOURCE;
```

La tabella employee viene presa una volta come coordinator, una volta come resource. Tutte le righe vengono combinate con tutte le altre, come abbiamo visto in precedenza. Ora evitiamo questo orrore tramite l'uso del predicato di join.

```
SELECT * FROM EMPLOYEE COORDINATOR, EMPLOYEE RESOURCE WHERE COORDINATOR.ID = RESOURCE.COORDINATORID;
```

MySQL "crede" di avere due tabelle, coordinator e resource, che in realtà esistono solo in questa query, ma le vede come tabelle separate e possono essere usate esattamente come se lo fossero, quindi col consueto join 1-n. Adesso possiamo proiettare i campi di nostri interesse:

```
SELECT RESOURCE.SURNAME AS RESOURCESURNAME, COORDINATOR.SURNAME AS COORDINATORSURNAME FROM EMPLOYEE COORDINATOR, EMPLOYEE RESOURCE WHERE COORDINATOR.ID = RESOURCE.COORDINATORID;
```

La sintassi tabella.campo è obbligatoria in questo caso: le due tabelle sono identiche e hanno gli stessi campi. MySQL non saprebbe distinguere altrimenti.

Poste queste differenze (gli alias di tabella obbligatori, la sintassi tabella.campo) si possono applicare tutti gli altri join. Ad esempio, possiamo scrivere una query pensata per identificare i possibili genitori e figli a partire dalle età nella tabella Person:

```
SELECT PARENT.ID AS PARENTID, CHILD.ID AS CHILID FROM PERSON PARENT, PERSON CHILD WHERE PARENT.SURNAME = CHILD.SURNAME AND YEAR(PARENT.DATEOFBIRTH) - YEAR(CHILD.DATEOFBIRTH) BETWEEN 20 AND 50
```

Troviamo tutte le persone con lo stesso cognome e per cui ci sia una differenza fra i 20 e i 50 anni fra la possibile riga genitore e la possibile riga figlio. In questo caso il predicato di join è costituito da tutte le condizioni dopo il where.

10.14 Il rapporto n-n

Il rapporto n a n non è supportato nativamente da MySQL, pur essendo molto comune. E' il tipo di rapporto per cui una riga di t1 è correlata con n righe di t2, e una riga di t2 con n righe di t1. Prendiamo un esempio concreto. Per molti arriva il momento in cui si torna a casa dicendo "ho preso la patente". E' una espressione imprecisa, come molte del linguaggio naturale. Non esiste una sola patente (potrebbero essere la B per gli autoveicoli, la A per le moto, una delle varie C per i veicoli pesanti), e soprattutto non c'è una sola patente A per tutta la popolazione mondiale.

Il signor James, tornando a casa con la sua patente in tasca, non ha l'unica patente A esistente. E' uno dei tanti a disporre di quella specifica certificazione. Non solo: il signor James potrebbe anche avere preso la patente B in precedenza, e quindi non è detto che James "abbia" una sola patente.

Il rapporto fra una persona e una patente è di tipo n-n se intendiamo la patente come "tipologia", non come oggetto fisico. La maggior parte di quelli che mi leggeranno avranno preso la patente B, ma c'è differenza fra il concetto di patente B e l'oggetto in plastica riposto nel portafogli.

Poniamo di avere, nella nostra solita tabella Person, i signori James e Jill, con id 1 e 2 rispettivamente, e di avere la seguente tabella License:

License

ID Type

1 A

2 B

3 C

La signorina Jill ha la patente A e B, il signor James solo la B. Una maniera barbara di esprimere questo sarebbe la seguente:

Person

ID Name Licenses

1 James B

2 Jill A,B

Ma questo è anatema e renderebbe estremamente complesse le query. Non si fa mai nella pratica.

La soluzione corretta è quella di creare una tabella intermedia. La regola generale è la seguente: un rapporto n-a-n fra t1 e t2 si scomponete in due rapporti 1-n verso una tabella intermedia che viene detta associativa (a). t1 n-a-n t2 diventa t1 1-n a e t2 1-n a. a deve contenere le chiavi primarie delle due tabelle lato 1 (t1 e t2), che fungeranno da chiavi esterne verso le rispettive tabelle.

Detta così è formalmente corretta ma oscura. Vediamo di chiarirlo col nostro esempio.

t1 è Person. Viene lasciata così com'è. t2 è License, e vale lo stesso discorso. Dobbiamo creare "a", la tabella associativa:

CREATE TABLE LICENSECERTIFICATION

```
(  
    ID INT PRIMARY KEY AUTO_INCREMENT,  
    PERSONID INT,  
    LICENSEID INT,  
    FOREIGN KEY(PERSONID) REFERENCES PERSON(ID),  
    FOREIGN KEY(LICENSEID) REFERENCES LICENSE(ID)  
)
```

E questo dovrebbe rendere le cose molto più chiare:

LicenseCertification id personid licenseid

1	1	2
2	2	1
3	2	2

L'id è chiave primaria per la certificazione, come al solito, ma i due id servono ad associare fra loro due righe. La riga 1 di person (James) è associata solo alla riga 2 di License, mentre la riga 2 di Person (Jill) è associata sia alla riga 1 che alla riga 2 di License. La riga 2 è associata sia a James che a Jill (entrambi "hanno" la patente B).

E adesso come rimettere tutto assieme? Abbiamo tre tabelle, quindi due predicati di join (n-1):

```
SELECT PERSON.NAME, PERSON.SURNAME, LICENSE.TYPE FROM PERSON INNER JOIN LICENSECERTIFICATION ON PERSON.ID =  
LICENSECERTIFICATION.PERSONID INNER JOIN LICENSE ON LICENSECERTIFICATION.LICENSEID = LICENSE.ID;
```

La tabella LicenseCertification viene detta associativa perché il suo scopo è esprimere l'associazione fra altre due tabelle ("entità"). Nasce per dire quali righe devono essere legate, e quali no, ma risponde anche a un'altra necessità.

Appurato il fatto che James e Jill hanno entrambi la patente B, James potrebbe averla presa prima o dopo di Jill. La patente di James potrebbe essere scaduta, e quella di Jill no, o viceversa. Il costo sostenuto per l'esame potrebbe essere diverso fra i due, e da patente in patente.

Queste informazioni non riguardano la persona, e non riguardano la patente. Riguardano il loro rapporto, la loro associazione, e infatti vengono detti attributi associativi. Servono a fornire maggiore dettagli sull'associazione di una riga di t1 a una riga di t2.

Miglioriamo la tabella sopra fornendo due attributi associativi:

```
CREATE TABLE LICENSECERTIFICATION
```

```
(  
    ID INT PRIMARY KEY AUTO_INCREMENT,  
    PERSONID INT,  
    LICENSEID INT,  
    OBTAINEDON DATE,  
    COST INT,  
    FOREIGN KEY(PERSONID) REFERENCES PERSON(ID),  
    FOREIGN KEY(LICENSEID) REFERENCES LICENSE(ID)  
)
```

id	personid	licenseid	obtainedon	cost
1	1	2	2020-01-01	2000
2	2	1	2021-01-01	1000
3	2	2	2019-01-01	1500

Da questo evinciamo che Jill ha preso prima la patente A e poi quella B. Notiamo anche che abbiamo dovuto specificare due date di ottenimento (una per certificazione), per cui sarebbe stato impossibile mettere questa informazione in un solo campo "licensedate" in Person.

Ricapitolando, e definendo le regole generali, un rapporto n-a-n fra due tabelle t1 e t2 viene riclassificato (scomposto) in due rapporti 1-n fra t1 e una associativa e t2 e la stessa associativa. t1 e t2 non sono collegati direttamente (infatti t1 non contiene chiavi esterne verso t2, e viceversa), ma indirettamente tramite l'associativa. Le informazioni che riguardano il rapporto fra le righe di t1 e quelle di t2 sono contenute nell'associativa, e vengono dette attributi associativi.

10.15 View in MySQL

Le query, come abbiamo visto, possono diventare complicate e lunghe da scrivere. MySQL ci mette a disposizione uno strumento per semplificare il compito riducendo le query più comuni a "pseudo tabelle".

Riprendiamo l'esempio del rapporto molti a molti. Supponiamo di avere spesso bisogno dei dati delle persone incrociati con quelli delle loro patenti. La query completa potrebbe essere la seguente:

```
SELECT PERSON.*, LICENSE.TYPE, LICENSECERTIFICATION.OBTAINEDON, LICENSECERTIFICATION.COST FROM PERSON INNER JOIN  
LICENSECERTIFICATION ON PERSON.ID = LICENSECERTIFICATION.PERSONID INNER JOIN LICENSE ON LICENSECERTIFICATION.LICENSEID  
= LICENSE.ID
```

Questa query prenderà tutti i campi di person, il tipo della patente e i dati del suo conseguimento, e li impacchetterà su di una sola riga. Per evitare di riscriverla ogni volta, possiamo fare come segue:

```
CREATE VIEW VIEWLICENSES AS  
SELECT PERSON.*, LICENSE.TYPE, LICENSECERTIFICATION.OBTAINEDON, LICENSECERTIFICATION.COST FROM PERSON INNER JOIN  
LICENSECERTIFICATION ON PERSON.ID = LICENSECERTIFICATION.PERSONID INNER JOIN LICENSE ON LICENSECERTIFICATION.LICENSEID  
= LICENSE.ID
```

Dopo di che, per ottenere gli stessi risultati, ci sarà sufficiente scrivere:

```
SELECT * FROM VIEWLICENSES;
```

viewlicenses è una query salvata che "finge" di essere una tabella. Avrà solo i campi che abbiamo specificato nella sua definizione (il codice "create view" di sopra), ma potremo usarla per le interrogazioni come se fosse una vera tabella:

```
SELECT * FROM VIEWLICENSES WHERE YEAR(OBTAINEDON) = 2020;
```

Ci restituirà i dati di tutte le patenti ottenute nel 2020.

Non è insolito avere viste che eseguono i join di tabelle che vengono spesso usate assieme, come in questo caso le tabelle person, licensecertificate e license.

Bisogna però fare attenzione alle omonimie. Siccome la vista "crede" di essere una tabella, non possiamo avere due campi con lo stesso nome. Eventuali omonimie in fase di definizione (due campi con lo stesso nome) andranno distinti tramite alias. Notiamo anche che non possiamo agire sui campi che non sono stati specificati esplicitamente in fase di definizione:

```
SELECT * FROM VIEWLICENSES WHERE LICENSEID = 1;
```

Non funzionerà, perchè la vista non ha proiettato licenseid dalla tabella licensecertificate. Dovremo scrivere:

```
SELECT * FROM VIEWLICENSES WHERE TYPE = 'A';
```

Ed è type, non license.type. Il riferimento alle tabelle originali si è perso, e viewlicenses è convinta di essere una tabella a sé. Nella pratica, esegue comunque una query sulle tre tabelle, a cui noi applichiamo successivamente dei filtri, dei raggruppamenti o quello che ci servirà.

11 - Estensioni di SQL in MySQL - Stored Procedures, Stored Functions e Triggers

Estensioni di SQL

SQL fa tutto tranne il caffè. E' grossomodo quello che hanno pensato gli sviluppatori di database attorno agli anni 90. In effetti SQL è uno strumento meraviglioso per quel che riguarda la ricerca dei dati e la loro manipolazione, ma è limitato. E' uno strumento specializzato ma non un linguaggio di programmazione completo. Manca dei concetti di sequenza, selezione e iterazione che sono alla base dei linguaggi di programmazione *universal*i.

Quindi perchè non darglieli?

Diversi DBMS vendors (un esempio su tutti, Oracle, col suo PL/SQL) si sono adoperati per offrire estensioni *procedurali* a MySQL. La programmazione procedurale è l'antenata diretta della programmazione a oggetti: potremmo dire che contiene i tipi primitivi, i cicli, il concetto di sequenza, le strutture di selezione e poco altro. Non ci sono i concetti di classe e oggetto, di metodo e di proprietà, ma non è un problema: in effetti la maggior parte dei programmi fino ai primi anni 90 venivano scritti secondo questo stile.

Un linguaggio dotato di questi strumenti è Turing-completo, o capace di computazione universale. Aggiungere questi elementi a SQL, con delle *estensioni proprietarie* significa elevarlo a qualcosa di più di un tool di ricerca o manipolazione del dato, per quanto specializzato. Significa renderlo un linguaggio, per quanto possibile, generale purpose.

Nella pratica questo si traduce nel poter aggiungere al database del codice, di cui riconosciamo principalmente tre *forme*:

- le *stored procedures*, letteralmente equivalenti alle vecchie procedure o subroutine, blocchi di codice che non sono tenuti a restituire un valore (ma possono farlo: in effetti possono restituire più valori). Possiamo immaginarle come metodi void, o come metodi a ritorno "mixed", come vedremo in Javascript, appartenendo all'oggetto database. Come tutti i metodi, accettano parametri.
- le *function*, che somigliano molto ai metodi che conosciamo, dovendo restituire un ritorno di un tipo specifico. Distingueremo le function in READS SQL DATA e DETERMINISTIC.
- i *trigger*, che sono la parte più utilizzata e più interessante. Si tratta di sottoprogrammi che vengono eseguiti automaticamente prima o dopo il verificarsi di un evento su una tabella del database. Possiamo pensarli come stored procedures con parametri predefiniti e impliciti, che ci consentiranno di controllare i dati inseriti nel database e di programmare azioni automatiche in risposta a degli eventi.

Ora andremo a esaminare ciascuna forma di codice a sè, facendo presente che si tratta di qualcosa di specifico di MySQL, a questo punto, e non di SQL in generale. Gli altri DBMS (Oracle, SQL Server, ecc...) hanno versioni molto simili dello stesso codice, ma non identico. SQL è, in massima parte, standardizzato. Le sue estensioni no. Troverete gli stessi concetti (procedure, function, trigger) in tutti i DBMS relazioni più avanzati, ma i dettagli del codice cambieranno.

Stored Procedures

Una stored procedure è un sottoprogramma memorizzato per esser richiamato in seguito. Viene tipicamente utilizzata per eseguire operazioni di "manutenzione" (ad esempio backup periodico dei dati) o query parametriche sui dati del database. In effetti la comodità di aggiungere costrutti di programmazione a un database è che "abbiamo" i dati come vicini di casa. Non serve caricarli: sono già nostri.

Vediamo qualche esempio:

```
DELIMITER &&
CREATE PROCEDURE GETSUNNYDAYS ()
BEGIN
SELECT DAY,LOCATION FROM WEATHERDAYS WHERE STATUS IN ('SUNNY','VERY SUNNY');
END &&
DELIMITER ;

CALL GETSUNNYDAYS();
-- RESTITUIRÀ GIORNO E LOCALITÀ DEI GIORNI DI SOLE
```

E' un esempio banale di creazione di una stored procedure (dal primo delimiter all'ultimo) e di richiamo (istruzione call). CALL invoca il "metodo" getSunnyDays. Ovviamente ci aspettiamo di avere una tabella weatherDays nel database, o questo codice non funzionerà.

Rispetto a una query semplice, o a una view, la stored procedure può restituire *diversi* resultsets, il risultato di molte query, e stamparle tutte a schermo. Non è un caso comune, ma può essere utile a volte.

Si tratta di un "metodo" banale, senza parametri. Vediamone un altro:

```
DELIMITER &&
CREATE PROCEDURE GETSUNNYDAYSFORLOCATION (LOC VARCHAR(100))
BEGIN
-- LOC È UN PARAMETRO IN INGRESSO. LO USO NELLA QUERY
SELECT DAY FROM WEATHERDAYS WHERE STATUS IN ('SUNNY','VERY SUNNY') AND LOCATION = LOC;
END &&
DELIMITER ;

CALL GETSUNNYDAYSFORLOCATION('RIMINI');
-- RESTITUIRÀ I GIORNI DI SOLE PER RIMINI.
```

Anche qui potremmo porci delle domande. E' davvero utile? (TODO)

12 - JDBC

12.1 Introduzione a JDBC

JDBC (Java Database Connectivity) è un insieme di interfacce, per la verità piuttosto vecchie ma ancora molto usate, che descrivono le procedure per l'interfacciamento di Java ai Database.

JDBC ci permetterà di scrivere e leggere dal DB tramite Java, che diventerà il client del DBMS. Questo avviene tramite oggetti che implementano determinate interfacce (Connection, Statement, PreparedStatement, ResultSet), e che verranno creati per noi da altri oggetti.

In questo caso, come in quasi tutti gli altri, ci interessano i tipi formali, non quelli concreti. Tuttavia, per disporre delle classi concrete che implemetteranno le interfacce dobbiamo scaricare le librerie (i "connettori") relativi al nostro DBMS e collegarli al path dell'applicazione.

Ci procureremo la versione 5.1.49 (attualmente reperibile a questo indirizzo: <https://downloads.mysql.com/archives/c-j/>) in versione platform independent (un file zip). Scaricato il file zip lo spacchetteremo e collegheremo il file .jar al suo interno al path dell'applicazione. A questo punto disporremo delle classi necessarie per la connessione con MySQL. Avremo "installato" l'implementazione JDBC per MySQL.

Ora andiamo a esaminare le interfacce principali.

12.2 Connection

Possiamo immaginare una connection (connessione) come un tubo che collega il programma Java e il server MySQL (o comunque un DBMS). Il traffico sul "tubo" è bidirezionale: Java invia comandi SQL al DBMS (l'unica lingua che questo possa capire), mentre il DBMS risponde fornendo dati (righe) presi dalle sue tabelle o messaggi di errore o di successo.

La connection resta aperta e attiva finchè non viene chiusa, e può essere usata per creare gli oggetti Statement che vedremo a breve, e che corrisponderanno a comandi SQL.

La creazione della connessione può generare eccezioni, che andrebbero gestite di conseguenza (si tratta di solito di un errore fatale: difficile lavorare senza un database sotto, ma a volte si può provare su un db di diversa, o a connettersi a un altro server). La forma classica è la seguente:

```
import java.sql.Connection;
// ometto il resto della classe
Class.forName("com.mysql.jdbc.Driver");
//verifico di avere le classi necessarie
Connection connection = DriverManager.getConnection(connectionstring);
```

DriverManager è una classe, e getConnection un metodo statico che prova a creare una connessione, non ci interessa come o di che tipo particolare, a partire da una stringa formattata in maniera particolare che viene detta stringa di connessione ("connection string").

La stringa di connessione specifica in maniera compatta il server e il database da cui prendere i dati, il nome utente e la password da utilizzare. Vediamo un esempio:

```
jdbc:mysql://localhost:3306/census?user=root&password=pwd
```

Si legge come: "crea una Connection JDBC" (jdbc:), verso un DBMS mysql (mysql:), che è attivo sul server fisico localhost (localhost) sulla porta 3306 (:3306). Di quel server ci interessa il database census. Per accedere manda i parametri user (?user=root) e password(& password=pwd);

Se tutti i parametri sono corretti il metodo getConnection ci restituirà un oggetto che implementa l'interfaccia java.sql.Connection, vale a dire un qualche tipo di connessione adatta al DBMS su cui stiamo lavorando.

Per ora abbiamo omesso la gestione delle eccezioni, che vedremo nell'esempio finale.

Il "tubo" a questo punto è aperto e aspetta i nostri comandi.

12.3 Statement

Uno Statement è, letteralmente, un comando SQL inviato al DBMS, nella speranza che lui lo esegua. Anche Statement è una interfaccia, non una classe, e verrà creato da qualcun altro per noi (dalla connessione, per essere precisi). Noi lo popoleremo col comando che vogliamo eseguire e lo invieremo lungo il "tubo". A seconda dei casi, ci aspetteremo di avere un ritorno di un qualche tipo.

L'uso è abbastanza immediato:

```
// ponendo di avere già la connessione Statement newPersonSQL = connection.createStatement(); newPersonSQL.execute("INSERT INTO Person VALUES('Mario', 'Rossi'); newPersonSQL.close(); //buona norma "chiudere" gli statement dopo l'uso //a indicare che abbiamo finito di usarli
```

Abbiamo di nuovo omesso la gestione degli errori per semplicità. Lo statement "incapsula" la sintassi SQL e la invia al database. Il database, auspicabilmente, inserirà il signor Mario Rossi nella tabella Person.

Cosa potrebbe andare storto? Tutto. Potremmo non avere le autorizzazioni per scrivere sulla tabella Person. Potrebbe non esistere una tabella Person. Person potrebbe avere più di due campi (e in questo caso la Insert fallirà, visto che non abbiamo specificato i nomi dei campi secondo la sintassi INSERT (campi) INTO tabella VALUES(valori)).

Tutte queste evenienze prendono il nome di SQLException, e vanno gestite o propagate esplicitamente (eccezioni checked).

Notiamo che insert era un'operazione di scrittura. Non ci aspettiamo un particolare ritorno (anche se potremmo averlo, volendo). Diverso il discorso nel caso di una operazione di lettura:

```
Statement readPeopleSQL = connection.createStatement(); ResultSet peopleRows = readPeopleSQL.executeQuery("select * from person");
```

Questo ci porta a dover introdurre una nuova interfaccia: ResultSet. ResultSet significa letteralmente "insieme di risultati", ed è nella pratica una lista di righe. In questo caso, tutte le righe della tabella Person, con tutti i campi.

ResultSet è, di gran lunga, l'interfaccia più complessa e più difficile da gestire in JDBC, e ora vedremo come usarla in pratica.

12.4 ResultSet

ResultSet è un'interfaccia pensata per definire il comportamento di un insieme di risultati, intesi come risultati di una query SQL inviata al DBMS.

Possiamo vederlo come una lista di righe, ciascuna delle quali con le stesse colonne, ed è prodotto dall'esecuzione di uno statement di lettura (metodo executeQuery dell'oggetto con interfaccia Statement).

Cominciamo con un esempio di base, supponendo che la tabella Person abbia id, name, surname, salary e gender:

```
Statement readPeopleSQL = connection.createStatement();
ResultSet peopleRows = readPeopleSQL.executeQuery("select name,surname,salary from person");
while(peopleRows.next())
    System.out.println(peopleRows.getString("name")+" "+peopleRows.getString("surname")+" earns:"+peopleRows.getInt("salary"));
```

Questo codice presenta diverse "trappole". La prima è nel metodo next(). Noi siamo abituati ad usare hasNextLine() e nextLine(), con Scanner. hasNextLine() ci dice se c'è una riga da leggere, nextLine() la legge e si sposta alla successiva.

next() di ResultSet è più sbrigativo. Se restituisce true "cambia" la riga selezionata nel ResultSet, andando alla successiva. In pratica, fonde il funzionamento di hasNextLine() e nextLine(). E per quanto il ResultSet contenga tutte le righe del risultato, ne "seleziona" una per volta. Ipotizziamo che la tabella contenga Mario Rossi, Luigi Verdi e Penelope Bianchi:

```
peopleRows.next();
System.out.println(peopleRows.getString("name"));
// "Mario"
peopleRows.next();
System.out.println(peopleRows.getString("name"));
// "Luigi"
peopleRows.next();
System.out.println(peopleRows.getString("name"));
// "Penelope"
```

Lo stesso metodo (peopleRows.getString(nomcampo)) restituisce valori diversi a seconda della *riga selezionata*. Potete immaginare di avere un cursore che si muove avanti (e potenzialmente indietro) sulle righe selezionate.

E' anche necessario dare un primo next() per usare il ResultSet. La prima riga selezionata è la riga "0". Dovremo sempre verificare di avere delle righe (while(resultSet.next()) prima di operare.

Una volta selezionata una riga (tramite next() di solito, la riga successiva), andremo a leggere i campi che ci interessano. ResultSet dispone di una serie di metodi pensati per convertire automaticamente i valori dei campi in tipi di Java. Il più basilare è getString():

```
peopleRows.next();
System.out.println(peopleRows.getString("name"));
peopleRows.next();
System.out.println(peopleRows.getString("salary"));
//salario di Luigi, stampato come stringa
//resultSet cerca di convertire automaticamente il valore del campo nel tipo che ci serve. Non sempre ci riesce.
```

Possiamo specificare il nome del campo da stampare o il suo posto nella riga. Se salary è il terzo campo della query, possiamo anche scrivere:

```
peopleRows.next(); System.out.println(peopleRows.getString(3)); //salario di Mario
```

Sì, non vi state sbagliando. ResultSet comincia a contare da 1, e sì, stiamo trattando un numero come una stringa. E' possibile, per quanto non consigliato.

Notiamo anche che non contano i campi della tabella: solo quelli della query. ResultSet non vede la tabella, ma il risultato della query, che potrebbe essere su una vista, su una tabella, su dieci. Vediamo un esempio più sensato:

```
Statement readPeopleSQL = connection.createStatement();
ResultSet peopleRows = readPeopleSQL.executeQuery("select name,surname,salary from person");
double sum = 0;
while(peopleRows.next())
{
    System.out.println(peopleRows.getString("name")+" "+peopleRows.getString("surname")+" earns:"+peopleRows.getDouble("salary"));
    sum+=peopleRows.getDouble("salary");
}
System.out.println("Total paid:"+sum);
```

Ma se ci fosse interessato solo il totale, avremmo potuto fare più rapidamente in questo modo:

```
Statement totalPaidSQL = connection.createStatement();
totalPaidRow = totalPaidSQL("select sum(salary) as s from Person");
if(totalPaidRow.next())
    System.out.println(totalPaidRow.getString("s"));
```

Questo esempio illustra la potenza di SQL in combinazione con Java. Il calcolo è stato eseguito da SQL (e molto, molto rapidamente) e poi trasmesso a Java. Il ResultSet non è vincolato alla forma della tabella, ma alle colonne che noi decidiamo di proiettare. In questo caso, sto stampando il risultato di una funzione di gruppo.

Essendo sicuri del fatto che il risultato era composto da una sola riga, non è servito usare un while (non dovevamo spostare il cursore per n righe, era sufficiente spostarlo alla prima), per cui ci è bastato usare un if. if(rs.next()) è una forma comoda per dire "se c'è una riga di risultato, prendila".

12.5 Un primo esempio completo

A questo punto riportiamo un esempio completo di uso di JDBC su un database con una tabella "Person" e una tabella "Log", per la stampa di dati riassuntivi:

```
import java.sql.*;
public class ResultSetDemo
{
    public static void main(String[] args)
    {
        String connectionString = "jdbc:mysql://localhost:3306/census?user=root&password=pwd";
        Connection connection = null;
        try
        {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection(connectionString);
        }
        catch(SQLException e)
        {
            System.out.println("Cannot connect to db using " + connectionString+ ". Aborting.");
            return;
        }
    }
}
```

```

    {
        double sum = 0;
        Statement readPeopleSQL = connection.createStatement();
        ResultSet peopleRows = readPeopleSQL.executeQuery("select name,surname,salary from person");
        while(peopleRows.next())
        {
            System.out.println(
                (
                    peopleRows.getString("name")+" "+
                    peopleRows.getString("surname")+" earns:"+
                    peopleRows.getDouble("salary")
                );
            sum+=peopleRows.getDouble("salary");
        }
        System.out.println("Total paid:"+sum);
        //terminato di usarli, li chiudiamo
        peopleRows.close();
        readPeopleSQL.close();
    }
    catch(SQLException e)
    {
        System.out.println("Problems accessing the DB for reading.");
        e.printStackTrace();
        return;
    }
    try
    {
        Statement updateLogSQL = connection.createStatement();
        updateLogSQL.execute("insert into log (reason) values('Test queries')");
        //registriamo l'uso del db a scopo statistico
        updateLogSQL.close();
    }
    catch(SQLException e)
    {
        System.out.println("Problems updating the log.");
        e.printStackTrace();
        return;
    }
    connection.close();
}
}

```

Questo copre l'uso basilare delle interfacce di JDBC. Possiamo vederlo come un modo "furbo" di leggere da file, con cui carichiamo solo i dati che ci interessano per lavorarci sopra. Se avessimo voluto vedere solo i milanesi, ad esempio, sarebbe stato sufficiente porre "select name, surname, salary from Person where city = 'Milan'". Questo ci porta all'elemento successivo: l'uso di queries parametriche.

12.6 Queries parametriche in JDBC e PreparedStatement

L'esempio precedente era accademico, ma in generale vorremo lavorare solo su sottoinsiemi dei dati, tipicamente selezionati dall'utente. In questi casi lavoreremo su query parametriche, vale a dire contenenti porzioni di input inserite dall'utente:

```

// omettiamo il codice di gestione degli errori e gli import per concentrarci sul nocciolo del discorso
String city = scanner.nextLine();
double sum = 0;
Statement readPeopleSQL = connection.createStatement();
ResultSet peopleRows = readPeopleSQL.executeQuery("select name,surname,salary from person where city='"+city+"'");
while(peopleRows.next())
{
    System.out.println(peopleRows.getString("name")+" "+peopleRows.getString("surname")+" earns:"+peopleRows.getDouble("salary"));
    sum+=peopleRows.getDouble("salary");
}
System.out.println("Total paid in "+city+": "+sum);

```

Questa metodica risulta comoda anche per il calcolo di totali, che vengono poi facilmente "metodizzati":

```

public double totalPaid(String city)
{
    double res = 0;
    Statement readPeopleSQL = connection.createStatement();
    ResultSet total = readPeopleSQL.executeQuery("select sum(salary) as s from person where city='"+city+"'");
    res = total.next() ? total.getDouble("s") : 0;
    total.close();
    readPeopleSQL.close();
}

```

```
    return res;
}
```

E' circa quello che succede quando inseriamo una parola chiave in un qualunque motore di ricerca: `SELECT * FROM TheWeb WHERE testo like '%!'+key+'%'`.

Questo modo di lavorare ci espone a dei problemi però. Cosa succederebbe se invocassimo il metodo `totalPaid("L'Aquila");`? Sostituendo il parametro, vedremo che il comando SQL risultante è `select sum(salary) as s from person where city='L'Aquila'`. L'apostrofo spezza la stringa di SQL, e manda in crash la query, quando va bene.

Quando va male, permette ad hacker e altri loschi figuri di entrare nel nostro database e rubare o distruggere dati. Questa tecnica viene detta *SQL Injection*, ed è un errore difficile da giustificare in fase di produzione. E' derivante dal fatto che non abbiamo *sanitizzato* l'input.

Ci sono vari modi di farlo. Uno di questi è sanitizzare "a mano" le stringhe che inseriremo nelle query parametriche:

```
readPeopleSQL.executeQuery("select sum(salary) as s from person where city='"+city.replace("'", "\")+"');
```

Questa tecnica funziona, ma è primitiva e c'è il modo di aggirarla. JDBC dispone di una interfaccia apposita che permette di inserire parametri dentro le query e li sanitizza automaticamente. Viene detta `PreparedStatement` e si può usare in questo modo:

```
PreparedStatement ps = connection.prepareStatement("select * from Person where city = ?");
ps.setString(1, city);
ResultSet rows = ps.executeQuery();
```

Il punto interrogativo viene detto placeholder (segnaposto) e deve essere sostituito con un valore prima dell'esecuzione della query. Il metodo `setString` è l'analogia di `getString` per il `resultset`, e lavora sulla posizione. Stiamo dicendo "sostituisci il primo segnaposto col valore della variabile `city`".

I `PreparedStatement` possono essere usati sia per operazioni di scrittura che di lettura, e sanitizzano automaticamente il dato. In effetti, in tutti i casi in cui si vogliono passare parametri a una query, è sempre meglio usare i `PreparedStatement`. Vediamolo con un inserimento:

```
public void save(Person p) throws SQLException
{
    String sql = "INSERT INTO Person(name,surname,gender, salary) VALUES (?, ?, ?, ?)";
    PreparedStatement ps = connection.prepareStatement(sql);
    ps.setString(1, p.getName());
    ps.setString(2, p.getSurname());
    ps.setString(3, p.getGender());
    ps.setDouble(4, p.getSalary());
    ps.execute();
}
```

Ipotizzando che l'id sia `auto_increment`, abbiamo appena salvato una persona nel database con un metodo apposito, che "traduce" l'oggetto in riga. Questo viene detto **ORM**, Object Relational Mapping, letteralmente "mappatura da oggetto a db relazionale", e viceversa.

12.7 Object Relational Mapping (ORM) e DAO

Java parla e pensa in oggetti, collegati fra loro da relazioni di uso o ereditarietà. SQL pensa in tabelle collegate fra loro da chiavi. Si tratta di due rappresentazioni differenti dello stesso dato.

Java potrebbe usare direttamente il Resultset e lo abbiamo anche fatto in precedenza, ma è un approccio scomodo. Sarebbe più naturale avere una entity completa di metodi e campi ben tipizzati.

In generale, vorremo trasformare le righe del DB in entities, e viceversa. A tale scopo utilizzeremo un componente specifico che chiameremo DAO - Data Access Object.

In generale, se abbiamo una entità E avremo un DAO corrispondente. Ponendo di avere la entity Person, potremmo avere:

```
public interface PersonDAO
{
    //questa interfaccia definisce il comportamento di un componente che, nella vita, legge e scrive persone
    default Person load(int id) throws SQLException
    {
        List <Person> res = list (" id="+id);
        return res.size()==0 ? null : res.get(0);
    }
    Person save(Person person) throws SQLException;
    List <Person> list (String criteria) throws SQLException;
    default List <Person> list() throws SQLException
    {
        return list(" 1 = 1");
    }
    boolean delete (int id) throws SQLException;
}
```

E in effetti questi metodi potremmo riproporli per tutti i DAO. Sono le operazioni CRUD di base (Create, Read, Update, Delete): save(Person) viene utilizzato per creare o aggiornare persone (create ed update), delete per cancellare (delete), list e load per leggere.

List, in particolare, è definito come "leggi delle righe dal db che rispettino un criterio (criteria) e trasformale in una lista di oggetti". Partendo dalla ricerca condizionata per criterio ricaviamo la lista generale (criterio 1=1, una condizione sempre vera che passeremo alla SELECT di ricerca).

Allo stesso modo, dalla ricerca condizionata recuperiamo il caricamento della singola Person (metodo load), impostando come condizione una uguaglianza su id. A quel punto potremo avere al massimo una persona come risposta. Se c'è, la restituiamo. Altrimenti restituiamo null. Vediamolo implementato:

```
package dao;
import java.sql.*;
import java.util.List;
import java.util.ArrayList;

public PersonDAOMySQL implements PersonDAO
{
    //Il DAO per funzionare ha bisogno della connessione
    Connection connection;
    public PersonDAOMySQL(Connection connection)
    {
        this.connection = connection;
    }
    public Person save(Person person) throws SQLException
    {
        //questo metodo fa da create (insert)
        //e da update (update) per una person
        //dobbiamo verificare che esista già, o meno
        Person present = load(person.getId());
        if(present == null)
        {
            //ok, non esiste. Vediamo se la nuova Person ha già un id, altrimenti glielo dobbiamo assegnare
            if(person.getId() <= 0)
            {
                Statement maxIdSQL = connection.createStatement();
                maxIdRs = maxIdSQL.executeQuery("select max(id)+1 as m from Person");
                person.setId(maxIdRs.getInt("m") > 0 ? maxIdRs.getInt("m") : 1);
                maxIdSQL.close();
                maxIdRs.close();
            }
            // e ora va salvato
            String sql = "INSERT INTO Person (id, name, surname, dateofbirth) VALUES (?, ?, ?, ?)";
        }
    }
}
```

```

        PreparedStament insertSQL = connection.prepareStatement(sql);
        insertSQL.setInt(1, person.getId());
        insertSQL.setString(2, person.getName());
        insertSQL.setString(3, person.getSurname());
        insertSQL.setString(4, person.getDateofbirth().toString());
        insertSQL.execute();
        insertSQL.close();
    }
}
else
{
    //esiste e va aggiornato
    //UPDATE, non INSERT
    String sql = "UPDATE Person SET name=? , surname=?, dateofbirth = ? WHERE id=?";
    PreparedStament updateSQL = connection.prepareStatement(sql);
    insertSQL.setInt(4, person.getId());
    insertSQL.setString(1, person.getName());
    insertSQL.setString(2, person.getSurname());
    insertSQL.setString(3, person.getDateofbirth().toString());
    updateSQL.execute();
    updateSQL.close();
}
// lo restituiamo, con in più l'id se prima non lo aveva.
return person;
}
List <Person> list (String criteria) throws SQLException
{
    List <Person> res = new ArrayList <Person>();
    Statement listSQL = connection.createStatement();
    // criteria è una condizione da mettere nel where
    // in modo da prendere solo le righe che ci interessano
    ResultSet listRs =
        listSQL.executeQuery("select * from Person where "+criteria);
    //Ora ho un elenco di righe: OGNI RIGA VA TRASFORMATA IN UN OGGETTO PERSON: per farlo utilizzo un metodo privato, come ci siamo già abituati a fare
    while(listRs.next())
    {
        Person newperson = _rsToPerson(listRs);
        res.add(newperson);
    }
    return res;
}
// metodo privato: entra una result, che punta a UNA sola riga, e ne esce una Person ricostruita a partire dai dati della riga, come facevamo per i file.
private Person _rsToPerson(ResultSet rs) throws SQLException
{
    Person p = new Person();
    p.setId(rs.getInt("id"));
    p.setName(rs.getString("name"));
    p.setSurname(rs.getString("surname"));
    p.setDateofbirth(rs.getString("dateofbirth"));
    return p;
}
public boolean delete (int id) throws SQLException
{
    // vediamo se esiste. Se non esiste restituiamo false
    if(load(id)==null) return false;
    Statement deleteSQL = connection.createStatement();
    deleteSQL.execute("delete from Person where id="+id);
    deleteSQL.close();
    //ok, cancellato.
    return true;
}
}

```

Avrete notato una somiglianza notevole col lavoro sui file. Il funzionamento di base non cambia, ma dobbiamo notare che possiamo caricare solo quello che ci interessa, e disponiamo anche di una interfaccia di scrittura "naturale". Vediamo un esempio di codice:

```

Connection connection = DriverManager.getConnection(connectionstring);
PersonDAO dao = new PersonDAOMySQL(connection);
System.out.println("Inserire il nome di una città");
String city = keyboard.nextLine();
for(Person p:dao.list("city='"+city+"'"))
    System.out.println(p.getId()+" "+p.getName()+" "+p.getSurname()+":"+p.getAge());
System.out.println("Inserire id da cancellare");
int id = Integer.parseInt(keyboard.nextLine());
if(dao.delete(id))
    System.out.println("Elemento cancellato");

```

```
else
    System.out.println("Impossibile trovare questo id");
```

Per ricapitolare: data una entità P, avremo PDAO, che offrirà i servizi necessari per leggere, scrivere e ricercare l'entità sul DB. DAO sta per "Data Access Object", e si occupa di fare ORM, vale a dire di trasformare le righe del DB in oggetti (e viceversa).

12.8 ORM per entità complesse: il caso 1-n

Ipotizziamo un rapporto di composizione fra la entity Person e una seconda Entity di nome Good. Nel nostro schema, un bene è di proprietà di una sola persona. E' una semplificazione che accettiamo per amor di esempio.

Supponiamo di voler sapere, per ogni persona, l'elenco dei beni a sua disposizione. Questo significherà che Person conterrà una lista di Good.

```
joel.getGoods(); // {guitar, gasmask, brokenclock}  
joel.getGoods().get(0); // guitar
```

Abbiamo quello che viene detto un collegamento a direzionalità non specificata 1-n. Joel conosce i suoi beni, mentre non abbiamo specificato se il bene abbia un riferimento al proprio proprietario.

Nel db, questo si tradurrebbe in una relazione 1-n fra Person e Good, con Good avente tracciato record (id, name, description, cost, ownerid foreign key verso Person), e ci crea un problema: come carichiamo una Person dal db?

Possiamo caricare solo il lato 1, quindi solo il lato Person, senza caricare la lista dei beni, che resterà vuota, o caricarla "completa", e quindi creare un oggetto di tipo Person significherà creare n+1 oggetti: n Good e una Person, per popolare la lista di Person. Cominciamo col farlo "a mano":

```
PersonDAO persondao = new PersonDAOMySQL(connection);  
GoodDAO gooodao = new GoodDAOMySQL(connection);  
  
//ipotizzando che joel sia il numero 1  
Person joel = persondao.load(1);  
// perchè ownerid=1?  
// sto caricando le righe della tabella good in cui  
// owner = 1. Vale a dire, le righe collegate a joel.  
joel.setGoods(gooodao.list("ownerid=1"));
```

Per caricare Joel da DB ho dovuto utilizzare due DAO: uno di Person (l'entità *master*) e uno di Good (l'entità *detail*). Per pulizia del codice, questo potrebbe essere assemblato in un oggetto "business logic" che funzionerà, fra l'altro, come super-DAO:

```
public class PersonBL  
{  
    PersonDAO persondao;  
    GoodDAO gooodao;  
    public PersonBL(Connection connection)  
    {  
        persondao = new PersonDAOMySQL(connection);  
        gooodao = new GoodDAOMySQL(connection);  
    }  
    public Person load(int id, boolean complete) throws SQLException  
    {  
        Person res = persondao.load(id);  
        if(complete)  
            res.setGoods(gooodao.list("ownerid="+id));  
        return res;  
    }  
    public Person save(Person person, boolean complete) throws SQLException  
    {  
        Person res = persondao.save(person);  
        if(complete)  
            for(Good good:person.getGoods())  
                gooodao.save(good);  
        return res;  
    }  
    public boolean delete(int id) throws SQLException  
    {  
        //perchè non tocco gooodao? un indizio: ipotizziamo di avere on delete cascade fra person e good nel database.  
        return persondao.delete(id);  
    }  
}
```

Sia per load che per save abbiamo specificato un boolean, un "flag", complete. Se scelgo "complete" caricare un oggetto Person significherà usare *due* DAO: un DAO per caricare la parte di Person che è nella tabella Person, e il secondo per caricare tutte le righe collegate in Good. Verranno creati n+1 oggetti.

Lo stesso vale per il salvataggio. Posso scegliere di salvare la persona "incompleta" (verrà scritta una sola riga in Person) o completa (una riga in Person, n righe in Good, una per ogni bene).

Per delete il discorso è più semplice. Ipotizzando un rapporto di composizione, possiamo avere impostato on delete cascade come clausola di integrità referenziale fra person e good: cancellando person, vengono cancellati a ruota i suoi good.

Queste tecniche si chiamano anche ORM 1-n, e possono diventare molto complesse. Ci sono diversi altri casi da considerare. Come modelliamo in ORM i rapporti di ereditarietà? Come modelliamo in ORM i rapporti n-n? E se volessimo collegare non solo la Person al Good (1-n unidirezionale), ma anche Good a Person (relazione 1-n bidirezionale)?

Queste problematiche sono comuni, così come la trasformazione da riga a oggetto e viceversa, e disponiamo di strumenti automatici o semi-automatici per risparmiarci la fatica. E' però necessario che lo studente capisca il meccanismo di base (oggetti collegati) per sfruttare correttamente quegli strumenti, che prendono il nome di JPA.

13 - JPA

13.1 Introduzione a JPA

JPA (Java Persistence API, dove API sta per Application Programming Interface) è una *specifica*, vale a dire un insieme di interfacce che implementare che dovranno fornire un insieme di servizi, nel nostro caso i servizi relativi alla permanenza degli oggetti sul DB.

I DB potrebbero essere di qualunque tipo. Non tutti i DB sono DB SQL. In alcuni casi potrebbe essere possibile salvare gli oggetti tali e quali, senza bisogno di trasformazione, ma non è il caso comune. Tipicamente, come abbiamo visto, dovremo mappare gli oggetti a tabelle. E' il concetto che abbiamo indicato come ORM.

JPA è uno standard che indica *come* mappare gli oggetti (che sono "cittadini di prima classe", in Java) ai database tramite una serie di annotation. Le annotation, lo ricordiamo, sono arricchimenti del codice pensati per essere letti da altro codice, e sono usate pesantemente nella pratica di Java.

Vediamo un esempio:

```
@Entity  
public class Person  
{  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int id;  
    private String name;  
    private String surname;  
    private int age;  
    //getter, setter, metodi...  
}
```

Le annotation sono abbastanza chiare. @Entity vuol dire che questa classe è una entità: qualcosa che "esiste" e che vogliamo salvare nel database. JPA sarà abbastanza furbo da riuscire a farlo in automatico o quasi.

@Id è intuitivo. La proprietà, o meglio il *campo* sopra cui scriviamo l'annotazione è l'id della tabella. JPA definirà anche che possa essere popolato automaticamente, tramite la seconda annotazione, subito sotto. La proprietà id è un @Id (chiave primaria) autogenerato (@GeneratedValue), la cui strategia è "Identity" (auto_increment).

Fornendo queste informazioni di base, JPA sarà in grado di caricare, ricercare e salvare questa entità con uno sforzo minimo da parte del programmatore.

13.2 Una implementazione di JPA - Eclipse Link e sua configurazione

EclipseLink è una implementazione di JPA. Questo vuol dire che fornisce componenti che implementano i servizi (le interfacce) definite in JPA. Nel concreto si tratta di una serie di JAR da collegare al path del progetto. La versione di riferimento per il corso è EclipseLink 2.5.2 (<https://www.eclipse.org/eclipselink/releases/2.5.php>), ma Eclipse dispone anche di un wizard per il download e il collegamento al path di EclipseLink. Vi si può accedere, al momento della scrittura di questo testo, tramite File-> Configure -> Convert to JPA Project, partendo da un progetto Java standard. Nel caso in cui la versione 2.5.2 non fosse scaricabile, è possibile lavorare altrettanto bene con la 2.2.1.

A questo punto sarà necessario specificare i dettagli della connessione al DB. JPA ha comunque bisogno di una connessione: possiamo vederlo come uno strato "superiore" posto su JDBC. La configurazione della connessione viene rimandata all'apposita appendice. A noi interessa qualcosa di diverso: la Persistence Unit.

Come in molti altri casi per Java, la Persistence Unit è un oggetto configurato tramite un file XML. Il file XML (persistence.xml, la cui posizione non bisogna toccare salvo far collassare il sistema) è il file di configurazione di EclipseLink, e definisce la Persistence Unit, vale a dire una "unità di persistenza". Volendo essere meno fiscali, rappresenta la connessione "intelligente" a un database, il punto di accesso per i suoi dati.

La configurazione contiene l'indirizzo del database, la tipologia di connessione, i dati relativi all'utente e alla password e altre informazioni necessarie a gestire il database. Vediamo un esempio:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

  <persistence-unit name="CensusJPA" transaction-type="RESOURCE_LOCAL">

    <!-- la classe che abbiamo mappato da principio, quella contenente @Entity -->
    <class>com.generation.censusjpa.model.entities.Person</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/censusjpa"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="rootpassword"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    </properties>
  </persistence-unit>
</persistence>
```

Cosa stiamo dicendo in questo file? Che ci stiamo connettendo a un database mysql. Che stiamo usando il relativo driver (com.mysql.jdbc.Driver). Che ci stiamo connettendo al database censusjpa, con l'utente root e con la relativa password. E' tutto abbastanza esplicativo. L'unico elemento interessante è il tag class, che in questo caso indica una "classe gestita". Questa persistence unit si prenderà carico della gestione di quella classe, di leggere e scrivere i suoi oggetti sul database.

13.3 Mappatura e operazioni di base su una entità semplice. Entità gestite e non gestite

Una volta ottenuta la persistence unit (persistence.xml), possiamo riferirci alla stessa tramite il suo nome (attributo name nel tag persistence-unit) per creare un *Entity Manager*.

Un Entity Manager è letteralmente un gestore di entità, vale a dire di classi marcate con @Entity, e con una corrispondenza a una o più tabelle del database, secondo le logiche di ORM che abbiamo visto in precedenza. Creiamo la entity manager a partire dalla persistence unit, in questo modo:

//una factory di entity manager. Per chiarezza vedere la sezione sui pattern

```
EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("CensusJPA");
EntityManager em = emfactory.createEntityManager();
```

Una volta creata possiamo usarla per leggere e per scrivere in maniera "naturale" e object-oriented, a patto che le classi siano state annotate correttamente. *EntityManager dipende dalle annotazioni*. Se la classe Person non presenta le annotazioni corrette (lo vedremo a breve) il sistema non funzionerà.

A configurazione finita, il meccanismo è abbastanza semplice:

```
Person p = em.find(Person.class,1);
//Carichiamo la persona con id=1. Person.class indica alla entity manager in quale classe, intesa stavolta come "contenitore", guardare
// stampo il toString()
System.out.println(p);
```

Le righe del database sono state caricate e convertite in una persona. L'entità così caricata è ora "collegata" a JPA, che la riconosce come "sua". Quando andremo a modificarla e a risalvarla, per lei sarà una vecchia conoscenza e riuscirà a capire che la deve aggiornare, non crearne una nuova. Un metodo per farlo è il seguente:

```
// perchè casto a Person? find restituisce Object in questo caso
Person p = (Person) em.find(Person.class,1);
// compleanno
p.setAge(p.getAge()+1);
em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();
```

Le ultime tre righe necessitano di una spiegazione più approfondita, ma per ora concentriamoci sulla seconda: persist. Potete leggerlo come "salva", o letteralmente "persisti quel dato" sul database.

In questo caso stiamo salvando una Person, e si tradurrà in una INSERT o in una UPDATE su database a seconda che la persona sia già presente o ancora da inserire (nel nostro caso, un update).

Ci sono dei tranelli da evitare. Supponiamo di avere già una persona con id 1, e di scrivere questo codice:

```
Person p = new Person();
p.setId(1);
p.setName("John");
p.setSurname("Smith");
p.setAge(40);
em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();
```

Ci aspetteremmo che aggiornasse la persona con id=1, ma invece creeremo altri John Smith. L'id verrà ricalcolato ogni volta (è un valore generato automaticamente, come specificato nelle annotazioni), e noi avremo ripetizioni della stessa riga.

L'oggetto Person che abbiamo creato è una entità *non gestita*, è qualcosa di nuovo che JPA non conosce. Salvandolo con persist lo interpreta come un oggetto nuovo, gli fornisce un id e lo salva, dopo di che lo "conosce", ed è *gestito*.

```
Person p = new Person();
p.setId(1);
p.setName("John");
p.setSurname("Smith");
p.setAge(40);
//insert
em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();
p.setAge(41);
//update
```

```

em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();

```

Il primo persist creerà un "oggetto" nuovo nel db. Al secondo l'oggetto sarà gestito e riconosciuto e verrà aggiornato, non creato. In generale, per modificare un oggetto già esistente prima dobbiamo "collegarlo". Per un oggetto nuovo è sufficiente salvarlo, come abbiamo appena visto, ma per uno già esistente è necessario caricarlo dal db, come abbiamo visto prima, tramite la combinazione di find (o di altre operazioni di caricamento) e persist. Non è la soluzione ottimale, ma andrà bene per ora.

Ora bisogna spiegare cosa intendiamo con getTransaction(), begin() e commit(). getTransaction(), letteralmente "prendi transazione", possiamo leggerlo come "accedi o crea una transazione verso il database". Una *transazione* è un insieme di operazioni (comandi SQL) che saranno eseguite tutte assieme o di cui non ne verrà eseguita nessuna ("all or nothing").

Vediamo un esempio. Supponiamo di voler salvare un prodotto (una riga) con le sue recensioni (n righe). Questo richiederebbe n+1 insert. Supponiamo che la insert della riga del prodotto vada bene, ma che una di quelle delle recensioni fallisca. Senza l'uso di una transazione (transaction) avremmo avuto l'illusione di salvare il prodotto, ma in realtà questo sarebbe stato salvato solo in parte.

Se avessimo pacchettizzato il salvataggio in una transazione invece le scritture sarebbero state "atomiche". Nel caso in cui una qualunque non fosse andata a buon fine, le altre sarebbero state annullate ("rollback della transaction", in gergo) e il programma avrebbe avuto un'eccezione.

Noi sceglieremo di lavorare sempre con le transazioni (o dentro le transazioni) di modo da salvare correttamente gli oggetti o da avere una eccezione da gestire nel caso in cui questo non fosse possibile. Faremo sempre getTransaction().begin() ("iniziare" la transaction, per quanto non sia accurato al 100%) e commit(). commit è un termine tecnico: significa "scrivi". persist() si limita a schedulare la scrittura, ma è commit() che scrive davvero.

Se togliamo l'ultimo commit() dal programma sopra il signor John Smith resterà un prode quarantenne, altrimenti avrà 41 anni. Queste regole valgono anche per le operazioni di rimozione:

```

EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("CensusJPA");
EntityManager em = emfactory.createEntityManager();
Person p = (Person) em.find(Person.class, 5);
em.getTransaction().begin();
em.remove(p);
em.getTransaction().commit();

```

Siamo in grado di modificare questo comportamento, ma per i nostri scopi lavoreremo sempre e solo dentro le transazioni, come è il caso comune. A partire da questi pochi elementi, è facile scrivere un DAO minimo in JPA, sfruttando l'EntityManager:

```

public class PersonDAOJPA
{
    EntityManager em;
    public PersonDAOJPA(EntityManager em)
    {
        this.em = em;
    }
    public Person save(Person p)
    {
        em.getTransaction().begin();
        em.persist(p);
        em.getTransaction().commit();
    }
    public Person load(int id)
    {
        //cosa restituirà se non trova l'id?
        //verificare...
        return (Person) em.find(Person.class,id);
    }
    public Person remove(Person p)
    {
        em.getTransaction().begin();
        em.remove(p);
        em.getTransaction().commit();
    }
}

```

Mancano solo le operazioni di caricamento di gruppo (list()), che vedremo in seguito parlando di JPQL.

Prima di proseguire è obbligatorio segnalare che questo è solo uno dei tanti modi di usare EclipseLink per la permanenza. L'argomento meriterebbe un libro apposito e verranno forniti rimandi in webografia.

13.4 Mappatura uno a molti e politica di propagazione.

JPA permette di definire non solo le singole entità, ma anche i rapporti fra le stesse, e questo si rispecchia nelle operazioni di caricamento e scrittura secondo le regole di ORM che abbiamo visto in precedenza.

Ipotizziamo un database piuttosto semplice, con solo due tabelle, Product e Review, legate da una relazione 1-n. La tabella Product ha i campi id, name, category e price. La tabella Review ha id, score, content e productid, facente da chiave esterna verso Product. Questa struttura può essere tradotta variamente in Java. La più brutale è la seguente:

```
public class Product { int id; String name,category; int price; List < Review > reviews; // getter, setter, costruttori... } public class Review { int id; int score; String content; int productid; // getter, setter, costruttori, metodi }
```

Ma questa traduzione (mappatura) non si usa praticamente mai. La recensione non sa neanche veramente a quale prodotto appartiene. Conosce il suo id, ma non ha un riferimento all'oggetto. Ricordiamolo: *un DB SQL ha le chiavi esterne, Java ha riferimenti a oggetto*. Invece potremmo avere una di queste due mappature:

```
// primo caso: relazione 1-n con navigabilità bidirezionale. Il prodotto conosce le sue recensioni, le recensioni conoscono il prodotto cui afferiscono  
public class Product { int id; String name,category; int price; List < Review > reviews; // getter, setter, costruttori... } public class Review { int id;  
int score; String content; Product product; // getter, setter, costruttori, metodi } // secondo caso: relazione 1-n con navigabilità unidirezionale. Il  
prodotto conosce le proprie recensioni ma le recensioni non conoscono il proprio prodotto  
public class Product { int id; String name,category; int price; List < Review > reviews; // getter, setter, costruttori... } public class Review { int id; int score; String content; //manca qualunque  
riferimento al prodotto // getter, setter, costruttori, metodi }
```

La navigabilità è una scelta da fare in fase di progettazione. Ci interessa che la recensione possa accedere al proprio prodotto o è superfluo? Se partiremo sempre dal prodotto, non serve che la recensione abbia un riferimento a un prodotto. Altrimenti avremo un campo di tipo Product in Review.

Studiamo il primo caso in JPA:

```
// rapporto 1-n con navigabilità bidirezionale in JPA  
// codice generato automaticamente dal wizard di Eclipse  
// e leggermente modificato  
// Product.java, l'entity Product  
package model;  
import java.io.Serializable;  
import javax.persistence.*;  
import java.util.List;  
import java.util.ArrayList;  
/**  
 * The persistent class for the product database table.  
 */  
@Entity  
@NamedQuery(name="Product.findAll", query="SELECT p FROM Product p")  
public class Product implements Serializable  
{  
    private static final long serialVersionUID = 1L;  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int id;  
    private String category;  
    private String name;  
    private int price;  
    //bi-directional many-to-one association to Review  
    @OneToMany(mappedBy="product")  
    private List <Review> reviews = new ArrayList <Review> ();  
    public Product()  
    {}  
    public int getId()  
    {  
        return this.id;  
    }  
    public void setId(int id)  
    {  
        this.id = id;  
    }  
    public String getCategory()  
    {  
        return this.category;  
    }  
    public void setCategory(String category)  
    {  
        this.category = category;  
    }
```

```

public String getName()
{
    return this.name;
}
public void setName(String name)
{
    this.name = name;
}
public int getPrice()
{
    return this.price;
}
public void setPrice(int price)
{
    this.price = price;
}
public List <Review> getReviews()
{
    return this.reviews;
}
public void setReviews(List <Review> reviews)
{
    this.reviews = reviews;
}
public Review addReview(Review review)
{
    getReviews().add(review);
    review.setProduct(this);
    return review;
}
public Review removeReview(Review review)
{
    getReviews().remove(review);
    review.setProduct(null);
    return review;
}
}

// Review.java, lentity Review
package model;
import java.io.Serializable;
import javax.persistence.*;

/**
 * The persistent class for the review database table.
 */
@Entity
@NamedQuery(name="Review.findAll", query="SELECT r FROM Review r")
public class Review implements Serializable
{
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String content;
    private int score;
    //bi-directional many-to-one association to Product
    @ManyToOne
    @JoinColumn(name="productid")
    private Product product;
    public Review()
    {
    }
    public int getId()
    {
        return this.id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
    public String getContent()
    {
        return this.content;
    }
    public void setContent(String content)
    {
        this.content = content;
    }
    public int getScore()

```

```

    {
        return this.score;
    }
    public void setScore(int score)
    {
        this.score = score;
    }
    public Product getProduct()
    {
        return this.product;
    }
    public void setProduct(Product product)
    {
        this.product = product;
    }
}

```

Ci sono due annotazioni nuove di cui tenere conto, senza le quali JPA non funzionerà. La prima è @OneToMany in Product, e si legge "Un Prodotto, Molte Recensioni". E' scritta sopra la proprietà reviews, di tipo List < Review >, e quindi la possiamo anche leggere come "a un Product collega una lista di Review, caricate dal db".

La seconda è @ManyToOne in Review, che specifica che molte Review saranno collegate a un prodotto, e che la review saprà automaticamente a quale prodotto è collegata tramite il campo Product. Ma cosa vuol dire automaticamente?

Supponiamo di avere nel db un prodotto (un videogame, "Super Java Fighter V", con id=1), con due recensioni, punteggi 4 e 5 rispettivamente, e leggiamo questo codice:

```

EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("CensusJPA");
EntityManager em = emfactory.createEntityManager();
Product p = (Product) em.find(Product.class, 1);

System.out.println(p.getName());
for(Review r:p.getReviews())
    System.out.println(r.getScore());
//OUTPUT:
//Super Java Fighter V
//4
//5

```

Caricare una entity con JPA significa caricare anche tutte le entities associate. Abbiamo esplicitato una relazione fra Product e Review tramite le due annotation viste sopra, e JPA è stato abbastanza furbo da caricare le righe corrispondenti alle recensioni e da collegarle al prodotto: ORM automatico.

Ora supponiamo di voler salvare:

```

Product p = new Product();
p.setName("Super Mortal Kart");
p.setCategory("Videogame");
p.setPrice(6);
Review r = new Review();
r.setScore(5);
r.setContent("Nice kart game");
// collego la review al suo prodotto...
// consiglio di andare a leggere il metodo
p.addReview(r);
em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();

```

Questo codice genererà una eccezione: During synchronization a new object was found through a relationship that was not marked cascade PERSIST: model.Review@471a9022. Questo perché JPA richiede di specificare una *politica di propagazione* per le entità correlate. In poche parole, dobbiamo specificare se vogliamo che l'entità Review venga salvata o meno quando salviamo un Product. La maniera più rapida di risolvere la questione è la seguente:

```

// modificare in Product.java
@OneToMany(mappedBy="product", cascade=CascadeType.PERSIST)
private List reviews = new ArrayList();

```

Con questo codice stiamo impostando la politica di cascade per la relazione fra Product e Review, vale a dire quali azioni vogliamo propagare. Cosa facciamo quando persistiamo Product? Persistiamo anche le Review. Cascade significa letteralmente "propagare". Qui stiamo dicendo "propaga il salvataggio (persist) da padre a figlio": se salvo Product salvo anche le sue Review. Ci sono diverse altre possibilità, e una delle più naturali è "ALL": propaga qualunque tipo di operazione (inclusa la rimozione) da padre a figlio.

Per approfondire rimandiamo a questo link: <https://www.baeldung.com/jpa-cascade-types>, ma per i nostri scopi tipicamente sarà sufficiente specificare PERSIST.

13.5 Mappatura di entità padre ed entità figlio

JPA dispone di diverse annotazioni per mappare in maniera automatica il rapporto padre-figlio, che producono esiti diversi in Java. Partiamo da un esempio minimo: una Person potrebbe essere un Teacher (con un campo Subject) o uno Student (con un campo average, inteso come media dei voti). Come abbiamo visto questo si traduce in relazioni 1-1 opzionali. In JPA, possiamo semplificare il lavoro aggiungendo un campo DTYPE, di tipo varchar, alla classe padre (in questo caso Person) prima di creare le altre due tabelle.

```
ALTER TABLE PERSON ADD COLUMN DTYPE VARCHAR(100);
CREATE TABLE STUDENT
(
    ID INT PRIMARY KEY,
    AVERAGE INT,
    FOREIGN KEY(ID) REFERENCES PERSON(ID)
);

CREATE TABLE TEACHER
(
    ID INT PRIMARY KEY,
    SUBJECT VARCHAR(100),
    FOREIGN KEY(ID) REFERENCES PERSON(ID)
);
//E ORA POPOLIAMO LE TABELLE
INSERT INTO PERSON VALUES
(1, 'JOHN', 'SMITH', 41, 'TEACHER'),
(2, 'JAMES', 'WATSON', 20, 'STUDENT');

INSERT INTO STUDENT VALUES
(2, 9);

INSERT INTO TEACHER VALUES
(1, 'MATH');
```

Il signor Smith è un insegnante, il signor Watson uno studente. La colonna DTYPE serve a JPA a capire il "tipo" della riga, e prende il nome di discriminante. Leggendo "Teacher" nella colonna DTYPE alla prima riga JPA cercherà un collegamento nella tabella Teacher, e le assemblerà, a patto di avere le annotazioni corrette. Noi utilizzeremo la tecnica della joined inheritance, letteralmente l'unione delle righe in un singolo oggetto:

```
package model;
import java.io.Serializable;
import javax.persistence.*;

/*Person.java, la classe padre. Contiene dtype ed è l'unica ad avere ancora un id. Conterrà l'id e tutti i campi comuni alle classi figlie.
 */
/* The persistent class for the person database table.
 */
@Entity
@NamedQuery(name="Person.findAll", query="SELECT p FROM Person p")
@Inheritance(strategy = InheritanceType.JOINED)
public class Person implements Serializable
{
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private int age;
    private String name;
    private String surname;
    private String dtype;
    public String getDtype()
    {
        return dtype;
    }
    public void setDtype(String dtype)
    {
        this.dtype = dtype;
    }
    public Person()
    {
    }
    public int getId()
    {
        return this.id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
}
```

```

    }
    public int getAge()
    {
        return this.age;
    }
    public void setAge(int age)
    {
        this.age = age;
    }
    public String getName()
    {
        return this.name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public String getSurname()
    {
        return this.surname;
    }
    public void setSurname(String surname)
    {
        this.surname = surname;
    }
}

// Student.java, una classe figlia. Non ha un id, lo eredita dalla classe padre (Person), così come i campi comuni (name, surname,age). Deve estendere Person
package model;

import java.io.Serializable;
import javax.persistence.*;

/**
 * The persistent class for the student database table.
 */
@Entity
public class Student extends Person implements Serializable
{
    private static final long serialVersionUID = 1L;
    private int average;
    public Student()
    {
    }
    public int getAverage()
    {
        return this.average;
    }
    public void setAverage(int average)
    {
        this.average = average;
    }
}
// Teacher.java. Come sopra. Contiene solo ciò che non è in Person.
package model;

import java.io.Serializable;
import javax.persistence.*;

@Entity
@NamedQuery(name="Teacher.findAll", query="SELECT t FROM Teacher t")
public class Teacher extends Person implements Serializable
{
    private static final long serialVersionUID = 1L;
    private String subject;
    public Teacher()
    {
    }
    public String getSubject()
    {
        return this.subject;
    }
    public void setSubject(String subject)
    {
        this.subject = subject;
    }
}

```

Imposte queste annotazioni, caricare un Teacher non è diverso dal caricare una qualunque person:

```
EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("CensusJPA");
EntityManager em = emfactory.createEntityManager();
Teacher t = (Teacher) em.find(Teacher.class, 1);
System.out.println(t.getName()+" "+t.getSubject());
```

Sarà JPA a preoccuparsi di fondere le righe per noi, come accadeva già nel caso dei rapporti 1-n, di composizione. Occorre notare che questo è solo uno dei modi di codificare l'ereditarietà, probabilmente quello più affine alle nostre procedure. Per approfondire rimandiamo a questo link: <https://www.baeldung.com/hibernate-inheritance>

Prima di proseguire, notiamo che possiamo essere polimorfi con queste annotazioni:

```
EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("CensusJPA");
EntityManager em = emfactory.createEntityManager();
Person p = em.find(Person.class, 1);
System.out.println(p.getClass().getSimpleName());
// "Teacher". Caricare la Person con id=1 è sempre caricare un teacher
```

13.6 Operazioni di base con JPQL

JPQL (Java Persistence Query Language) è il linguaggio di interrogazione proprio di JPA. Presenta delle somiglianze superficiali con SQL, ma la logica di funzionamento sottostante è molto diversa.

JPQL nasce per lavorare su oggetti, non su tabelle. Le sue query restituiscono liste di oggetti, e per essere più precisi liste di entità mappate, complete di entità correlate. Tenendo conto dell'esempio precedente (Product e Review), vediamo come sarebbe caricare tutti i prodotti:

```
SELECT P FROM PRODUCT P;
```

Si può leggere come "seleziona tutti gli oggetti p appartenenti alla classe Product del nostro db". Notate come io stia parlando di oggetti e classi, non di tabelle e righe. In questo caso la "classe" fa davvero da contenitore per i suoi oggetti: JPA traduce le righe del DB in una "scatola" di oggetti che prendono il nome della classe. Da quella scatola stiamo selezionando tutti i p, dove p è il nome generico che diamo a un oggetto di quella classe.

Notiamo che bisogna obbligatoriamente specificare il nome dell'oggetto dopo il nome della classe (NON è una tabella).

Per JPQL non abbiamo una workbench, ma dobbiamo lavorare in Java:

```
List <Product> res = (List <Product>) em.createQuery("SELECT p FROM Product p").getResultList();
```

```
//Il toString di tutti i product  
for(Product x:res) System.out.println(x);  
// il punteggio della prima recensione. Avendo caricato un prodotto, ed essendo il prodotto mappato 1-n con le recensioni, caricare un prodotto  
vuol dire caricare anche le sue recensioni System.out.println(res.get(0).getReviews().get(0).getScore());
```

Piuttosto intuitivo. JPA, come abbiamo visto negli esempi precedenti, si occupa di fare ORM al posto nostro e di collegare gli oggetti che andavano collegati fra di loro.

Valgono alcune delle clausole a cui siamo abituati:

```
List <Product> res = em.createQuery("SELECT p FROM Product p where p.category='IT'").getResultList();
```

Avendo l'accortezza di specificare l'alias di oggetto su cui vogliamo applicare la condizione. Notate che con JPQL è naturale restituire oggetti interi, mentre non è altrettanto facile restituire campi singoli: questo perché JPA avrebbe difficoltà a definire il tipo dell'elemento della lista da restituire se selezionassimo, ad esempio, solo nome e prezzo del prodotto. List di cosa ? List < String, Integer > è una oscenità. Come regola generale restituiamo oggetti *interi* con le query JPQL. Ci sono maniere per superare questo limite ma non le vedremo in questa sede.

Vale ovviamente il discorso relativo alle query parametriche:

```
TypedQuery <Product> query = em.createQuery("SELECT p FROM Product p WHERE p.category = :cat" , Product.class);  
String c = "IT";  
List <Product> productsIT = query.setParameter("cat", c).getResultList();
```

Per terminare, possiamo memorizzare delle query usate più di frequente in quelle che vengono dette NamedQuery. Sono specificate dentro le classi mappate, con questa forma:

```
@Entity  
@NamedQuery(name="Person.findAll", query="SELECT p FROM Person p")  
public class Person implements Serializable {  
{  
    // ometto classe  
}  
  
// per usarla invece:  
List <Person> res = (List <Person>) em.createNamedQuery("Person.findAll").getResultList();
```

"Person.findAll" è soltanto un alias per quella query, memorizzato in una annotation e salvato dentro la classe. Lo usiamo come abbiamo usato le String static final fino ad ora.

13.7 Una business logic da completare in JPA

Visti questi elementi, è semplice scrivere una business logic per gestire prodotti e recensioni. Intendiamo con business logic qualcosa di più della semplice permanenza: sensu strictu la business logic è l'insieme delle procedure che regolano il salvataggio, il caricamento, il trasferimento delle informazioni e il calcolo dei risultati. Non solo il come vengono salvati i dati, non solo le loro relazioni, ma anche il cosa possiamo ragionevolmente salvare, trattare, caricare, muovere e quindi tutta una serie di concern (responsabilità) più avanzate.

Alcuni metodi non sono ancora presenti nelle entità e vengono lasciati da implementare allo studente come esercizio, così come i vari import e la gestione delle eccezioni.

```
public class ShopBL
{
    public ShopBL(EntityManager em)
    {
        this.em = em;
    }
    public Product save(Product p)
    {
        if(!p.isValid())
            throw new RuntimeException("Invalid product");
        // p potrebbe essere Teacher o Student
        em.getTransaction().begin();
        em.persist(p);
        em.getTransaction().commit();
    }
    public Product load(int id)
    {
        return (Product) em.find(Product.class, id);
    }
    public boolean delete(int id)
    {
        Product p = load(id);
        if(p==null) return false;
        em.getTransaction().begin();
        em.remove(p);
        em.getTransaction().commit();
        return true;
    }
    public List <Product> list()
    {
        // usando una named query
        return (List <Product>) em.createNamedQuery("Product.findAll").getResultList();
    }
}
```

13.8 Approfondimenti

In aggiunta ai link visti in precedenza, consigliamo l'ottimo corso di Tutorial Points: <https://www.tutorialspoint.com/jpa/index.htm>, che approfondisce anche alcune problematiche di configurazione e la struttura sottostante JPA.

Per chi volesse studiare anche Hibernate, che è una implementazione estremamente diffusa e potente di JPA, consigliamo anche la guida relativa: <https://www.tutorialspoint.com/hibernate/index.htm>

14 - Design Pattern

14.1 Concetto di Pattern

Un pattern è un "motivo ricorrente", una soluzione astratta e riproducibile a un problema concreto, o anche una "metodica di lavoro".

Il concetto nasce in architettura, in cui determinate strutture e motivi si ripetono in maniera abbastanza naturale, ed è stato popolarizzato per l'architettura del software da un famoso libro della Gang of Four (https://en.wikipedia.org/wiki/Design_Patterns). Gang of Four è il nome informale dato al gruppo composto da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, e questo loro lavoro in particolare ha avuto una eco notevole sullo sviluppo del software orientato agli oggetti.

I pattern emergono, almeno in teoria, dal ripresentarsi di problemi pratici a cui si tende a dare una stessa soluzione. La soluzione, ripetuta nel tempo, astratta e migliorata, diventa un pattern, e potrà essere riutilizzata per un caso concreto successivo.

Nella pratica i pattern vengono a volte "forzati" per garantire una uniformità di progetto, e richiedono una certa aderenza alla lettera anche laddove i benefici non siano immediatamente evidenti. Questo è particolarmente vero per progetti grandi, dove l'aderenza a una struttura ben nota comporta enormi risparmi di gestione del progetto via via che questo passa di mano.

Un programma progettato secondo pattern comuni e ben noti è un programma che può essere gestito da terze parti, ampliato e migliorato senza essere legati alla singola risorsa. Questo ha un'importanza fondamentale per progetti che, necessariamente, coinvolgono centinaia di persone e periodi di sviluppo di anni.

Quando parliamo di pattern possiamo parlare dei 23 pattern definiti nel libro della Gang of Four (i 23 "well-known" patterns), o di tutta una serie di introduzioni successive, variamente formalizzate, che sono diventate pattern de facto. C'è anche un elemento interpretativo in alcuni pattern, con discussioni feroci su cosa costituisca una applicazione "valida" di un pattern e cosa no.

Per il nostro lavoro saremo strettamente pragmatici, presentando i pattern per come vengono usati nella pratica, riferendoci al libro originale dove possibile e segnalando dove ci discostiamo dalla versione ufficiale per presentare qualcosa di più recente.

I pattern hanno senso principalmente nell'ambito della programmazione a oggetti (in effetti nascono anche per rimediare ad alcune scomodità della programmazione a oggetti), e sono tradizionalmente divisi in creazionali (relativi alla creazione degli oggetti), strutturali (relativi al modo in cui oggetti pre-esistenti sono legati fra loro da rapporti statici), comportamentali (che specificano le interazioni fra oggetti) e architetturali (che riguardano la struttura del sistema più in generale, non limitandosi ai singoli oggetti ma definendo responsabilità per ogni parte). Si tratta comunque di accademia e i confini sono spesso labili.

14.2 Factory

Cominciamo con un pattern estremamente diffuso ma che non è veramente presente nel libro originario. Il pattern Factory può essere visto come una "semplificazione" del pattern Factory Method presente nel libro originale.

L'idea di base di questo pattern è quella di disporre di una "fabbrica" di oggetti. Noi diremo alla fabbrica di cosa abbiamo bisogno (un tipo formale), e lei si preoccuperà di creare un oggetto di un tipo adatto ai nostri bisogni. A noi (il Client) non deve interessare il tipo concreto, ma solo quello astratto. La scelta del tipo concreto è compito della factory, così come tutti i dettagli di creazione dello stesso.

L'essenza del pattern è quindi separare il tipo concreto da creare dal tipo formale richiesto.

Supponiamo di avere bisogno di una connessione a un database, come abbiamo visto nel capitolo precedente relativamente a JDBC. Sappiamo che il codice per crearla è il seguente:

```
Connection connection=DriverManager.getConnection(  
"jdbc:mysql://localhost:3306/db","user","pwd");
```

DriverManager è una classe, getConnection è un metodo che funziona da factory. E' letteralmente una fabbrica di connessioni.

Notiamo che Connection è una interfaccia di JDBC, quindi l'oggetto creato di sicuro non sarà solo di tipo connection, ma di una sua qualche implementazione. A noi non interessa il tipo concreto dell'oggetto che andremo a creare, e non ci interessa in che modo verrà creato. Potrebbe anche non essere un oggetto nuovo, ma uno già usato in precedenza, facendo risparmiare memoria e tempi di calcolo al sistema. Noi abbiamo esplicitato un bisogno, o per essere più precisi una *dipendenza*, e qualcuno (getConnection) ci ha fornito il pezzo di cui avevamo bisogno.

Questo pattern è quindi diviso in tre parti:

- il **Client** (in questo caso, chi invoca getConnection), che ha bisogno di un qualche oggetto e ne chiede la creazione alla factory.
 - il Product, che è l'oggetto che vogliamo creare. Il client specifica il tipo formale (Connection, in questo caso), mentre la Factory ne specifica quello concreto (dovendolo creare).
- Il tipo formale del Product potrebbe essere una classe concreta, una classe astratta, una interfaccia... non ci sono vincoli, ma spesso si preferisce definire quello di cui abbiamo bisogno come interfacce, quindi spesso il tipo formale è una interfaccia (come come Connection).
- la Factory, che viene invocata dal Client per produrre il Product. La Factory potrebbe essere un oggetto o una classe (come in questo caso, DriverManager). Nella pratica, ci sarà un metodo accessibile a cui noi passeremo dei parametri. Questo metodo farà la scelta per noi e si occuperà del lavoro di creazione vero e proprio.

In realtà abbiamo già visto delle Factory, nel momento in cui il programma doveva decidere che tipo di oggetto creare a partire dalla lettura dal file (o dal database). Questo ci ha garantito il vantaggio della centralizzazione, ma non solo: una Factory può restituire un oggetto già esistente (vedere il lavoro fatto con le Date), o anche null, o una eccezione personalizzata.

Praticamente tutti i package in commercio forniscono i propri oggetti tramite Factory. JDBC ha factory per creare Connection, Statement e ResultSet (interfacce, tutte e tre, come abbiamo visto). Questo ci permette anche di controllare quali oggetti vengono creati concretamente, e potenzialmente anche di cambiare quello che verrà prodotto senza che il resto del mondo se ne accorga.

Vediamo un esempio concreto. Supponiamo di avere due componenti per l'accesso ai dati (DAO, un pattern anch'esso, come abbiamo visto prima). Entrambi i DAO si occupano di persone, ed entrambi implementano PersonDAO. Questi DAO sono PersonDAOMySQL e PersonDAOMYSQLLogged.

Il secondo DAO fa lo stesso lavoro del primo (implementano la stessa interfaccia) ma aggiunge una funzione di logging delle richieste, che possono poi essere passate alle forze dell'ordine o a motori di reportistica.

Potremmo volere il DAO "basile" o quello loggato, a seconda del contesto in cui ci troviamo. Ad esempio, tutte le modifiche alle persone di un database con dati sensibili (giudiziari, medici) dovrebbero essere tracciate, mentre non è così importante per la gestione di una squadra hobbistica di calcetto.

Vediamo una soluzione con una PersonDAOFactory:

```
public abstract class PersonDAOFactory  
{  
    public static PersonDAO make(String type, Connection connection)  
    {  
        if(type.equalsIgnoreCase("basic"))  
            return new PersonDAOMySQL(connection);  
        if(type.equalsIgnoreCase("logged"))  
            return new PersonDAOMYSQLLogged(connection);  
        return null;  
    }  
    public static PersonDAO make(Connection connection)  
    {  
        return make("basic", connection);  
    }  
}
```

// e lo potremmo usare in questo modo:

```
PersonDAO dao = PersonDAOFactory.make(connection);
```

```
// o così  
PersonDAO dao = PersonDAOFactory.make("logged", connection);
```

La factory creerà un oggetto sulla base delle nostre richieste (basic per un DAO di base, logged per uno loggato, null altrimenti). Se non specifichiamo un tipo (secondo metodo, overload di un metodo statico) deciderà lei per noi, creando un DAO standard (senza logging).

E se volessimo cambiare il comportamento standard? Ci basterebbe cambiare il secondo metodo, scrivendo "logged" (o qualunque altra cosa) al posto di logged. E dovendo cambiare la logica di creazione potremo sempre farlo qui, dentro la factory. In questo caso il nostro Product (formale) era PersonDAO, mentre i due Product concreti erano PersonDAOMySQL e PersonDAOMSQLLogged rispettivamente.

Alcuni programmatore ritengono addirittura "sbagliato" l'utilizzo di new, e lo ammettono solo all'interno delle factory, per essere sicuri di mantenere la separazione fra tipo formale (servizio richiesto) e tipo concreto (fornitore del servizio). Può essere una posizione estrema, ma non è del tutto sbagliata e la struttura che avete appena visto lo evidenzia.

14.3 Singleton

Singleton è uno dei 23 pattern "ben noti", ed è presentato nella sua versione originale, che ha resistito bene al passare del tempo. L'idea di base alla base del singleton è che di alcune classi ci serve al massimo una istanza.

Possiamo decidere che ci serve una sola istanza, un solo oggetto, per varie ragioni. La più comune è un principio che conosciamo già, DRY, don't repeat yourself. Semplicemente non ci serve due volte il codice contenuto in quell'oggetto, e non vogliamo che venga creato una seconda volta.

In un linguaggio procedurale o ad approccio misto questo sarebbe stato risolto con una libreria di funzioni, simili a quelle che abbiamo visto in precedenza, ma Java non ha il concetto di funzione (dispone delle interfacce funzionali, non delle funzioni), e in generale l'approccio OOP preferisce parlare di oggetti, non di funzioni.

Un'altra possibilità è quella di una classe con metodi statici, eventualmente astratta. E' il caso della classe Math della libreria standard di Java, che contiene una serie di metodi che sono in realtà funzioni travestite, e in effetti Math è una libreria di funzioni travestita da classe. La soluzione più elegante tuttavia è quella di avere una classe istanziabile da 0 a 1 volte, e questo si ottiene tramite l'uso del pattern Singleton, che è estremamente diffuso e richiesto sul lavoro.

A prescindere da queste riflessioni che riguardano le limitazioni di architettura del linguaggio, ci sono casi in cui la logica pretende di avere un solo componente di un dato tipo. Pensiamo a un registro delle presenze: potessimo averne due questi potrebbero andare in conflitto. Uno studente potrebbe risultare presente per un registro ed assente per un altro. In questo caso è opportuno che esista una e una sola copia dell'oggetto.

Identificato cosa vogliamo, e perché, ora dobbiamo capire il come ottenerlo, vale a dire la struttura del pattern singleton. Il primo passo è identificare la classe che dovrà essere singleton, supponiamo un log di sistema, e rendere il suo costruttore privato:

```
public class SystemLog
{
    private SystemLog()
    {
    }
}
```

Il costruttore del singleton è sempre privato e zero-arguments (niente argomenti). In questo modo l'oggetto potrà essere creato solo dall'interno della classe.

A questo punto si procede a definire le funzionalità offerte dal nostro oggetto:

```
public class SystemLog
{
    private List events = new ArrayList();
    public void addEvent(String event)
    {
        events.add(event);
    }
    public List getEvents(String key)
    {
        List res = new ArrayList();
        for(String s:events)
            if(s.contains(key))
                res.add(s);
        return res;
    }
    public List getEvents()
    {
        return getEvents("");
    }
    private SystemLog()
    {
    }
}
```

Abbiamo un log di eventi (stringhe) con funzioni di inserimento e di ricerca basili. Notiamo che i vari metodi (addEvent e i due getEvents) sono di oggetto, non di classe. Apparterranno all'evento di classe SystemLog, non alla classe, e per ora non possono essere usati. Non c'è modo per il sistema fuori di creare un SystemLog, essendo il costruttore privato.

Procediamo a creare l'oggetto all'interno della classe (l'unico posto in cui è possibile farlo), e un metodo per renderlo disponibile. Sia l'oggetto che il metodo sono statici: appartengono alla classe.

```
public class SystemLog
{
```

```

private List events = new ArrayList();
private static SystemLog instance = new SystemLog();
public static SystemLog getInstance()
{
    return instance;
}
public void addEvent(String event)
{
    events.add(event);
}
public List <String> getEvents(String key)
{
    List <String> res = new ArrayList <String>();
    for(String s:events)
        if(s.contains(key))
            res.add(s);
    return res;
}
public List <String> getEvents()
{
    return getEvents("");
}
private SystemLog()
{
}
}

```

La classe SystemLog possiede l'unico oggetto di tipo SystemLog, e lo fornisce al mondo tramite il metodo getInstance. Il resto del mondo accederà al log in questo modo:

```
SystemLog.getInstance().addEvent("E' successo qualcosa");
```

Analizzandolo col consueto gioco dei tipi, SystemLog è la classe SystemLog, getInstance() è l'unico oggetto possibile e addEvent è il metodo di quell'evento.

Notiamo che il mondo esterno non ha idea del fatto che SystemLog sia un singleton (può intuirlo, non saperlo con sicurezza). Il mondo esterno sapeva di avere bisogno di un log, e gli è stato fornito.

Qualunque processo che vada a richiedere un SystemLog finirà per ottenere lo stesso oggetto, quello creato dalla classe e nella classe. L'oggetto potrà a questo punto essere passato come parametro ad un metodo (utile in molti casi), potremo creare dei riferimenti (SystemLog s = SystemLog.getInstance()), e così via.

C'è un altro elemento da considerare: e se non ci servisse mai il SystemLog? In questo modo lo abbiamo comunque creato, spendendo tempo di calcolo e risorse, per quanto poche. L'ideale sarebbe crearlo solo laddove fosse necessario.

Questo si ottiene con un metodo noto come lazy initialization:

```

public class SystemLog
{
    private List <String> events = new ArrayList<String>();
    private static SystemLog instance = null;
    public synchronized static SystemLog getInstance()
    {
        if(instance==null)
            instance = new SystemLog();
        return instance;
    }
    public void addEvent(String event)
    {
        events.add(event);
    }
    public List <String> getEvents(String key)
    {
        List<String> res = new ArrayList();
        for(String s:events)
            if(s.contains(key))
                res.add(s);
        return res;
    }
    public List <String> getEvents()
    {
        return getEvents("");
    }
    private SystemLog()
    {
    }
}

```

}

instance non viene creata subito, ma solo al momento della richiesta con getInstance(). Invocando getInstance() viene fatto un controllo: esiste già l'oggetto? Se non esiste lo creiamo e lo assegniamo al nostro riferimento interno. In tutti i casi, lo restituiamo.

Occorre fermarsi un attimo su quel "synchronized". synchronized indica che il metodo è accessibile a un solo processo per volta, che lo richiama, ottiene il risultato e poi lo rilascia. Questo non è il caso tipico in Java: i metodi tipicamente possono essere eseguiti in parallelo da diversi processi, ma non vogliamo farlo in questo caso.

Cosa succederebbe se chiamassimo getInstance() "allo stesso istante" da due processi diversi? Entrambi potrebbero valutare la condizione if e trovarla falsa, e quindi procedere a creare due volte l'oggetto, nullificando il nostro lavoro. Questa è detta rush condition (più o meno "incidente stradale", come due auto che si tocchino a un incrocio). synchronized evita questo problema.

Ricapitolando, i punti chiave del singleton sono:

- costruttore privato
- unico oggetto esistente statico e privato
- metodo getInstance statico e pubblico, per ottenere accesso all'unico oggetto esistente

Partendo da questi si può rendere singleton qualunque classe con lo stesso procedimento meccanico. Le factory sono spesso singleton, così come tutti quegli oggetti che contengono solamente codice e non stato ("librerie travestite").

14.4 Adapter

Nella pratica ci si trova spesso a scrivere varianti di componenti scritte in precedenza. Magari un nuovo problema da risolvere è molto simile a uno risolto in precedenza, o magari possiamo adattare un vecchio algoritmo per gestire un nuovo caso.

La prima tentazione è di andare a modificare il codice originale, aggiungendo nuovi casi ai metodi per includere le nuove necessità, o nuovi metodi alle classi. Questo non è sempre auspicabile, perché ogni modifica a un componente già testato richiede di eseguire nuovamente i test e può portare a instabilità di sistema e a comportamenti non richiesti.

Non solo: a furia di modifiche, aggiunte e correzioni un componente che era solito svolgere un compito x e che ora ne svolte una dozzina diversi, più o meno collegati ad x, rischia di essere in massima parte inutilizzato e difficile da gestire e da manutenere, e violerebbe il principio di responsabilità, per cui ogni oggetto o classe è responsabile di un solo compito.

In generale, preferiamo scrivere un nuovo componente. Allo stesso tempo, sarebbe sciocco fare copia-incolla del vecchio. La soluzione pratica è quella di scrivere un adapter, un adattatore, un nuovo componente che risolva il nuovo problema tramite l'utilizzo di un vecchio componente.

Diamone una definizione formale. Ipotizziamo di avere bisogno di un nuovo servizio avente una interfaccia I2. I2 è ottenibile partendo da I1, una vecchia interfaccia già implementata da una classe C.

Ci comporteremo come segue:

- Creeremo una classe A ("adapter", adattatore) che implementerà I2 ("nuova interfaccia, servizio desiderato") e conterrà un oggetto di classe C ("adattato"), che a sua volta implementa I1 ("vecchia interfaccia, servizio già esistente")
- Un oggetto di classe A userà l'oggetto di classe C al suo interno per implementare i metodi di I2.

Informalmente, A traduce I1 in I2, utilizzando il componente C che implementa I1. In mancanza di una interfaccia I1 esplicita, potete immaginare che A "cambi interfaccia" a un componente, trasformando i suoi servizi in qualcosa di più adatto alle nostre esigenze. Lo scopo finale è arrivare a I2 partendo da qualcosa di già esistente.

Questa tecnica viene detta "adapter", ed è un pattern strutturale classico definito del libro della GoF. Offre diversi vantaggi: in primis non dobbiamo modificare il componente originale. Poi, qualunque modifica migliorativa al componente originale si riflette sull'adattatore. Per finire, possiamo scrivere quanti adattatori vogliamo del componente originale senza appesantirlo o renderne complicata la gestione.

Vediamo un esempio concreto. Ci siamo trovati spesso a usare gli oggetti di classe Scanner per il nostro lavoro, e spesso abbiamo dovuto leggere numeri interi. Scanner dispone di un nextInt(), ma noi vorremmo qualcosa di più raffinato: la possibilità di richiedere un numero intero compreso fra due cifre c1 e c2, con tanto di messaggio di errore e ripetizione della richiesta fino ad ottenere un valore corretto, e già che ci siamo anche una istruzione che stampi un messaggio prima di richiedere una stringa.

In questo caso Scanner ha il ruolo di I1 e di C. Esprime implicitamente la nostra vecchia interfaccia (in particolare, useremo il metodo nextLine() definito nella classe Scanner). Noi vogliamo arrivare ad avere questa interfaccia:

```
// I2
public interface IScanner2
{
    int readInt(String msg, String errMsg, int lower, int higher);
    String readLine(String msg);
}
```

Il modo più naturale per arrivarci è tramite il componente che già abbiamo (Scanner):

```
//Adapter da Scanner a IScanner2 ("A" nel pattern)
//Implementa I2
public class Scanner2 implements IScanner2
{
    //C" nel pattern: adaptee, adattato
    //implementerebbe I1, se avessimo una data interfaccia sotto Scanner
    Scanner oldcomponent;
    public Scanner2(Scanner scanner)
    {
        this.oldcomponent = scanner;
    }
    public int readInt(String msg, String errMsg, int lower, int higher)
    {
        int res = lower-1;
        do
        {
```

```

        System.out.println(msg);
    try
    {
        res = Integer.parseInt(oldcomponent.nextLine());
        if(reshigher)
            System.out.println(errMsg);
    }
    catch(NumberFormatException e)
    {
        System.out.println(errMsg);
    }
}
while(reshigher);
return res;
}
public String readLine(String msg)
{
    System.out.println(msg);
    return oldcomponent.nextLine();
}
}

```

Abbiamo automatizzato due operazioni di uso comune, ponendo dei "controlli in ingresso". Il nuovo componente (`IAdapter`, `Scanner2`) implementa la nuova interfaccia (`Scanner2`), basandosi su un componente (`oldcomponent`) che implementa la vecchia interfaccia ("Scanner", per quanto in questo caso sia una classe e non una interfaccia separata). Per concludere, il punto chiave che vogliamo ricordare è che il nuovo componente contiene il vecchio e lo usa per fornire un nuovo servizio che onora un nuovo contratto.

14.5 Proxy

Nel caso dell'Adapter, la vecchia interfaccia non era più adatta ai nostri scopi e ne abbiamo ricavato una nuova a partire da una vecchia. Nel caso del Proxy, un secondo pattern strutturale classico, la vecchia interfaccia ci va ancora benissimo, ma vogliamo cambiare il come verrà implementata, aggiungendo funzionalità ai metodi già presenti senza cambiare il contratto.

Prendiamo il caso di PersonDAOMySQL. Supponiamo, come abbiamo detto prima, di avere bisogno di aggiungere funzioni di logging alle interrogazioni. Non ci serve cambiare il contratto dei metodi: le operazioni di lettura, di lista e di scrittura rimangono identiche. Semplicemente, vogliamo che vengano registrate in maniera completamente trasparente.

Non avrebbe senso riscrivere il componente PersonDAOMySQL, nè modificarlo, per le stesse ragioni di cui abbiamo parlato per Adapter. Analogamente a quanto fatto prima, useremo il vecchio componente (PersonDAOMySQL, che implementa PersonDAO), per ricavare quello nuovo. Notiamo che in questo caso sia il vecchio componente che quello nuovo implementano la stessa interfaccia (differenza chiave con Adapter). Il nuovo componente prende il nome di proxy ("intermediario") perchè si pone come intermediario fra il vecchio componente e il mondo:

```
//"Proxy"
public class PersonDAOMySQLLogged implements PersonDAO
{
    //Vecchio componente da "proxare"
    PersonDAOMySQL old;
    public List <String> requests = new ArrayList<String>();
    public PersonDAOMYSQLLogged(PersonDAOMYSQL old)
    {
        //come prima, il vecchio componente
        //è una dipendenza del nuovo
        this.old = old;
    }
    @Override
    public Person load(int id)
    {
        requests.add("Requested data for person "+id);
        return old.load(id);
    }
    @Override
    public Person save(Person p)
    {
        requests.add("Requested save operation for person "+p);
        return old.save(id);
    }
    @Override
    public List <Person> list(String criteria)
    {
        requests.add("Requested filter for people with criteria "+criteria);
        return old.list(criteria);
    }
    @Override
    public boolean delete(int id)
    {
        requests.add("Requested deletion for person "+id);
        return old.delete(id);
    }
}
```

Abbiamo aggiunto funzionalità senza variare il contratto. Il vecchio componente continua a fare il suo lavoro, e noi (il proxy) facciamo il resto. Il punto chiave nell'utilizzo del pattern proxy è che il proxy dovrebbe essere, almeno in teoria, intercambiabile con l'oggetto originale:

```
PersonDAO dao = new PersonDAOMYSQL(connection);
PersonDAO dao = new PersonDAOMYSQLLogged(new PersonDAOMYSQL(connection));
```

In questo caso funziona fino a un certo punto: per vedere il log che viene prodotto siamo costretti a castare, e questo non è piacevole. Supponiamo invece di voler scrivere un proxy per filtrare determinate richieste, mandandole "a vuoto" senza bisogno di toccare il database:

```
://"Proxy"
public class FilteredPersonDAOMYSQL implements PersonDAO
{
    //Vecchio componente da "proxare"
    PersonDAOMYSQL old;
    public FilteredPersonDAOMYSQL(PersonDAOMYSQL old)
    {
        //come prima, il vecchio componente
        //è una dipendenza del nuovo
```

```

        this.old = old;
    }
    @Override
    public Person load(int id)
    {
        if(id <= 0) return null;
        return old.load(id);
    }
    @Override
    public Person save(Person p)
    {
        if(!p.isValid()) return null;
        return old.save(id);
    }
    @Override
    public List <Person> list(String criteria)
    {
        return old.list(criteria);
    }
    @Override
    public boolean delete(int id)
    {
        if(id<=0) return false;
        return old.delete(id);
    }
}

```

Questo è un uso più in linea con lo spirito del pattern. Possiamo avere diversi FilteredDAO, con logiche di filtro diverse per impedire al mondo esterno di "disturbare" il componente se non è strettamente necessario, come del resto è tipico nel caso dei componenti che usano il database e che di conseguenza fanno perdere millisecondi preziosi all'esecuzione.

Il proxy viene spesso utilizzato per aggiungere funzioni di logging, di controllo e in generale di ottimizzazione contestuale su di un componente già esistente. Possiamo avere diversi proxy e scegliere di usare uno o l'altro a seconda delle necessità.

14.6 Facade

Possiamo vedere una Facade, il nostro terzo pattern strutturale, come una super-interfaccia, una "copertura" per un numero arbitrario di interfacce che devono spesso essere usate assieme per produrre un qualche risultato di interesse, e il cui uso risulta scomodo, ripetitivo o comunque indesiderabile.

Vediamo un caso concreto: la lettura di una riga da un database tramite JDBC:

```
...
Statement s = connection.createStatement();
ResultSet rs = s.executeQuery("select * from Person");
...
```

Ci troviamo a dover usare tre interfacce (Connection, Statement, ResultSet) per ottenere qualcosa che dovrebbe essere "naturale", una riga. Per altro la riga è anche restituita sotto forma di qualcosa di "inusuale", un recordset. Sarebbe molto più comodo avere a disposizione una mappa (< String, String >, volendo trattare tutti i campi come stringhe) o liste di mappe per identificare più righe.

Lo stesso vale per la scrittura. Non sarebbe molto più semplice scrivere qualcosa del tipo "database.execute(sql)"?

In effetti, le interfacce di JDBC sono spesso "coperte" con una facciata che ne automatizza le interazioni. Ci troviamo spesso a usare assieme Connection, Statement e ResultSet, e il tutto per ricavare poche informazioni per volta.

Per evitarlo, cominciamo definendo l'interfaccia a cui vogliamo arrivare per gestire il database, la super-interfaccia, la facade. Questi sono i metodi che vorremo utilizzare in pratica, e sotto cui troveremo JDBC.

Lo studente attento potrebbe vedere una somiglianza col pattern Adapter, che in effetti c'è. La differenza fra Facade ed Adapter è che Facade copre diverse interfacce (non solo una), che vengono tipicamente utilizzate assieme per fornire un servizio. In questo senso non è la traduzione di un solo componente a un'altra interfaccia, ma di molti componenti correlati che vengono "coperti" assieme.

Tornando al nostro esempio, quale potrebbe essere una "buona" interfaccia per parlare col db?

```
//Facade per Connection, ResultSet e Statement
public interface IDatabase
{
    // restituisce una lista di righe sotto forma di mappe
    // il nome della colonna è la chiave della mappa, il valore del campo è il suo valore associato. Anche i campi numerici e di tipo data vengono presi come stringa per semplicità e poi adattati nel caso
    List <Map <String, String>> rows ( String sql) throws SQLException;
    default Map <String, String> row ( String sql) throws SQLException
    {
        List <Map <String, String>> rows = rows(sql);
        return rows.size() == 1 ? rows.get(0) : null;
    }
    boolean execute (String sql) throws SQLException;
}
```

Una interfaccia tutto sommato semplice, con solo tre metodi: leggere un elenco di righe, leggere una singola riga (un caso particolare dell'elenco, quindi ridotto a metodo di default) e un metodo per eseguire una query di modifica (execute).

A questo punto vediamo di implementarlo, utilizzando le interfacce di JDBC. Notiamo che nessuna delle interfacce viene passata come dipendenza (a differenza di quanto accade con Adapter). Vengono create e gestite dentro la Facade, e il mondo esterno non si rende conto del loro funzionamento.

```
public class Database implements IDatabase
{
    Connection connection;
    public Database(String conn) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        connection = DriverManager.getConnection(conn);
    }
    @Override
    public List<Map <String , String>> rows(String sql, String[] parameters) throws SQLException
    {
        List <Map <String, String>> res = new ArrayList <Map <String, String>>();
        PreparedStatement ps = connection.prepareStatement(sql);
        for(int i=0;i<parameters.length;i++)
            ps.setString(i+1, parameters[i]);
        ResultSet rs = ps.executeQuery();
        while(rs.next())
        {
            Map <String, String> row = new LinkedHashMap <String, String>();
            for(int i=0;i <rs.getMetaData().getColumnCount();i++)

```

```

        row.put
        (
            rs.getMetaData().getColumnLabel(i+1).toLowerCase(),
            rs.getString(i+1)
        );
        res.add(row);
    }
    rs.close();
    return res;
}
@Override
public void execute(String sql, String[] parameters) throws SQLException
{
    PreparedStatement ps = connection.prepareStatement(sql);
    for(int i=0;i<parameters.length;i++)
        ps.setString(i+1, parameters[i]);
    ps.execute();
}
}

```

A questo punto l'utilizzo è abbastanza banale:

```

IDatabase db = new Database(connectionString);
Map <String, String> personData = database.row("select * from Person where id=1");
System.out.println(personData.get("name"));

```

Il client non si rende conto di stare usando JDBC, e lavora solo con interfacce e componenti familiari (mappe e liste, first class citizens in Java). Se invece dovesse passare una connessione non si parlerebbe più di facade.

La facade semplifica il lavoro, velocizza i compiti comuni e riduce le dipendenze. Il client dipende solo da IDatabase, non, formalmente, da JDBC. Quando volessimo cambiare JDBC e utilizzare qualcosa di diverso per accedere al database sarebbe sufficiente modificare un solo componente (Database.java) non tutto il resto del sistema.

14.7 Model View Controller (MVC)

Model View Controller è un pattern architetturale che definisce le parti di un sistema in termini di responsabilità e interazioni fra le parti. E' di livello più alto rispetto ai pattern strutturali, in quanto non definisce il rapporto fra singole classi o oggetti, ma fra intere sezioni del sistema.

MVC è soggetto a interpretazioni (ci sono antiche polemiche relative all'appartenza a questo pattern di questo o quel framework). Noi ne presentiamo una abbastanza basilare, semplificata e adatta alla programmazione web.

Un sistema può essere diviso in tre parti:

- **Il Model.**

Il Model è la parte del sistema che si occupa di fornire un modello della realtà, di "contenere" i dati e di implementare la logica di calcolo, di produrre i risultati che poi forniremo all'utente.

Fanno parte del model le entities, i dao e in generale tutte le classi che lavorano "nell'ombra" per fare calcoli e permanenza. Una regola importante del model è che non è pensato per produrre stampe. I dati del model sono "amorfi", memorizzati e prodotti in forma grezza per poi poter essere formattati e presentati diversamente a seconda delle circostanze (ad esempio, tradotti in diverse lingue). Lo potete ricordare come "dati e logica di calcolo".

- **La View.**

La view, o "le viste", essendo questa sezione del programma tipicamente composta di diverse interfacce grafiche, ha come responsabilità quella di presentare i dati che le arrivano dal model e di ricevere i comandi da parte dell'utente.

Possiamo pensare alla vista come all'interfaccia grafica del programma, che riceve gli input e stampa i risultati prodotti dal model. La vista non può in alcun modo memorizzare i risultati o fare calcoli che non siano estremamente basilari o afferenti alla visualizzazione. Possiamo dire che la vista è "logica di presentazione": cosa visualizzare, come visualizzarlo, quali input richiedere per poi passarli al controller, che è la terza parte del nostro pattern.

- **Il Controller.**

Il Controller collega vista e model. Riceve i comandi che arrivano dalla vista (ad esempio un input da tastiera o un click su un pulsante) e li trasforma in comandi per aggiornare il model (i dati, il database) e la stessa vista.

Un esempio chiarirà tutto: l'utente clicca sul pulsante "carica dati". Il segnale arriva dalla vista al controller. Il controller invoca un metodo del model e carica i dati di una persona. Questi dati vengono poi ripassati alla vista che li grafica.

Possiamo pensare al controller come "logica di comportamento", "cosa faccio quando succede questo?", e anche come tramite fra vista e model. Il nostro "main" faceva da controller e da view nella maggior parte dei primi lavori.

Seguiremo poche regole per collocarci nell'ambito di un MVC sufficientemente "rigoroso". Un primo passo sarà dividere i nostri packages in tre super-packages: model, view e controller. In qualche caso avremo anche un package "libraries", ma tipicamente si preferisce mettere tutto in model.

Il package model conterrà le entities, i dao, eventuali facade per l'accesso al db (come la nostra IDatabase) e in generale tutto quanto lavora direttamente sul dato per fare dei calcoli.

Le entities non saranno pensate per produrre stampe. I vari toString hanno senso solo in fase di debug.

Idealmente, se abbiamo la entity Person, avremo PersonDAO per la permanenza e poco altro in model.

Il package view conterrà le classi che graficheranno il dato e produrranno le interfacce grafiche per l'utente. Idealmente, se abbiamo una entity Person, avremo PersonView, una classe o interfaccia pensata per graficare le persone. Vediamo un esempio:

```
public interface PersonView
{
    String render(Person person);
    default String render( List <Person> list)
    {
        String res = "";
        for(Person p:list)
            res+=render(p)+"\n";
        return res;
    }
}

public class PersonViewItaly implements PersonView
{
    @Override
    public String render(Person person)
    {
        return person.getGender().equals("M") ? "Signor "+person.getName() : "Signora / ina "+person.getName();
    }
}

public class PersonViewUSA implements PersonView
{
    @Override
    public String render(Person person)
    {
        return person.getGender().equals("M") ? "Mr "+person.getName() : "Mrs "+person.getName();
    }
}
```

L'esempio è minimo, ma rende l'idea. Lo stessa entità (model) può avere più rappresentazioni, e in effetti non è insolito creare le viste tramite factory, per rispondere a un'esigenza o a un'altra. In questo caso parliamo di localizzazione, ma immaginiamo di voler esportare i dati in formato CSV o XML:

```
public class PersonViewCSV implements PersonView
{
    @Override
    public String render(Person person)
    {
        return person.getName() + "," + person.getSurname() + "," + person.getGender();
    }
}

public class PersonViewXML implements PersonView
{
    @Override
    public String render(Person person)
    {
        return "" + person.getName() + "" + person.getSurname() + "" + person.getGender() + "";
    }
    // sono costretto a fare override. XML pretende di avere un elemento radice.
    @Override
    public String render( List <Person> list)
    {
        String res = "";
        for(Person p:list)
            res+=render(p)+"\n";
        res+="";
        return res;
    }
}
```

Viviamo nel mondo di Babele, delle mille lingue e dei mille formati. E' sempre opportuno separare il dato dalla sua rappresentazione, di modo da poterlo adattare successivamente alle nostre necessità. Lo vedremo bene con un esempio concreto con un controller, in questo caso rappresentato da un pezzo del main:

```
PersonDAO persondao = PersonDAOFatory.getInstance().make(connection);

System.out.println("INSERT CMD");
cmd = keyboard.nextLine();
switch(cmd)
{
    case "EXPORTXML":
        res = PersonViewFactory.getInstance().make("XML").render(personDAO.list());
        break;
    case "EXPORTCSV":
        res = PersonViewFactory.getInstance().make("CSV").render(personDAO.list());
        break;
    default:
        res = "BAD COMMAND";
}
System.out.println(res);
```

Questo esempio combina diversi pattern. Le factory sono tutte singleton (esiste al massimo una factory per ogni tipologia di oggetto da creare). Le PersonViewFactory restituiscono la PersonView corretta, scegliendo per noi il tipo concreto (il lavoro della factory), su cui chiamiamo il metodo render, passandogli la lista degli oggetti da renderizzare, presi dal DAO (il Model).

Il controller, come dicevamo, mette in contatto model e vista, che sono per loro natura separati completamente, e risponde ai comandi dell'utente (in questo caso letti da tastiera). Vedremo una applicazione specifica e molto diffusa di questo pattern nello sviluppo web.

15 - Il linguaggio HTML

15.1 Introduzione ad HTML

HTML è l'acronimo di Hyper Text MarkUp Language, letteralmente linguaggio a "mark up" per ipertesti.

Un ipertesto è un testo contenente elementi multimediali (immagini, suoni, filmati) e interattivi (pulsanti, link, caselle di testo), vale a dire quella che quasi tutti conosciamo come pagina web.

Un documento HTML è un documento scritto rispettando le regole del linguaggio HTML. Un documento di questo tipo viene renderizzato dal browser, che trasforma il codice in quello che l'utente vedrà e con cui andrà a interagire. HTML nasce per la trasmissione di dati scientifici e per la definizione formale di documenti. Non nasce per costruire esperienze grafiche accattivanti, né per adattarsi agli schermi dei cellulari o dei tablet, né per creare applicazioni. In effetti HTML non è un linguaggio di programmazione ma un linguaggio di definizione documentale che è stato adattato a svolgere altri compiti. Per i nostri scopi, HTML sarà il linguaggio con cui scriveremo le interfacce grafiche per le applicazioni, integrandolo eventualmente con CSS, che vedremo in seguito.

15.2 Basi del linguaggio

Un documento HTML è un file di testo costituito da elementi annidati.

Un *elemento*, o tag, è definito tramite la sintassi `<e> contenuto </e>`, oppure `<e />` per gli elementi privi di un contenuto. La parte viene detta apertura del tag. La parte `</e>` viene detta chiusura del tag. Tutto ciò che viene aperto deve essere chiuso. Per gli elementi privi di contenuto la chiusura è contestuale all'apertura, e viene detta chiusura veloce: `<e />`.

Un elemento può avere degli attributi, che all'interno di un elemento non dovrebbero ripetersi. Gli attributi sono scritti come : l'elemento e ha gli attributi `a` e `b`, con valori `1` e `2` rispettivamente. Un elemento può contenere qualunque attributo, ma solo alcuni hanno senso all'interno di un dato elemento. Gli attributi di un elemento sono una mappa con chiave String e valore String.

Forniamo di seguito tre esempi: Un paragrafo di testo: `<p> Io sono un elemento paragrafo. Ho un contenuto. </p>`. L'elemento è `p`, non ha attributi ma ha un contenuto.

Una immagine: ``. Questo tag non ha contenuto ma ha l'attributo `src` con valore "foto.jpg".

Una casella di testo in cui l'utente potrà scrivere: `<input type="text" name="test" />`. L'elemento è `input` e ha due attributi, `type` e `name`, rispettivamente con valori `text` e `test`, e non ha un contenuto. Notiamo che un elemento può essere scritto indifferentemente con le lettere maiuscole o minuscole o anche mischiandole. Per chiarezza e comodità, si preferisce il minuscolo.

Un documento HTML è composto di elementi (tag) annidati. Alcuni elementi si possono ripetere, mentre altri dovrebbero essere presenti solo una volta. La struttura basilare di un documento prevede un unico tag (detto anche elemento radice, nodo radice o radice) che contiene tutti gli altri, e questo tag è HTML e non si deve ripetere. HTML deve contenere due tag, rispettivamente HEAD e BODY. BODY conterrà quasi tutta la parte visibile del documento, mentre HEAD conterrà informazioni di configurazione e import.

In codice, scriviamo: `
`

```
<html>
  <head>
    <title> La mia prima pagina </title>
  </head>
  <body>
    Io sono una pagina HTML.
  </body>
</html>
```

In gergo diciamo che `html` è il nodo radice. `head` e `body` sono figli di `html`. `title` è figlio di `head`, e nipote di `html`. `html` contiene direttamente `head` e `body`, e indirettamente `title`. `title` contiene un testo ("la prima pagina") che verrà visualizzato (renderizzato) nella barra del titolo del browser. "Io sono una pagina HTML" verrà visualizzato nel corpo del browser, su sfondo bianco.

Il rapporto fra gli elementi è un rapporto di composizione: un elemento è composto di altri elementi e di attributi, oltre che del proprio contenuto. Possiamo vedere questa struttura come scatole che contengono altre scatole (alla stessa maniera dei blocchi di codice) o come un albero, e formerà la base della nostra interfaccia grafica.

15.3 Un primo esempio

Per noi HTML è uno strumento per la creazione di interfacce grafiche per programmi Java. Di conseguenza non studieremo tutti i tag possibili, ma solo quelli di nostro interesse. Riporto di seguito un esempio con una serie di elementi e di attributi di uso comune per il lavoro di programmazione:

```
<html>
    <head>
        <title> Ticket Calculator </title>
    </head>
    <body>
        <h1> Museum Entrance Ticket Calculator </h1>
        <p>
            Questa pagina web permetterà di calcolare e di registrare
            il costo del biglietto di ingresso per il nostro museo, a partire dall'età del cliente
            e da eventuali esenzioni. <br />
            Il tag p indica un paragrafo. Il tag br indica un ritorno a capo. br è senza contenuto,
            mentre p tendenzialmente lo ha.
            Un paragrafo è circondato da linee che servono per separarlo dagli altri paragrafi.
        </p>
        <!--
            form significa letteralmente "modulo", e la sintassi
            che state leggendo è un commento in HTML.
        -->
        <form method="get" action="TicketCalculator">
            Nome del cliente
            <input type="text" name="clientname" />
            Età
            <input type="number" name="age" />
            Dichiara di essere un donatore del sangue
            <select name="donor">
                <option value="N"> No </option>
                <option value="Y"> Sì </option>
            </select>
            <input type="submit" value="invia" />
            <a href="http://www.ilnostromuseo.it">
                Clicca qui per visitare il sito del museo
            </a>
        </form>
    </body>
</html>
```

Questo esempio, pur semplice, introduce una serie di elementi ricorrenti nello sviluppo di interfacce applicative per web applications, vale a dire utilizzabili tramite browser. Vale la pena di analizzarlo punto per punto.

15.4 h1

I tag h* sono gli header, e modificano lo stile del testo ingrandendolo. Si parte da h1, il più grande, scalando verso h2, h3 e così via (il limite è h6). I tag h avevano un significato grafico maggiore in html prima di CSS (Cascading Style Sheet, che vedremo in seguito, e che è un linguaggio che serve a migliorare la grafica delle pagine HTML, inizialmente non pensate per essere attraenti), mentre oggi hanno un significato semantico.

I motori di ricerca cercano i tag H1, H2 ed H3 per capire di cosa parla una pagina, e in effetti presentano spesso collegamenti alle varie sezioni. H1 serve come titolo principale della pagina, e si preferisce averne solo uno per documento (uno per pagina), h2 per le sezioni principali, h3 per le sottosezioni in h2, e così via. Tipicamente non si scende sotto h3.

15.5 p

p indica un paragrafo in HTML, e come specificato sopra isola una porzione di testo e immagini. Nella pratica lo si utilizza principalmente per il testo, mentre per creare "zone" della pagina contenenti testo e immagini (e tipicamente molti p) si utilizza il tag "div", divisorio, letteralmente "un'area generica" della pagina.

In HTML come in ogni altro documento di testo è il paragrafo l'unità fondamentale da considerare, ed è a livello di paragrafo che vengono definiti bordi, spazi, rientri, caratteri e così via. Lo vedremo bene in seguito con CSS.

15.6 form

Form si traduce etteralmente con "modulo".

E' un tag pensato per contenere una collezione di elementi interattivi (caselle di testo, pulsanti, checkboxes, liste di selezione). L'utente interagirà con gli elementi interattivi (detti anche "controlli") terminando tipicamente con la pressione di un tasto "invia".

Il contenuto inserito dall'utente verrà quindi inviato a un programma che si occuperà di processarlo. Si tratta del meccanismo alla base delle web applications, e verrà dettagliato nel capitolo sulle servlet.

Per adesso notiamo un enigmatico attributo method, che chiariremo parlando di HTTP, e un attributo action. action corrisponde al nome del programma che verrà chiamato a gestire i dati inseriti dall'utente.

15.7 input

Gli elementi (o tag) di tipo input sono in realtà una famiglia che viene renderizzata in maniera diversa a seconda dell'attributo type. input type="text" viene tradotto in una casella di testo a inserimento libero. input type="number", che non è riconosciuto da tutti i browser, genera una casella di testo che permette l'inserimento solo di numeri. input type="submit" genera un pulsante che invia i dati della form (in questo caso il nome del cliente e la sua età, oltre alla dichiarazione relativa al suo essere donatore o meno) al programma definito alla voce "action" del tag form che contiene input type=submit .

Questo è uno dei punti chiave da ricordare. Il tag input type="submit" /> comporta l'invio, ma è la form a decidere a chi andranno i dati, tramite l'attributo action.

Per gli input, come anche per molti altri elementi interattivi, ha molta importanza l'attributo name. La form contiene sia del testo che altri tag, e ogni controllo inserito dentro la form verrà tradotto in una variabile (una stringa, sempre e comunque) che verrà inviata al programma specificato dall'attributo action della form.

I nomi delle variabili corrispondono ai valori degli attributi name dei controlli della form: nel nostro caso, il programma di gestione (TicketCalculator) riceverà clientname, age e donor, stringhe in tutti e tre i casi (anche age). Queste variabili vengono anche dette parametri.

15.8 select

Il tag select è un controllo, affine alla famiglia input, che permette all'utente di scegliere fra un numero predefinito di alternative (in questo caso, sì o no) tramite una lista. Il tag select contiene a sua volta dei tag option, e ogni option corrisponde a una possibile risposta alla "domanda" posta sopra la lista. Notiamo che il nome della variabile inviata viene specificato tramite l'attributo name della select (name="donor"), mentre il valore inviato viene specificato tramite l'attributo value della option.

A questo punto ci fermiamo per una domanda. Quali dati verranno inviati al programma TicketCalculator supponendo che il biglietto sia per il signor John, di 40 anni, non donatore? I dati saranno, nella consueta notazione insiemistica usata anche per le mappe, i seguenti: {clientname:"John", age:"40", donor:"N"}.

Notiamo che la variabile donor vale "N", non "no". Il contenuto di una option è il testo visualizzato dall'utente, ma il valore inviato non è il contenuto della option selezionata, ma il suo attribute value.

Notiamo anche come gli attributi abbiano senso solo all'interno di elementi che li supportino. Ad esempio, avrebbe poco senso scrivere p type="text", essendo p un paragrafo e non un controllo (vale a dire un elemento di input).

15.9 a

Ultimo ma estremamente importante come elemento interattivo, A viene usato (anche) per definire dei link, vale a dire dei collegamenti verso altre parti dello stesso sito o della stessa applicazione. Un link è un elemento A con un attributo href.

href specifica l'indirizzo verso cui navigare (potrebbe trattarsi in realtà di una web app da eseguire, e tipicamente lo è, come vedremo fra poco), mentre il contenuto specifica quello che verrà visualizzato dall'utente, e sui cui l'utente potrà cliccare per andare al sito o all'applicazione desiderata. Vediamo un esempio più concreto:

```
<a href="http://www.generation-italy.com">
    
    Clicca qui per andare al portale di Generation
</a>
```

Il contenuto del link è composto da due righe. La prima riga crea una immagine, che essendo interna a un link diventa cliccabile. La seconda presenta del testo normale. Che si clicchi sul testo o sull'immagine, il comportamento dell'applicazione sarà lo stesso: il browser navigherà verso il sito <http://www.generation-italy.com>. In gergo tecnico diremo che è stata effettuata una request HTTP, che è quanto accade premendo su input type="submit", ma lo vedremo meglio in seguito.

Controlli e link sono i principali elementi interattivi di una pagina Web. E' possibile rendere interattivo ogni elemento con altre tecnologie che vedremo in seguito, ma quanto visto sopra costituisce la parte maggiore del nostro lavoro.

15.10 Approfondimenti

HTML vale un approfondimento oltre questi pochi elementi e oltre quello che vedremo in seguito. Consigliamo la guida di W3Schools all'indirizzo <https://www.w3schools.com/html/>. Segnaliamo inoltre che, a differenza di quanto non accada con Java, un documento HTML scritto male, o tecnicamente parlando "malformato", con attributi ripetuti, elementi non chiusi o attributi posti all'interno di elementi che non li supportano, non verrà rifiutato dal browser. Il browser proverà comunque a renderizzarlo, a volte anche producendo risultati apprezzabili.

Per lo sviluppatore questo è un problema, in quanto l'errore non è immediatamente evidente e interi pezzi di pagina possono "scomparire" perché qualcuno si è scordato di chiudere le virgolette doppie.

16 - CSS

16.1 Introduzione a CSS

CSS è l'acronimo di Cascading Style Sheet, letteralmente "foglio di stile a cascata". Si tratta di un linguaggio nato per specificare al meglio la grafica di una pagina web HTML, che come abbiamo visto in precedenza non era nato per questo scopo.

Mentre HTML definisce struttura e contenuto di una pagina web, CSS definisce il modo in cui questi verranno renderizzati dal browser - sfondi, colori, allineamento, colore e font del testo, rientri, margini e imbottiture ma anche visibilità o allineamento degli elementi stessi nella pagina. E' anche importante notare che questi potranno cambiare automaticamente al variare del dispositivo usato per accedere alla stessa: un cellulare vedrà una pagina diversa rispetto a un tablet che la vedrà una pagina diversa rispetto a un PC, in maniera automatica, se così verrà deciso dagli author CSS.

Occorre notare che HTML fornisce già degli strumenti per fare della grafica elementare, ma sono considerati in buona parte deprecati. h1 è un esempio di questo tipo, ma sarebbe possibile ottenere un minimo di grafica utilizzando dei tag pensati a tale scopo come ad esempio font, che permette di specificare dimensione e colore del testo tramite gli attributi size e color rispettivamente (font size="20" color="red"> Testo rosso con corpo di 20 px), oppure tramite tag effettivamente ancora usati per la propria semplicità, come b (bold,grassetto) o i (italic).

Questa sovrapposizione di compiti, il poter ottenere lo stesso effetto tramite HTML o CSS, inquadra i momenti diversi dell'evoluzione del mezzo, il passaggio da strumento scientifico con grafica elementare a strumenti per la scrittura di prodotti mediatici di massa (siti web). In effetti lo sviluppo web in generale è una somma quasi geologica di strati, alcuni nati allo scopo, altri adattati.

Come prassi, noi cercheremo di usare sempre e solo CSS per la grafica, e HTML per la struttura, con pochissime eccezioni.

16.2 Basi del linguaggio e uso di CSS in HTML

CSS può essere scritto dentro un file HTML o scritto in un file CSS autonomo e collegato ad HTML tramite un apposito tag di collegamento, come vedremo in seguito. Ha una sintassi propria completamente diversa da quella di HTML, ma che serve a governare uno o più pezzi della pagina. Di conseguenza, parlare di CSS senza parlare di HTML è impossibile.

In generale, CSS attribuisce regole di stile alla pagina. Dice al browser in quale maniera renderizzare uno o più elementi. Sarà più chiaro con un esempio concreto:

```
<html>
  <head>
    <title> La mia prima pagina </title>
    <style>
      BODY
      {
        color:red;
        font-size:30px;
      }
    </style>
  </head>
  <body>
    Io sono una pagina HTML.
  </body>
</html>
```

Il blocco di codice all'interno del tag style è espresso in CSS. E' composta da un selettore (in questo caso il nome del tag body) e da un insieme di regole di stile. La possiamo leggere come "prendi tutti gli elementi BODY, che siano uno o diecimila, e attribuisci loro due regole: il testo deve essere rosso (color:red) e la dimensione del carattere deve essere 30 px (pixel)".

In questo caso parliamo di CSS innestato dentro HTML, un pezzo di codice scritto in un linguaggio alieno rispetto a quello della struttura generale. Sarà il caso anche con Javascript, che vedremo in seguito. Noi sappiamo, dal modulo precedente, che ci sarà un solo tag BODY, ma per CSS è indifferente. A meno che non venga specificato esplicitamente, non tiene conto della cardinalità.

Vediamo un secondo esempio:

```
<html>
  <head>
    <title> La mia prima pagina </title>
    <style>
      P
      {
        font-size:30px;
        color:red;
      }

      B
      {
        color:green;
      }
    </style>
  </head>
  <body>
    <p>
      Io sono un paragrafo. Il mio testo sarà rosso, salvo
      quello che metto in grassetto, che sarà <b>verde</b>,
      essendo maggiormente specifico.
    </p>
  </body>
</html>
```

Approfondiamo la struttura. Il blocco style, che è un tag di HTML il cui contenuto è espresso in CSS, viene tipicamente scritto dentro il tag head, ma non è una regola universale. Possono esserci diversi tag style in una pagina, e le regole possono anche contraddirsi a vicenda.

La logica che porta a decidere quale regola applicare è la seguente: vince la regola più specifica. Nell'esempio sopra abbiamo definito una regola per tutti i paragrafi (p), tale per cui il testo sarà rosso, ma c'è una regola più specifica (relativa cioè a un tag più piccolo, contenuto nel tag p che è grande) che vince in termini di priorità rispetto a quella più generale.

Lo studente è pregato di soprassedere sull'oggettiva bruttezza dell'esempio renderizzato e di concentrarsi solo sulla logica, per ora.

Le cose possono complicarsi notevolmente con l'introduzione di un altro metodo di inserire stile CSS in un file HTML, che viene detto inline-CSS. Possiamo cioè applicare del codice CSS direttamente a un singolo elemento di nostro interesse tramite l'attributo style, che funziona con qualunque elemento visibile (vedremo in seguito che non tutti gli elementi sono pensati per essere visibili).

L'attributo style ha la precedenza rispetto alle regole definite a livello generale nel tag style:

```
<html>
  <head>
    <title> La mia prima pagina </title>
    <style>
      P
      {
        font-size:30px;
        color:red;
      }

      B
      {
        color:green;
      }
    </style>
  </head>
  <body>
    <p>
      Io sono un paragrafo. Il mio testo sarà rosso, salvo
      quello che metto in grassetto, che sarà <b>verde</b>,
      essendo maggiormente specifico. Volendo però potrei
      anche avere un elemento bold e nero, usando l'attributo
      style sul tag b, come sto <b style="color:black"> facendo
      ora </b>
    </p>
  </body>
</html>
```

Tipicamente si preferisce mantenere le informazioni di stile in file separati e poi collegarli ad HTML. Nel caso del terzo esempio, la maniera "corretta" di agire sarebbe creare un file (di nome, ad esempio, style.css), col seguente contenuto:

```
P
{
  font-size:30px;
  color:red;
}

B
{
  color:green;
}
```

E poi collegarlo al file HTML come segue:

```
...
<head>
  <link href='style.css' rel='stylesheet'>
</head>
...
```

In questo caso il file style.css deve essere nella stessa cartella del file .html. Venendo da Java, potreste leggerlo come "import style.css". E' possibile, e tipicamente viene fatto, importare diversi file .css, le cui regole si vanno a sommare e potenzialmente a sovrascrivere a vicenda come abbiamo visto in precedenza.

16.3 Regole di stile per il resto

Per quanto riguarda il testo conviene definire almeno tre informazioni: il font da usare, il colore da usare e la dimensione da usare.

Il font è specificato tramite la regola font-family, tipicamente nella forma font-family:nomefont (e.g.: font-family:Arial). Spesso viene specificato più di un font, perché non è detto che il primo font specificato sia presente sulla macchina che renderizzerà la pagina o che il browser lo supporti. Un esempio classico potrebbe essere font-family:Arial, Helvetica, Sans Serif; .

Il colore è specificato tramite la regola color:nomecolore o color:codicecolore;. Il discorso sui codici dei colori esula dallo scopo di questo testo, ma viene approfondito nei collegamenti. Nella pratica, per i nostri scopi sarà spesso sufficiente scrivere color:black (tipicamente il default) o color:blue; o affini.

Il discorso delle dimensioni merita invece maggiore attenzione. La proprietà per impostare il font è font-size:valore, ma il valore può essere specificato in vari modi.

Il modo più intuitivo è attribuire una grandezza, specificata in una unità che potrebbe essere assoluta o relativa. Questo vale, per altro, per tutte le grandezze che vedremo in seguito (margini, bordi, imbottiture). Esempi di grandezze assolute sono i pixel, i cm, i mm o le inches:

```
font-size:20px;  
font-size:2cm;  
font-size:35mm;  
font-size:1in;
```

Notiamo come abbiamo dovuto specificare il tipo di grandezza di volta in volta.

Le grandezze relative sono invece riferite al contesto in cui si trovano, vale a dire, nel nostro caso, all'elemento che contiene il testo. Le due grandezze relative più usate sono % (letteralmente, percentuale rispetto alla dimensione del mio contenitore) o em (letteralmente, "misura dell'elemento standard del mio contenitore").

Vediamo un esempio:

```
<html>  
    <head>  
        <title> CSS - Testo </title>  
        <style>  
            DIV  
            {  
                font-size:20px;  
            }  
  
            P  
            {  
                font-size:200%;  
            }  
  
            span  
            {  
                font-size:3em;  
            }  
        </style>  
    </head>  
    <body>  
        <div>  
            Testo normale  
            <p>  
                Testo doppio  
            </p>  
            <span>  
                Testo triplo  
            </span>  
        </div>  
    </body>  
</html>
```

SPAN (un altro tipo di contenitore, simile ma non uguale a DIV) è all'interno di DIV, così come P. Per DIV abbiamo specificato una dimensione assoluta. Il resto muta di conseguenza. P ha un testo pari al 200% della dimensione del padre, mentre SPAN pari a tre volte la dimensione del testo standard (quindi, nella pratica, 300%).

Nella pratica, si tende a stabilire una dimensione per il font a livello di body e poi a ingrandirlo o a rimpicciolirlo a seconda delle necessità tramite le misure relative e le regole di composizione che vedremo a breve.

Per chiudere il discorso sul testo menzioniamo la proprietà text-align, che determina l'allineamento del testo nel proprio contenitore. Valori validi sono left (a sinistra, di default), right (a destra), center (centrato, utile per i titoli) o justify (giustificato), che è lo standard de facto per quasi tutti i testi, incluso quello che state leggendo. Concludendo con un esempio:

```
DIV
{
    font-family:Tahoma, Arial;
    font-size:110%;
    color:#222;
    text-align:justify;
}
```

Questa regola modificherà il testo di tutti i DIV della pagina, impostando il font al 110% del font del body (o del contenitore del DIV), il colore a un numero abbastanza scuro ma non assoluto (#222), giustificando il testo all'interno del DIV e cercando di utilizzare il font Tahoma se possibile, Arial altrimenti, e il default di sistema, quale che sia, nel caso in cui i primi due font non fossero disponibili.

16.4 Regole per margini, imbottiture e dimensioni (margin, padding, height, width)

Possiamo immaginare la pagina web come composta di "scatole", molto spesso DIV, ma anche P, SPAN e altri elementi con contenuto. Possiamo immaginare questi elementi come dei rettangoli a schermo, con relativi bordi, potenzialmente (quasi sempre) invisibili.

Rispetto al bordo, riconosciamo due tipologie di distanze: i margini, che separano gli altri elementi dal bordo (spazio all'esterno del bordo) e i padding, che separano il contenuto dell'elemento dal suo bordo (spazio all'interno del bordo). Un esempio chiarirà le cose:

```
<html>
  <head>
    <title> CSS - Margin e Padding </title>
    <style>
      DIV
      {
        margin:40px;
        padding:10px;
        background-color:green;
        color:black;
        border:1px solid black;
      }
    </style>
  </head>
  <body>
    <div>
      Esempio di margin, bordi e padding
    </div>
  </body>
</html>
```

Notiamo che il DIV è molto spaziato rispetto ai bordi della pagina (di 40 px su tutti e quattro i lati, per essere precisi), mentre il testo è spaziato rispetto al bordo del div di molto meno (10px su tutti e quattro i bordi, per essere precisi, per quanto se ne vedano bene solo tre: left, top e bottom, visto che il testo non arriva fino all'estrema destra del div). Il bordo del div è stato specificato come largo 1px (una grandezza assoluta, ma poteva non esserlo), solid (linea continua, ma poteva essere tratteggiata o di altri tipi) black (di colore nero).

Perchè tutta questa attenzione agli spazi? Un DIV, ma anche un P, senza padding presenta il testo attaccato ai bordi, o agli altri elementi della pagina, ed è poco chiaro e difficile da leggere. Allo stesso modo, senza margini gli elementi rischiano di essere attaccati fra di loro, bordo a bordo. Non è il caso dei DIV in generale, e lo vedremo fra poco, ma occorre pianificare in anticipo lasciando dello spazio perchè testo e immagini "respirino".

Nell'esempio sopra c'è una proprietà di cui non abbiamo parlato, ma dal significato evidente: background-color. I colori di sfondo si usano per distinguere aree diverse della pagina, ma conviene non esagerare con la varietà per evitare un effetto patchwork confusionario per l'utente. Notiamo anche che lo sfondo è solo interno al bordo, ed evidenzia il padding, mentre chiaramente non esiste nell'area del margin.

Padding e margin possono essere impostati (e in effetti tipicamente lo sono) lato per lato. Modifichiamo l'esempio di prima:

```
<html>
  <head>
    <title> CSS - Margin e Padding</title>
    <style>
      DIV
      {
        margin:40px;
        padding-left:5px;
        padding-top:7px;
        padding-bottom:7px;
        background-color:green;
        color:black;
        border:1px solid black;
      }
    </style>
  </head>
  <body>
    <div>
      Esempio di margin e padding
    </div>
  </body>
</html>
```

In questo caso il padding destro non è specificato (rimane quello di default), mentre abbiamo padding diversi orizzontalmente e verticalmente, come è spesso il caso. Il margin non è invece stato toccato.

Per terminare il discorso sulle "scatole", possiamo specificare altezza e larghezza degli elementi utilizzando le grandezze che abbiamo visto in precedenza. Se non specifichiamo una larghezza e una altezza esplicita, l'elemento cercherà di accomodare il proprio contenuto, deformandosi di conseguenza.

Questo ci renderà difficile specificare un layout per la pagina, in quanto non sapremo in precedenza quale contenuto andremo a presentare - il contenuto sarà il prodotto di un programma. Non avremo quella che viene detta tecnicamente la conoscenza degli ingombri.

Conviene quindi specificare una dimensione per i div e in generale per gli elementi che comporranno la pagina, tramite le proprietà width ed height, che vengono tipicamente espresse in px o in %. Ad esempio: `div style="height:50%;width:50%"` produrrà un div alto e largo la metà dell'elemento che lo contiene.

Rimandiamo all'esempio pratico successivo per capire come tutto questo servirà per creare interfacce grafiche.

16.5 Regole per l'allineamento e la visibilità degli elementi

Una delle funzioni principali di CSS è modificare l'allineamento degli elementi sulla pagina. Prendiamo il seguente esempio:

```
<html>
  <head>
    <title> Allineamento in CSS </title>
    <style>
      </style>
  </head>
  <body>
    <div> blocco di testo 1 </div>
    <div> blocco di testo 2 </div>
  </body>
</html>
```

I due DIV vengono messi uno sotto l'altro. Se fossero stati due SPAN sarebbero comparsi sulla stessa riga. Questo accade perché i DIV sono di default elementi "display:block", vale a dire che cercano di occupare una propria riga sullo schermo. Gli SPAN sono elementi "display:inline", che cercano di stare sulla stessa riga con gli altri elementi salvo essere costretti ad andare a capo per mancanza di spazio. CSS permette di mutare il comportamento degli elementi, permettendo ad esempio di mettere due DIV uno di fianco all'altro, come segue:

```
<html>
  <head>
    <title> Allineamento in CSS </title>
    <style>
      DIV
      {
        display:inline;
      }
    </style>
  </head>
  <body>
    <div> blocco di testo 1 </div>
    <div> blocco di testo 2 </div>
  </body>
</html>
```

La regola display permette di impostare la logica di allineamento e presentazione di un elemento. A noi interessano quattro possibili valori:

- **display:block**, per indicare elementi che prendono un'intera linea e che non "vogliono coesistere" con altri elementi sulla stessa riga
- **display:inline**, per indicare il caso opposto. Un div con display:inline, per un indicare elementi che cercano di restare sulla stessa linea con gli altri elementi attorno, e le proprietà height e width non gli si applicano
- **display:inline-block**, una via di mezzo fra i primi due, per elementi che cercano di stare in linea ma a cui è possibile attribuire altezze e larghezze fisse
- **display:none**, per elementi che debbano essere completamente rimossi dalla pagina. Potrebbe sembrare strano voler nascondere un elemento dalla pagina, ma è spesso utile, mostrando o nascondendo DIV contenenti messaggi di errore o di navigazione, tramite l'interazione di CSS e di Javascript.

Le possibilità sono molte più di queste, ma non ci servono per gli scopi del corso, e buona parte dei problemi relativi al posizionamento e all'allineamento degli elementi vengono risolti tramite framework o librerie CSS (ad esempio W3.CSS o Bootstrap, che vedremo in seguito).

16.6 Propagazione delle regole

Le regole vengono propagate "dall'alto verso il basso", o "dal contenitore al contenuto", letteralmente "a cascata" ("cascading"). Vediamo un esempio:

```
<div style="color:red">
    Testo rosso
    <div style="font-size:200%">
        Testo rosso grande
    </div>
</div>
```

Il primo div ha come regola di stile "color:red". Il secondo div è interno al primo, ed eredita (inherit) questa informazione dall'elemento padre. In sostanza le regole vengono sommate. In questo caso come negli altri, è la regola più specifica a valere:

```
<div style="color:red">
    Testo rosso
    <div style="font-size:200%;color:green">
        Testo verde grande
    </div>
</div>
```

In questo caso il figlio sovrascrive la regola del padre. Venendo da Java, potremmo dire che abbiamo fatto override della regola.

16.7 Selettori di tag

La potenza di CSS è la capacità di selezionare in maniera granulare un elemento o una famiglia di elementi per applicare loro delle regole di stile. Un selettore può essere qualcosa di estremamente semplice, come BODY (le regole di stile per l'intero corpo) o di estremamente specifico (un singolo elemento, o tutti gli elementi in una determinata posizione dell'albero della pagina).

HTML dispone anche di alcuni elementi specifici per "aiutare" CSS nell'opera di graficazione. Ora andremo ad analizzare alcuni dei selettori più comuni e del loro uso, partendo dai selettori di TAG (o selettori per elemento), che abbiamo già visto e che vanno a colpire gli elementi di quel tipo senza distinzione. Ad esempio:

```
BODY
{
    color:black;
}

P
{
    color:black;
}
```

Il body in generale, e i p in particolare, avranno testo nero. Un modo per sintetizzare potrebbe essere il seguente:

```
BODY, P
{
    color:black;
}
```

La virgola in CSS si legge "disgiunzione", oppure "or" se la vogliamo vedere alla maniera di Java. La regola color:black vale sia per BODY che per P.

16.8 Selettori di classe

Possiamo aggiungere un attributo class agli elementi HTML. Questo ci serve per "categorizzare" gli elementi e permettere a CSS di graficarli per categoria. E' interessante notare che le classi CSS non sono equivalenti alle classi Java. Un elemento HTML può avere un solo attributo class, ma l'attributo class potrebbe contenere riferimenti a diverse classi CSS. Vediamo con un esempio:

```
<html>
  <head>
    <title> Classi CSS </title>
    <style>
      .rosso
      {
        color:red;
      }

      .grande
      {
        font-size:30px;
      }

      p
      {
        color:green;
      }
    </style>
  </head>
  <body>
    <p class="rosso"> Testo rosso </p>
    <p class="rosso grande"> Testo rosso con dimensione 30 px </p>
    <p> Testo verde </p>
  </body>
</html>
```

Abbiamo definito due classi CSS con la sintassi .nomeclasse. Le classi non corrispondono a nessun elemento predefinito di HTML (o almeno non dovrebbero), e possiamo scegliere liberamente i loro nomi. In questo caso abbiamo definito le classi "rosso" e "grande". Abbiamo poi dato uno stile ai paragrafi in generale (sono verdi). Il primo paragrafo appartiene a due classi, anche se venendo da Java sarebbe più facile immaginarlo come implementante due interfacce, o comunque sottoposto a due set di regole diverse.

Notiamo che le classi sono state definite come .nomeclasse, mentre le abbiamo associate all'elemento p come p class="classe1 classe2", senza il punto. Un elemento può appartenere a più classi CSS. Una classe CSS può governare più elementi. La classe "rosso" governa in questo caso due elementi p.

Le classi sono molto usate in CSS, in quanto permettono di dividere gli elementi della pagina in base a categorie o "significati" diversi, a prescindere dal nome dell'elemento. Vedremo che ricoprono un ruolo importante anche in Javascript, e ne approfittiamo per enunciare un principio di sviluppo web: la forma implica la funzione.

Una volta abituato l'utente al fatto che i buttoni, le caselle di testo e il testo hanno un certo aspetto, non bisogna mandarlo in confusione creando buttoni, caselle di testo o blocchi di testo completamente diversi dal resto del sito (come colori, spaziature, caratteri) senza una valida ragione. Una buona interfaccia grafica non ha bisogno di essere spiegata.

La sintassi .nomeclasse permette alla classe di essere applicata a qualunque elemento, ma volendo possiamo restringere la validità della classe a una sola tipologia di elementi. Vediamo l'esempio seguente:

```
<html>
  <head>
    <title> Classi CSS </title>
    <style>
      DIV.rosso
      {
        color:red;
      }

      .grande
      {
        font-size:30px;
      }

      p
      {
        color:green;
      }
    </style>
  </head>
  <body>
    <div class="rosso"> Testo rosso </div>
```

```
<p class="rosso grande"> Testo con dimensione 30 px. Perchè non rosso? </p>
<p> Testo verde </p>
</body>
</html>
```

Il secondo p non è rosso, in quanto la classe è stata definita come DIV.rosso, vale a dire: "applica queste regole agli elementi DIV che contengano la classe rosso". Il secondo elemento p è di classe "rosso", ma non è un DIV. Le regole di classe non vengono applicate.

Ne approfittiamo per rivedere il concetto di DIV: un DIV è un divisorio, vale a dire un'area di pagina che forma concettualmente una "unità" indipendente, o se preferite un "box" da riempire.

Per le classi come per gli elementi è possibile utilizzare la virgola, ad esempio in questo modo:

```
P, .rosso
{
    color:red;
}
```

Tutti i paragrafi (quale che sia la loro classe, anche se non dovessero averla) e tutti gli elementi di classe rosso (quale che sia l'elemento) avranno testo di colore rosso.

16.9 Selettori per ID

E' possibile definire un attributo ID sui tag HTML. ID sta per "identificatore" e dovrebbe essere unico rispetto alla pagina: una pagina non deve avere due elementi con attributo id identico. Gli id servono a indicare un singolo elemento (ad esempio il menù della pagina), il cui valore non si ripeterà altrove, a differenza di altri attributi (come name o class) che possono ripetersi senza difficoltà.

Un ID è comunque un attributo di un elemento, e di conseguenza viene definito così: div id="menu" class="standardtext".

Un'altra sintassi può essere anche la seguente: div id="menu" class="standardtext" style="color:yellow", con classe, id e stile in linea, ma è piuttosto difficile da vedere nella pratica, a meno di non utilizzare particolari framework che incoraggiano lo stile in linea.

In CSS, le regole di id vengono specificate tramite il cancelletto. Riprendendo il codice sopra potremmo avere:

```
<html>
  <head>
    <title> ID in CSS </title>
    <style>
      #menu
      {
        color:green;
        background-color:black;
      }

      .standardtext
      {
        font-family:Arial;
        font-size:20px;
      }
    </style>
  </head>
  <body>
    <div id="menu" class="standardtext"> Menù </div>
  </body>
</html>
```

16.10 Selettori per discendenza

Possiamo selezionare un elemento a partire dai suoi "antenati" nell'albero o più graficamente a partire dalla scatola che li contiene. Ad esempio:

```
P DIV
{
    color:yellow;
}
```

Si legge come "tutti gli elementi di tipo DIV contenuti dentro un elemento di tipo P". In questo caso la composizione è intesa a qualunque livello: DIV potrebbe essere "figlio" diretto di P, nipote, pronipote... è sufficiente che P sia da qualche parte dentro il corpo di un DIV. Scrivendo a-spazio-b sto quindi dicendo "applica queste regole ai b che sono dentro a".

Questo funziona anche per le classi, e può creare notevole confusione. Vediamo questo esempio:

```
<html>
    <head>
        <title> Classi CSS </title>
        <style>
            DIV.rosso
            {
                color:red;
            }

            DIV .rosso
            {
                color:green;
            }
        </style>
    </head>
    <body>
        <div class="rosso"> Testo rosso </div>
        <div> <p class="rosso"> Testo verde </p> </div>
        <div> Testo nero </div>
    </body>
</html>
```

DIV.rosso e DIV-spazio.rosso sono due cose estremamente diverse. DIV.rosso si legge come "elementi DIV con attributo class contenente rosso". DIV .rosso si legge come "elementi contenuti in un div con class contenente rosso". CSS è potente e sintetico, ma questo porta anche a poter commettere sbagli di questo tipo e a spendere parecchio tempo per capirne l'origine.

Una forma meno generica di selettore per discendenza è l'operatore >.

Scrivere a>b{} significa "applica queste regole ai b FIGLI DIRETTI di a". La regola non verrà applicata ai nipoti. Se a contiene b e b contiene un altro, solo il primo b vedrà applicate le regole specificate. Vediamolo con un esempio:

```
<html>
    <head>
        <title> Classi CSS </title>
        <style>
            DIV.rosso
            {
                color:red;
            }

            DIV > .rosso
            {
                color:green;
            }
        </style>
    </head>
    <body>
        <div class="rosso"> Testo rosso </div>
        <div> <p class="rosso"> Testo verde </p> </div>
        <div> <span> <p class="rosso">Testo nero</p> </span> </div>
    </body>
</html>
```

Il terzo div contiene uno span (un altro divisorio, marginalmente diverso dai div come vedremo in seguito, parlando di allineamento). Di conseguenza il secondo p è figlio di span, non di div, e la regola DIV>.rosso non si applica, essendo il secondo p NIPOTE, non FIGLIO, di un DIV.

16.11 Selettori per attributo

Una forma di selettor meno usata ma comoda in alcuni casi è quella per attributo. Possiamo selezionare una famiglia di elementi HTML tramite il valore di un loro attributo. Ad esempio:

```
INPUT[type=number]
{
    color:red;
}
```

In questo modo tutti gli input con attributo type=number (le caselle di testo per l'inserimento di numeri) avranno colore rosso. Come sempre, possiamo combinare queste regole:

```
#formtaxcalculator INPUT[type=number]
{
    color:red;
}
```

Tutte le caselle di testo con type=number contenute (a qualunque livello) in un elemento con id=formtaxcalculator avranno il testo di colore rosso.

16.12 Un esempio pratico di HTML e CSS - ordinare una pizza

Ordinare da asporto è una attività estremamente comune al momento della stesura di questo libro. Andremo a realizzare, per step, una interfaccia grafica per una applicazione per ordinare pizze a domicilio, da un singolo negozio locale.

Per semplicità non richiederemo la registrazione dell'utente ma ci affideremo alla fiducia. Chiunque potrà connettersi al nostro sito, inserire un nome e un indirizzo, selezionare una tipologia di pizza e un numero di pizze da ordinare, visualizzare l'anteprima del totale e procedere con l'ordine.

Cominceremo definendo la struttura generale, quindi l'HTML. Divideremo la pagina in DIV, ognuno dei quali dovrà poi essere graficato correttamente con CSS. Essendo solo una interfaccia di prova, non sarà veramente funzionante ma farà da mock-up per il progetto finale.

Immaginiamo che l'applicazione debba disporre di un titolo, in alto, largo circa quanto la pagina, e al livello successivo di due "scatole" larghe ciascuna il 50% della pagina, una contenente le condizioni del servizio e la guida per ordinare, l'altra contenente la form (come abbiamo visto in precedenza) con i controlli da usare per permettere all'utente di inserire l'ordine.

Un abbozzo di struttura potrebbe essere il seguente:

```
<html>
    <head>
        <title> Pizza - v0 </title>
    </head>
    <body>
        <div> Titolo dell'applicazione </div>
        <div> Sezione con i termini di acquisto </div>
        <div> Form di acquisto </div>
    </body>
</html>
```

Notiamo che gli elementi vengono disposti da sinistra a destra, di default, seguendo la direzione di scrittura tipica dell'inglese e di molte altre lingue europee, e vanno a capo quando esauriscono lo spazio o quando un elemento vuole occupare una propria riga (display:block).

Essendo i div elementi con display:block, la struttura è verticale. Non abbiamo i termini di acquisto e la form affiancati, come sarebbe nostra intenzione. Tralasciamo per ora il discorso sulla grafica completamente assente, e concentriamoci almeno sul layout della pagina.

Vogliamo arrivare ad avere due fasce orizzontali larghe il 100% della pagina: la prima contenente il titolo, la seconda contenente i due box, ciascuno largo il 50% del proprio contenitore. Potremmo chiamare le due fasce "header" e "content", e non si ripeteranno nella pagina, di conseguenza possiamo assegnare loro un id, e ottenere qualcosa di simile:

```
<html>
    <head>
        <title> Pizza - v1 </title>
    </head>
    <body>
        <div id="header"> Titolo dell'applicazione </div>
        <div id="content">
            <div> Sezione con i termini di acquisto </div>
            <div> Form di acquisto </div>
        </div>
    </body>
</html>
```

Non è cambiato niente rispetto a prima, ma ora che abbiamo definito le parti possiamo assegnare le regole di stile:

```
<html>
    <head>
        <title> Pizza - v1 </title>
        <style>
            #content>DIV
            {
                display:inline-block;
                width:49%;
            }
        </style>
    </head>
    <body>
        <div id="header"> Titolo dell'applicazione </div>
        <div id="content">
            <div> Sezione con i termini di acquisto </div>
            <div> Form di acquisto </div>
        </div>
    </body>
</html>
```

```
</body>
</html>
```

Ora i due div sono affiancati e il testo ha un distanziamento rispetto ai bordi (invisibili) dello 0.3%. I due div interni a content (e solo i figli diretti, non eventuali nipoti) sono larghi il 49% del proprio contenitore. Cominciamo a farci un'idea degli ingombri ora. Inseriamo del testo reale, tralasciando solo la form vera e propria:

```
<html>
<head>
    <title> Pizza - v1 </title>
    <style>

        #content>DIV
        {
            display:inline-block;
            width:49%;
        }
    </style>
</head>
<body>
    <h1 id="header"> Generation Pizza - Order Form </h1>
    <div id="content">
        <div id="terms">
            <b>Welcome to the Generation Pizza Order Form!</b>
            <p>
                Please fill the form to the right with the information relative to your address
                and desired pizza. Make sure to report food allergies in the proper box,
                as well as any other fact relevant to your order.
            </p>
            <p>
                Payment is upon delivery, which is only possible in the towns served
                by our network.
                Please allow for a 5-10 minutes delay on the time of delivery.
                Delays above 15 minutes will grant a 50% discount on the total price of the order.
            </p>
        </div>
        <div>
            Form di acquisto
        </div>
    </div>
</body>
</html>
```

Qualche piccola modifica. Abbiamo deciso di usare direttamente un elemento h1 come header, e definito un nuovo id, per ora ancora non usato ("terms", a indicare la colonna di sinistra nella riga content). Finiamo di abbellirlo, usando gli elementi a nostra disposizione:

```
<html>
<head>
    <title> Pizza - v1 </title>
    <style>

        #content>DIV
        {
            display:inline-block;
            width:49%;
            padding:0.2%;
        }

        H1
        {
            padding-left:0.2%;
            padding-top:0.5%;
            padding-bottom:0.5%;
            background-color:#3AF;
        }

        BODY
        {
            font-family:Tahoma, Arial;
        }

        #terms>P
        {
            text-align:justify;
        }

    </style>
</head>
```

```

<body>
    <h1 id="header"> Generation Pizza - Order Form </h1>
    <div id="content">
        <div id="terms">
            <b>Welcome to the Generation Pizza Order Form!</b>
            <p>
                Please fill the form to the right with the information relative to your address
                and desired pizza. Make sure to report food allergies in the proper box,
                as well as any other fact relevant to your order.
            </p>
            <p>
                Payment is upon delivery, which is only possible in the towns served
                by our network.
                Please allow for a 5-10 minutes delay on the time of delivery.
                Delays above 15 minutes will grant a 50% discount on the total price of the order.
            </p>
        </div>
        <div>
            Form di acquisto
        </div>
    </div>
</body>
</html>

```

Notiamo ancora che il carattere, settato in BODY, viene ereditato dal resto della pagina. Salvo eccezioni o altre disposizioni, una regola viene passata dal contenitore al contenuto ("cascade").

Ora passiamo alla parte "divertente" del lavoro: la form. La form è uno dei due grandi metodi di interazione fra utente e programma. La nostra form dovrà contenere le caselle di testo per indicare:

- il nome dell'ordinante
- il suo indirizzo
- la località (da scegliere tramite un elenco)
- la tipologia della pizza (da scegliere tramite un elenco)
- una casella di testo larga per indicare problematiche allergiche e altre note
- una casella di testo con l'ora di consegna desiderata, in formato hh:mm
- il numero di pizze da ordinare (supponendole tutte dello stesso tipo per semplicità)
- una casella di testo col totale dell'ordine (per ora non funzionante)

Una possibile soluzione potrebbe essere la seguente (riportiamo solo la form, per ora):

```

<form method="post" action="PlaceOrder">
    Client name
    <input type="text" name="clientname" />
    Address
    <input type="text" name="address" />
    Pizza
    <select name="pizza">
        <option value="margherita"> Margherita </option>
        <option value="salmon"> Panna e salmone </option>
    </select>
    Pieces
    <input type="number" name="pieces" />
    Town
    <select name="town">
        <option value="city1"> City 1 </option>
        <option value="city2"> City 2 </option>
    </select>
    Time of delivery
    <input type="text" name="timeofdelivery" />
    Please make sure to report allergies and preferences here:
    <textarea name="notes"></textarea>
    Total
    <input type="number" name="total" readonly
          placeholder="this box will contain the total for the order, once we study Javascript" />
    <input type="submit" value="place order" />
</form>

```

Ci sono un paio di novità. textarea genera una "maxi casella di testo" a più righe, tipicamente usata per inserire testo libero di grandi dimensioni. L'attributo placeholder inserisce un testo in una casella di testo, che scompare quando l'utente o chiunque altro vi inserisce un valore, e serve per aiutare l'utente nella compilazione. La casella di testo "totale" per ora non funzionerà - vedremo come attivarla studiando Javascript. Per evitare problemi è stata marcata come "readonly", vale a dire non modificabile dall'utente.

Allo stato attuale la form è abbastanza orribile. Possiamo risolvere con qualche semplice regola di stile, per evitare che i controlli siano tutti ammassati e allineare tutto in maniera meno brutale:

```

<html>
    <head>
        <title> Pizza - v1 </title>
        <style>

            INPUT, SELECT, TEXTAREA
            {
                display:block;
                margin-top:0.1%;
                margin-bottom:0.5%;
                width:80%;
                padding:1%;
            }

            #content>DIV
            {
                display:inline-block;
                width:49%;
                padding:0.2%;
                vertical-align:top;
            }

            H1
            {
                padding-left:0.2%;
                padding-top:0.5%;
                padding-bottom:0.5%;
                background-color:#3AF;
            }

            BODY
            {
                font-family:Tahoma, Arial;
            }

            #terms>P
            {
                text-align:justify;
            }

            FORM
            {
                padding-left:2%;
            }

        </style>
    </head>
    <body>
        <h1 id="header"> Generation Pizza - Order Form </h1>
        <div id="content">
            <div id="terms">
                <b>Welcome to the Generation Pizza Order Form!</b>
                <p>
                    Please fill the form to the right with the information relative to your address
                    and desired pizza. Make sure to report food allergies in the proper box,
                    as well as any other fact relevant to your order.
                </p>
                <p>
                    Payment is upon delivery, which is only possible in the towns served
                    by our network.
                    Please allow for a 5-10 minutes delay on the time of delivery.
                    Delays above 15 minutes will grant a 50% discount on the total price of the order.
                </p>
            </div>
            <div>
                <form method="post" action="PlaceOrder">
                    Client name
                    <input type="text" name="clientname" />
                    Address
                    <input type="text" name="address" />
                    Pizza
                    <select name="pizza">
                        <option value="margherita"> Margherita </option>
                        <option value="salmon"> Panna e salmon </option>
                    </select>
                    Pieces
                    <input type="number" name="pieces" />
                    Town
                    <select name="town">
                        <option value="city1"> City 1 </option>
                        <option value="city2"> City 2 </option>
                    </select>
                    Time of delivery
                    <input type="text" name="timeofdelivery" />
                    Please make sure to report allergies and preferences here:
                </form>
            </div>
        </div>
    </body>

```

```

<textarea name="notes"></textarea>
Total
<input type="number" name="total" readonly
placeholder="this box will contain the total for the order,
once we study Javascript"
/>
<input type="submit" value="place order" />
</form>
</div>
</body>
</html>

```

Ogni input, select o textarea prenderà la propria riga (display:block), hanno tutte la stessa lunghezza (80% del loro contenitore, vale a dire l'80% della colonna a destra), e hanno tutte un padding interno pari all'1%, oltre a un margin idoneo. Abbiamo anche dato un padding-left alla form per separare opportunamente la form dal testo della colonna a sinistra (potevamo ottenere lo stesso risultato in molti altri modi).

Il risultato non vincerà un concorso di bellezza, ma è funzionale e chiaro. E' fondamentale che gli elementi siano allineati, orizzontalmente e verticalmente, perché l'occhio del lettore non "balli". C'è anche un'altra novità: text-align:top, riferito ai DIV in content. La forma era diventata più lunga del testo, e non erano allineati verticalmente. Con questa regola entrambi sono stati "tirati" verso la cima della pagina.

Questa è la prima fase del nostro processo di sviluppo per una semplice applicazione per la prenotazione di una pizza. Mancano due elementi fondamentali:

- il collegamento a un programma Java vero e proprio, con "sotto" un database che registrerà la prenotazione. Questa parte viene detta "sviluppo server side", o "lato server"
- l'inserimento di un programma Javascript per aiutare l'utente nella compilazione dell'ordine, prevenire ordini sbagliati e in generale migliorare l'esperienza utente. Questa parte termina lo "sviluppo client side", o "lato client", e non è strettamente indispensabile, ma è richiesta in qualunque sistema o lavoro moderno

16.13 Il problema della responsività

Il web nasce per i computer, per schermi tendenzialmente grandi e per un uso tramite mouse e tastiera, ma oggi è spesso frutto tramite tablet o smartphone - schermi più piccoli, per quanto molto definiti, in cui non avrebbe senso riproporre la stessa esperienza utente di un PC.

Questo si traduce nel dover ragionare su almeno due versioni: una pc e una mobile. Un tempo questo significava scrivere due versioni diverse della stessa interfaccia, e per quanto questo sia ancora consigliabile in alcuni casi, si tende a preferire la scrittura di una sola interfaccia (di modo da non doverne aggiornare due a ogni cambiamento, violando il principio di sviluppo noto come DRY), ma che si adatti in maniera intelligente al dispositivo su cui viene renderizzata.

Questa scelta di sviluppo prende il nome di layout responsivo, o anche mobile first, vale a dire che si adatti allo schermo su cui "girerà" o che sia ragionato "prima per il mobile".

L'esempio sviluppato in precedenza, Generation Pizza, non è responsivo. Il layout (la disposizione degli elementi della pagina) non cambia in base alla risoluzione a disposizione o al polliciaggio.

E' facile vederlo tramite il browser. Aprendo Chrome, possiamo attivare "Altri strumenti" dal menù in alto a destra, poi "Strumenti per sviluppatori" e poi cliccare sull'icona combinata di telefonino e tablet per attivare l'opzione "Google device toolbar", con cui Chrome simulerà un telefonino generico, un tablet o un device specifico, di modo da dare allo sviluppatore un'idea di come il suo lavoro verrà visualizzato su device di cui non dispone.

Aprendo l'esempio sopra, noteremo che la disposizione è rimasta uguale per un device 400x642, così come per diversi telefonini e tablet. Sarebbe opportuno cambiare il nostro layout, ragionando su un utente che voglia ordinare una pizza dal suo telefono. A questo punto ha senso lavorare su una struttura completamente verticale, con un titolo, i terms subito sotto, e alla fine la form col pulsante di invio.

Questo significa impostare la proprietà `display` dei due `div` interni (`i terms` e `la form`) a `block`, ma solo sotto una data risoluzione. Questa operazione, molto comune, si può ottenere tramite una stilizzazione condizionale con l'uso della direttiva CSS `@media`:

```
@media screen and (max-width:600px)
{
    #content>DIV
    {
        display:block;
        width:100%;
    }
}
```

Si legge come "quando lo schermo ha una larghezza in pixel inferiore o uguale a 600px, applica queste regole". Perchè questo funzioni però dovremo modificare anche l'html, aggiungendo una clausola:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Senza andare in dettaglio, "avvisa" il browser che stiamo creando un layout responsivo. Lo vedremo successivamente anche con Bootstrap, che semplificherà di molto questo lavoro.

Resta da chiarire perché "media screen" e non semplicemente media. E' possibile specificare informazioni diverse per media diversi. Ad esempio potremmo stilizzare diversamente la pagina renderizzata a schermo (media screen, appunto) e la pagina stampata (media print), con lo stesso meccanismo condizionale.

I valori possibili sono all, screen, print e speech, da usare per quei programmi che "leggono" la pagina per chi non volesse o non potesse leggerla dallo schermo.

L'esempio completo è il seguente:

```
<html>
  <head>
    <title> Pizza - v1 </title>
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <style>
      @media screen and (max-width:800px)
      {
        #content>DIV
        {
          display:block !important;
        }
      }
    </style>
  </head>
  <body>
    <div id="content">
      <div>Content</div>
    </div>
  </body>
</html>
```

```

        width:100% !important;
    }
}

#content>DIV
{
    display:inline-block;
    width:49%;
    padding:0.2%;
    vertical-align:top;
}

INPUT, SELECT, TEXTAREA
{
    display:block;
    margin-top:0.1%;
    margin-bottom:0.5%;
    width:80%;
    padding:1%;
}

FORM
{
    padding-left:2%;
}

H1
{
    padding-left:0.2%;
    padding-top:0.5%;
    padding-bottom:0.5%;
    background-color:#3AF;
}

BODY
{
    font-family:Tahoma, Arial;
}

#terms>P
{
    text-align:justify;
}

</style>
</head>
<body>
<h1 id="header"> Generation Pizza - Order Form </h1>
<div id="content">
    <div id="terms">
        <b>Welcome to the Generation Pizza Order Form!</b>
        <p>
            Please fill the form to the right with the information relative to your address  

            and desired pizza. Make sure to report food allergies in the proper box,  

            as well as any other fact relevant to your order.
        </p>
        <p>
            Payment is upon delivery, which is only possible in the towns served  

            by our network.  

            Please allow for a 5-10 minutes delay on the time of delivery.  

            Delays above 15 minutes will grant a 50% discount on the total price of the order
        </p>
    </div>
    <div>
        <form method="post" action="PlaceOrder">
            Client name
            <input type="text" name="clientname" />
            Address
            <input type="text" name="address" />
            Pizza
            <select name="pizza">
                <option value="margherita"> Margherita </option>
                <option value="salmon"> Panna e salmone </option>
            </select>
            Pieces
            <input type="number" name="pieces" />
            Town
            <select name="town">
                <option value="city1"> City 1 </option>
                <option value="city2"> City 2 </option>
            </select>
            Time of delivery
            <input type="text" name="timeofdelivery" />
            Please make sure to report allergies and preferences here:
            <textarea name="notes"></textarea>
        </form>
    </div>
</div>

```

```
Total
<input type="number" name="total" readonly
placeholder="this box will contain the total for the order,
once we study Javascript"
/>
<input type="submit" value="place order" />
</form>
</div>
</body>
</html>
```

16.14 Approfondimenti

Il riferimento consigliato è la guida di w3schools: <https://www.w3schools.com/css/default.asp>. E' utilissima per approfondire le dinamiche di display (e di conseguenza il layout della pagina), così come il discorso sulle grandezze (di cui riporto il link diretto: https://www.w3schools.com/cssref/css_units.asp).

Per quanto convenga scorrere almeno il primo link, le principali problematiche di layout e almeno una struttura basilare verrà fornita dal framework CSS / Javascript noto come Bootstrap, che introdurremo a breve.

17 - Bootstrap

17.1 Introduzione

I due moduli precedenti (HTML e CSS) ci hanno dato un'idea del funzionamento delle due principali tecnologie per la presentazione dei contenuti su web, ma nel corso del tempo è emersa l'esigenza per soluzioni più semplici da usare, maggiormente scalabili, che risolvessero una serie di problemi ricorrenti, fra i quali quello della responsività.

Allo stesso modo, sono state sviluppate soluzioni pensate per dare alle applicazioni un look-and-feel organico e professionale senza dover ogni volta reinventare la ruota, o scrivere regole di stile per ogni singolo componente come abbiamo fatto noi in precedenza.

Questo si è tradotto nello sviluppo di librerie e framework CSS o CSS/Javascript per la generazione di interfacce. Ne approfittiamo per chiarire la differenza fra libreria e framework: una libreria è uno strumento che possiamo utilizzare all'interno del nostro progetto, un framework è una cornice entro cui scriviamo un progetto, una sorta di punto di partenza standard con regole più o meno vincolanti, che siamo obbligati a rispettare se vogliamo usufruire dei suoi servizi.

Due strumenti molto interessanti sono W3.CSS, la libreria standard di W3Schools, e Bootstrap. W3.CSS è un framework solo CSS, molto semplice, che offre un sistema di layout a griglia simile a quello che vedremo fra poco con Bootstrap, così come colori e stili standard per la maggior parte dei componenti dell'interfaccia.

Bootstrap, maggiormente diffuso e potente ma meno semplice, integra CSS e Javascript (che per ora useremo in maniera indiretta e che approfondiremo in seguito) per offrire una interfaccia grafica ricca e raffinata.

Si tratta comunque di due possibilità fra molte, e allo studente conviene concentrarsi sui concetti, sul cosa vogliamo ottenere, per poi passare al come ottenerlo tramite lo strumento selezionato. Eventuali altri framework (CSS ma non solo) possono presentare dettagli implementativi diversi ma tipicamente riportano gli stessi concetti di base.

Bootstrap viene costantemente aggiornato e migliorato. Mentre scrivo, la versione 5 è sperimentale mentre la più diffusa è la 4, che è quella che tratteremo nelle sue funzioni basilari, capendo cosa offre e come funziona.

17.2 Installazione

La maniera più rapida di "attivare" Bootstrap all'interno dei nostri progetti è tramite un'inclusione da web. L'inclusione da web è un sistema semplice che permette di collegare alla propria pagina files CSS, JS o di altro tipo scaricandoli da un server online invece di mantenerli sulla propria macchina.

Questo da un lato ci costringe a essere connessi mentre lavoriamo, mentre dall'altro ci garantisce di lavorare sempre su file aggiornati e spesso anche di non doverli ricaricare, considerando che una volta scaricato una volta per un sito il file resta nella cache del browser.

Uno scheletro di pagina Bootstrap, che potremo copiare e incollare all'inizio di ogni nuovo progetto, è la seguente:

```
<!DOCTYPE html>
<html>
    <head>
        <title> Bootstrap template </title>
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <!-- css di bootstrap-->
        <link rel="stylesheet"
              href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
        >
        <!-- jQuery, una libreria Javascript che approfondiremo a breve -->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
        <!-- Popper. Un'altra libreria Javascript di cui Bootstrap fa uso. Non la vedremo.-->
        <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
        <!-- Javascript interno a Bootstrap-->
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
    </head>
    <body>
        <div class="container-fluid">
            Prototipo di pagina per ordinare la pizza in Bootstrap
        </div>
    </body>
</html>
```

Abbiamo qualche novità. In cima dobbiamo specificare il doctype. Le vecchie versioni di HTML non lo richiedevano (e noi ce la siamo cavata egregiamente senza fino ad ora), ma Bootstrap è pensato per lavorare su HTML 5 (l'ultima versione), che richiede il doctype. Doctype NON è un elemento, ma serve per indicare al browser la tipologia di documento che sta per gestire.

I tag link (già visto in precedenza) e script (nuovo) servono per effettuare quelle inclusioni via web di cui abbiamo parlato prima. Il tag link scarica il solo file .css, contenente le classi (lo vedremo a breve) che formano la base del funzionamento di Bootstrap. Gli altri files sono librerie Javascript necessarie per il funzionamento dei componenti avanzati (non presenti in HTML standard e non ottenibili in CSS puro).

Notiamo che solo il primo tag (link) è necessario, se non vogliamo usare le componenti avanzate, ma non ci farà male importarle tutte per non avere problemi in seguito. Venendo da Java, potete immaginare quelle righe come import di package.

Gli import, per Bootstrap ma non solo, vengono tipicamente fatti in HEAD, e le classi e gli script richiamati in BODY.

Ad ora, body contiene un unico DIV, che usa una classe container-fluid che non abbiamo definito noi. In effetti, è presente nel file .css che abbiamo importato tramite link. Bootstrap ha bisogno di avere un elemento contenitore (container, appunto) per racchiudere i contenuti del sito, potendo scegliere fra container-fluid (contenitore full-screen, largo quanto tutta l'area visualizzabile) e container (un contenitore a grandezza fissa, responsivo, non necessariamente largo quanto la pagina).

Entrambi i container hanno padding sinistri e destri di 15 pixel, mentre non vengono specificati padding top e bottom, che possono essere aggiunti successivamente in CSS puro o con "classi di utilità", come vedremo fra poco con un esempio.

Per la classe .container, la dimensione cambia in base alla dimensione dello schermo, e ne abbiamo anche diverse versioni (container-sm, container-md, container-lg, container-xl) che cambiano anch'esse di dimensione in base allo schermo. I dettagli relativamente alle dimensioni e alle risoluzioni a cui queste vengono attivate sono presenti all'indirizzo: https://www.w3schools.com/bootstrap4/bootstrap_containers.asp (W3Schools).

Noi dobbiamo però ricordarci la progressione da extra small (nessun suffisso) a sm, a md, a lg, a xl (small, medium, large, extra-large), corrispondente a larghezze massime di < 576px, 576 - 768px, 768 - 992px, 992 - 1200px, > 1200px rispettivamente. Queste saranno le dimensioni di cui doveremo preoccuparci, non necessariamente una per una, nel progettare un sito Bootstrap.

Per i nostri scopi, useremo alternativamente container e container-fluid, per garantire un minimo di varietà. All'interno dei container svilupperemo un sito per "colonne". Ogni fascia orizzontale sarà divisa in un certo numero di colonne (responsive), seguendo quello che viene detto "grid system" (sistema a griglia).

17.3 Grid System ed elementi di uso comune

Un layout Bootstrap è strutturato secondo una "griglia" di massimo 12 colonne. Avremo un div con class row (una "fascia", una "riga") divisa in colonne. Le colonne all'interno di una row occupano una percentuale della larghezza, in maniera simile a quanto abbiamo visto in CSS standard, e in effetti Bootstrap sfrutta le proprietà che abbiamo visto prima, semplificandone l'uso.

Cominciamo con un primo esempio, il più basilare:

```
<!DOCTYPE html>
<html>
    <head>
        <title> Pizza Bootstrap </title>
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <!-- css di bootstrap-->
        <link rel="stylesheet"
              href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
        >
        <!-- jQuery, una libreria Javascript che approfondiremo a breve -->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
        <!-- Popper. Un'altra libreria Javascript di cui Bootstrap fa uso. Non la vedremo.-->
        <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
        <!-- Javascript interno a Bootstrap-->
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
    <style>
        .generation-button
        {
            color:white;
            background-color:#69F;
            margin-top:2%;
        }

        H1
        {
            padding-left:0.2%;
            padding-top:0.5%;
            padding-bottom:0.5%;
            background-color:#3AF;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1 id="header"> Generation Pizza - Order Form </h1>
        <div class="row">
            <div class="col" id="terms">
                <b>Welcome to the Generation Pizza Order Form!</b>
                <p>
                    Please fill the form to the right with the information relative to your
                    address and desired pizza.
                    Make sure to report food allergies in the proper box,
                    as well as any other fact relevant to your order.
                </p>
                <p>
                    Payment is upon delivery, which is only possible in the towns served
                    by our network.
                    Please allow for a 5-10 minutes delay on the time of delivery.
                    Delays above 15 minutes will grant a 50% discount
                    on the total price of the order.
                </p>
            </div>
            <div class="col">
                <form method="post" action="PlaceOrder">
                    Client name
                    <input type="text" name="clientname" class="form-control"/>
                    Address
                    <input type="text" name="address" class="form-control" />
                    Pizza
                    <select name="pizza" class="form-control">
                        <option value="margherita"> Margherita </option>
                        <option value="salmon"> Panna e salmone </option>
                    </select>
                    Pieces
                    <input type="number" name="pieces" class="form-control"/>
                    Town
                    <select name="town" class="form-control">
                        <option value="city1"> City 1 </option>
                        <option value="city2"> City 2 </option>
                    </select>
                    Time of delivery
                    <input type="text" name="timeofdelivery" class="form-control" />
                    Please make sure to report allergies and preferences here:
                    <textarea name="notes" class="form-control"></textarea>
                </form>
            </div>
        </div>
    </div>
</body>
```

```

        Total
        <input type="number" class="form-control" name="total" readonly
               placeholder="this box will contain the total for the order,
               once we study Javascript"
        />
        <input type="submit" class="form-control generation-button" value="place order" />
    </form>
</div>
</div>
</body>
</html>

```

Abbiamo riportato la struttura precedente. La row contiene due colonne, con classe .col. Bootstrap le dimensiona automaticamente al 50% e le posiziona in linea invece che con display:block. Abbiamo anche riportato la form, ai cui elementi abbiamo applicato una classe predefinita di Bootstrap (form-control), che imposta la larghezza del controllo al 100% del proprio contenitore, e applica alcune impostazioni di stile per trasformare il controllo standard in qualcosa dal look più professionale.

Notiamo che abbiamo usato il nostro CSS assieme a Bootstrap. Il bottone di invio appartiene a due classi, una di Bootstrap (form-control) e una nostra (generation-button). E' il caso comune: usiamo Bootstrap come base su cui costruire.

In questo caso Bootstrap ha deciso per noi la larghezza delle colonne. Se ne avessimo usate tre, o quattro, o sei, le avrebbe ridimensionate per occupare l'intera row, creando colonne tutte uguali. Diamone una dimostrazione creando un sottomenu' con tre collegamenti sotto il menù principale:

```

<!DOCTYPE html>
<html>
    <head>
        (include e configurazioni varie)
    </head>
    <body>
        <div class="container">
            <h1 id="header"> Generation Pizza - Order Form </h1>
            <div class="row" id="menu">
                <div class="col" style="text-align:center">
                    <a href="https://it.wikipedia.org/wiki/Pizza">
                        A history of Pizza
                    </a>
                </div>
                <div class="col" style="text-align:center">
                    <a href="http://www.generationpizza.com">
                        Our pizza shop
                    </a>
                </div>
                <div class="col" style="text-align:right;padding-right:2%">
                    @Generation 2021
                </div>
            </div>
            <div class="row">
                <div class="col" id="terms">
                    (terms)
                </div>
                <div class="col">
                    (form)
                </div>
            </div>
        </div>
    </body>
</html>

```

Abbiamo tre fasce. La prima è un H1, un elemento block che prende tutto il contenitore (tutto il .container di Bootstrap). La seconda fascia è una row Bootstrap vera e propria, divisa in tre colonne di uguali dimensioni. Dispone anche di un id, menu, che utilizziamo per definire un padding specifico per quella riga, diverso da quello delle normali .col.

Il menù contiene due link: un collegamento alla pagina Wikipedia della Pizza e uno a un sito non esistente, la Pizzeria di Generation. A destra abbiamo una notifica di copyright. Le tre colonne sono uguali e ciascuna occupa il 33% della propria fascia. Sottoabbiamo la .row vista in precedenza, con due col automaticamente dimensionate al 50% della riga.

Ora vogliamo aggiungere la responsività alla pagina. Le .col ad ora cercheranno di conservare questa struttura, ma noi vogliamo ottenere un comportamento diverso. Sotto una determinata risoluzione, gradiremmo che il menù rimanesse uguale (dovrebbe starci), mentre form e terms andrebbero messi verticalmente, uno sotto l'altro, come abbiamo fatto in precedenza in CSS puro. Possiamo ottenerlo in questo modo:

```

<div class="container">
    <h1 id="header"> Generation Pizza - Order Form </h1>

```

```

<div class="row" id="menu">
    (menu)
</div>
<div class="row">
    <div class="col-sm" id="terms">
        (terms)
    </div>
    <div class="col-sm">
        (form)
    </div>
</div>
</div>

```

Abbiamo specificato .col-sm invece di .col. Il suffisso sm sta per small, e riconduce alle distinzioni di dimensione viste in precedenza. Abbiamo .col-sm, .col-md, .col-lg e .col-xl, e sono tutte direttive responsive vincolate a una risoluzione. Scrivendo .col-sm stiamo dicendo "colonna responsiva di larghezza automatica a partire da small". Le direttive coprono il caso superiore, per cui questa colonna viene indicata come responsiva per tutte le dimensioni (sm, md, lg e xl).

Le colonne della terza riga (terms e form) hanno dimensioni automatiche, ed essendo responsive scorrono in verticale quando la dimensione dello schermo non permette di mantenerle in linea.

Per adesso sappiamo usare due classi: .col, larghezza automatica a partire dagli schermi più piccoli, e .col-x, dove x è sm,md,lg o xl indica larghezza automatica a partire da una data dimensione, andando verso l'alto ove non specificato diversamente.

In generale, però, non vogliamo colonne tutte uguali, o almeno non sempre, anche quando sono responsive. In effetti, vogliamo che elementi diversi della pagina abbiano dimensioni diverse a seconda delle dimensioni dello schermo.

Bootstrap offre questa possibilità ragionando su righe di dodici colonne. Modifichiamo l'esempio di sopra definendo che i terms occupino "solo" quattro colonne su 12, in maniera responsiva, dalla taglia sm in su, mentre la form ne occuperà 8. Lo otteniamo come segue:

```

<div class="row">
    <div class="col-sm-4" id="terms">
        (terms)
    </div>
    <div class="col-sm-8">
        (form)
    </div>
</div>

```

A risoluzioni "accettabili" terms occuperà 4 colonne su 12 (quindi un terzo dello schermo), mentre la form ne occuperà 8 (quindi due terzi), ma sono possibili molte combinazioni. Collassando lo schermo (stringendo banalmente il browser, o aprendo la pagina da un cellulare) i due div saranno uno sotto l'altro, col div dei terms in cima, quindi entrambi occuperanno 12 colonne su 12, uno sotto l'altro, e avremo nella pratica due righe.

Siamo in grado di fare di meglio. Possiamo specificare un "consumo" di colonne diverse in base alla risoluzione. Supponiamo di volere, per schermi grandi, che i termini occupino di nuovo il 50% della riga (quindi sei colonne).

Avremo quindi due layout diversi: 1/3 - 2/3 per schermi piccoli e medi, e 1/2-1/2 per schermi grandi.
Una soluzione è la seguente:

```

<div class="row">
    <div class="col-sm-4 col-lg-6" id="terms">
        (terms)
    </div>
    <div class="col-sm-8 col-lg-6">
        (form)
    </div>
</div>

```

A schermo grande (basta un full-HD) i due DIV saranno uguali. Stringendo lo schermo, i terms saranno ridotti a un terzo, e la form sarà più grande. Stringendo ulteriormente, tutto verrà messo in colonna, rinunciando ad affiancare gli elementi.

La classe col-sm-4 si legge come "per small e medium questo div prenderà quattro colonne su dodici", mentre col-lg-6 si legge "per large ed extra large questo div prenderà sei colonne su dodici". Ricordiamoci che le regole "scalano verso l'alto". col-sm-4 viene "bloccata" dalla presenza di col-lg-6, che ne prende il posto per le alte risoluzioni.

La somma delle colonne non deve essere necessariamente dodici. E' possibile lasciarne vuote, per quanto non sia forse la prassi migliore.

Ricapitolando:

- dividiamo la schermata in "righe" responsive (div.row)
- ogni riga può contenere più colonne (fino a 12) raggruppate tipicamente in div a loro volta
- i div possono avere dimensione automatica (classe .col) o dimensioni esplicitate, responsive e rapportate a una definita risoluzione, col meccanismo visto prima. Possiamo specificare classi diverse per gestire risoluzioni diverse.

La forma generale è la seguente:

```
<div class="row">
    <div class="col-?-? col-?-? ..." > col1 </div>
    <div class="col-?-? col-?-? ..." > col2 </div>
    ...
</div>
```

In generale, per non complicarci la vita, possiamo ragionare su un paio di casi: schermi da pc (che sono ormai tutti full hd o abbastanza vicini) e cellulari / tablet, quindi un layout completo e uno completamente verticale. A tale scopo potrebbe essere sufficiente usare col-sm-x, che "scalerà" verso tutte le altre risoluzioni, e andrà in verticale sotto la dimensione "small" (meno di 576px di larghezza dello schermo). Nel lavoro pratico a volte si definisce anche un terzo caso, la misura intermedia, per vecchi laptop o tablet non particolarmente grandi.

17.4 Colori e loro significato in Bootstrap

All'interno di un sito web, come abbiamo anticipato, colore e forma devono suggerire la funzione. Bootstrap esemplifica questo concetto con la definizione di una serie di classi di stile con dei colori predefiniti associati a dei messaggi standard.

Vediamo un esempio rapido:

```
<input type="submit" class="form-control bg-primary text-white" value="place order" />
```

Prima avevamo definito noi un colore per il testo e un colore di sfondo. Ora ci appoggiamo a due colori standard definiti da bootstrap: `bg-primary` (colore primario di sfondo, di default un azzurro chiaro) e `text-white` (testo bianco).

Le classi `bg-*` definiscono il colore degli sfondi, le classi `text-*` dei caratteri. Possono essere applicate a qualunque elemento, ma quando applichiamo le `text-*` a un link queste acquisiscono un `hover` (una mutazione di colore contestuale al mouse che passa sul link) di un colore leggermente più scuro.

Possiamo usare le classi predefinite legate ai messaggi, ridefinirle o ignorarle definendo i nostri colori e la nostra logica dei colori, ma essendo quelle di default usate piuttosto spesso conviene riconoscerle.

Le possibilità definite di default sono `.text-muted` (testo mutato), `.text-primary` (colore primario del sito, non del testo), `.text-success` (testo o colore di un elemento che indica il successo di una operazione), `.text-info` (testo o box di informativa), `.text-warning` (testo di avvertimento), `.text-danger` (avvertimento di pericolo), `.text-secondary` (testo secondario, non rilevante quanto il corpo del testo principale), `.text-white` (bianco), `.text-dark` (testo scuro), `.text-body` (corpo standard del testo) e `.text-light` (testo grigio chiaro), mentre per gli sfondi abbiamo `.bg-primary`, `.bg-success`, `.bg-info`, `.bg-warning`, `.bg-danger`, `.bg-secondary`, `.bg-dark`, `.bg-light`, con significati analoghi.

17.5 Cards

Le cards sono "figurine", box bordati con padding attorno al contenuto, predisposte per avere un header e un footer. La struttura basilare prevede un div contenitore e tre div di contenuto, di cui solo il body è obbligatorio, con ruolo di header, body e footer:

```
<div class="card">
    <div class="card-header">Header (opzionale) </div>
    <div class="card-body">Contenuto</div>
    <div class="card-footer">Footer (opzionale)</div>
</div>
```

Possono essere pensate come "schede" per presentare un elemento, una didascalia all'interno di una pagina. Vediamo il seguente esempio che presenta una serie di prodotti con un pulsante di acquisto:

```
<!DOCTYPE html>
<html>
    <head>
        <!-- css di bootstrap-->
        <title> Pizza Bootstrap </title>
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <link rel="stylesheet"
              href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
        >
        <!-- jQuery, una libreria Javascript che approfondiremo a breve -->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
        <!-- Popper. Un'altra libreria Javascript di cui Bootstrap fa uso. Non la vedremo.-->
        <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
        <!-- Javascript interno a Bootstrap-->
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
        <style>
            H1
            {
                padding-left:0.2%;
                padding-top:0.5%;
                padding-bottom:0.5%;
                background-color:#3AF;
            }

            #menu DIV
            {
                padding:1%;
            }

        </style>
    </head>
    <body>
        <div class="container">
            <h1 id="header"> Shop </h1>
            <div class="row" id="products">
                <div class="col-sm-4 card">
                    <div class="card-header">
                        Generation Java - TextBook
                    </div>
                    <div class="card-body">
                        Lorem ipsum sit dolorem <br />
                        19.99 &euro;
                    </div>
                    <div class="card-footer">
                        <input
                            type="button"
                            class="btn btn-primary"
                            value="buy"
                        />
                    </div>
                </div>
                <div class="col-sm-4 card">
                    <div class="card-header">
                        Generation HTML - TextBook
                    </div>
                    <div class="card-body">
                        Lorem ipsum sit dolorem <br />
                        19.99 &euro;
                    </div>
                    <div class="card-footer">
                        <input
                            type="button"
                            class="btn btn-primary"
                            value="buy"
                        />
                    </div>
                </div>
            </div>
        </div>
    </body>
</html>
```

```

        </div>          </div>
<div class="col-sm-4 card">
    <div class="card-header">
        Generation SQL - TextBook
    </div>
    <div class="card-body">
        Lorem ipsum sit dolorem <br />
        19.99 &euro;
    </div>
    <div class="card-footer">
        <input
            type="button"
            class="btn btn-primary"
            value="buy"
        />
    </div>
</div>
<div class="col-sm-4 card">
    <div class="card-header">
        Generation Javascript - TextBook
    </div>
    <div class="card-body">
        Lorem ipsum sit dolorem <br />
        19.99 &euro;
    </div>
    <div class="card-footer">
        <input
            type="button"
            class="btn btn-primary"
            value="buy"
        />
    </div>
</div>
</div>
</body>
</html>

```

Notiamo che abbiamo violato una regola delle griglie. Abbiamo specificato un totale di colonne che supera 12 (sono in totale 16 colonne), e in effetti l'ultima card viene spostata alla riga sotto. E' il comportamento normalmente accettato quando si stampano elenchi di oggetti di cui non conosciamo a priori la lunghezza: prendono diverse righe anche se fanno, formalmente, tutti parte della stessa row. Succederà sempre quando stamperemo il risultato di una query di un db nel browser.

I button di acquisto non sono funzionanti, ma ci danno l'opportunità di mostrare la classe `btn` di Bootstrap associata a uno dei colori predefiniti (`primary`).

17.6 Barre di navigazione

Una barra di navigazione (navbar) è il classico menù che troviamo in cima alla pagina nella maggior parte dei siti e delle web app. Questo può essere ottenuto con html e css standard, ma Bootstrap offre una classe apposita che cambia l'aspetto della navbar a seconda della risoluzione, facendola "collassare" a risoluzioni sufficientemente piccole.

Vediamo un esempio per il negozio di libri di prima:

```
<nav class="navbar navbar-expand-sm bg-dark">
    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link" href="http://www.generationbookshop.com">
                Home page
            </a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="http://www.generationbookshop.com/catalogue">
                Catalogue
            </a>
        </li>
        <li class="nav-item">
            <form class="form-inline">
                <input type="text" class="form-control" placeholder="search for" />
            </form>
        </li>
    </ul>
</nav>
```

L'elemento è nav, e contiene un ul (unordered list, un tag di HTML che genera una lista). All'interno di ul abbiamo elementi li ("list item"), che hanno senso di esistere solo all'interno di ul. Sia la ul che le li hanno le classi corrispondenti per dire a Bootstrap di graficarle come navbar. Due dei li contengono dei link (elementi a) che vengono graficati di conseguenza. Il terzo elemento li contiene una casella di testo in una form con una particolare classe (form-inline) che allinea gli elementi della form a sinistra e cerca di mantenerli in linea, a differenza di quanto accadrebbe normalmente con form-control.

Il menù viene allineato in verticale per le risoluzioni troppo piccole, come è possibile verificare stringendo la finestra del browser.

17.7 Approfondimenti

Bootstrap dispone di molti altri componenti evoluti che migliorano l'aspetto e l'interazione con la pagina, ma che esulano dai nostri scopi. Consigliamo come riferimento di nuovo la guida di W3Schools per Bootstrap 4 (<https://www.w3schools.com/bootstrap4/default.asp>), così come il sito ufficiale di Bootstrap, per approfondire tutte le possibilità offerte da questo framework (<https://getbootstrap.com/docs/4.6/getting-started/introduction/>).

18 - Javascript

18.1 Introduzione

Javascript è un linguaggio di programmazione pensato per migliorare l'esperienza utente nella fruizione delle pagine web, e successivamente nelle web app. Può essere ospitato all'interno di pagine web alla stessa maniera di CSS, tramite un tag specifico (script) e può governarne il comportamento, aggiungendo interazioni non previste da HTML standard.

A dispetto del nome, ha solo una somiglianza superficiale con Java. Le "ossa" del linguaggio sono molto diverse.

- Java è a tipizzazione statica: una variabile, una volta dichiarata, è per sempre vista come dello stesso tipo. In Javascript lo stesso simbolo può cambiare tipo nel tempo.
- L'ereditarietà di Java è "classica", vale a dire definita in termini di classi e interfacce. Tutti gli oggetti di una data classe offrono gli stessi servizi, come abbiamo visto. In Javascript le classi non esistono "davvero", e due oggetti della stessa classe possono avere metodi diversi. L'ereditarietà di Javascript è prototipale, e non la approfondiremo in questo corso.
- In Javascript il codice stesso è un tipo di variabile. Possiamo scrivere `var v = function(x){return x*2;}`, e poi usare `v` come fosse un metodo (`v(4)` restituirà 8) o come una variabile, e passarla a un'altra funzione che potrà poi richiamarla. Lo chiariremo meglio in seguito, parlando di callback.
- Javascript non è rigido sui concetti di vero e falso. Javascript ha i concetti di "truthy" e "falsy", "circa vero" e "circa falso". In Java scrivere `if(0)` restituisce un errore. In Javascript, il valore 0 è considerato falso, così come una stringa vuota. Questo rende la valutazione delle condizioni più complessa ed è più facile commettere errori.
- Gli IDE faticano a fornire gli stessi aiuti forniti per Java nella scrittura del codice Javascript. Come vedremo, non sapremo mai quali proprietà ha un oggetto, o di quale tipo esse siano, per cui il consueto "missing field" con cui Eclipse ci avverte in Java non verrà ad aiutarci.

D'altro canto, Javascript è estremamente flessibile e permette una rapidità espressiva che è difficile ritrovare in Java. Soprattutto, Javascript è integrato con gli oggetti della pagina Web (li chiamiamo oggetti non per caso, come vedremo a breve) e può manipolarli agilmente.

18.2 Variabili in Javascript

Molto di ciò che troviamo in Java è presente anche in Javascript, per quanto con delle differenze. Partiamo con le seguenti dichiarazioni con assegnamento:

```
a = 4;  
var b = 10;  
let c = 20;
```

La prima sintassi (`a=4`) è equivalente alla seconda (`var b = 10`). Sono due dichiarazioni con assegnamento globale: il simbolo `a` sarà disponibile globalmente, in tutto il codice Javascript del nostro progetto. Venendo da Java questo dovrebbe provocare un brivido spiacevole, ma così stanno le cose: `var`, o l'assenza in generale della particella di dichiarazione, indica che il simbolo creato sarà disponibile ovunque, con tutti i naming conflict relativi.

Questo cambia nel caso in cui `var` sia stato dichiarato dentro una function. In questo caso il simbolo `b` apparterrà alla function, non al mondo. Chiariamolo con un esempio:

```
a = 4;  
function v(){return a;}  
v(); // restituisce 4  
function k(){var a=1; return a;}  
k() // restituisce 1  
a; // stampa 4  
function horrorvacui(){a = 1; return;}  
horrorvacui();  
a; // stampa 1. La funzione ha cambiato una variabile globale!
```

Direi che possiamo essere tutti d'accordo nel deprecare `var` salvo casi rari e nell'usare `let`. Le variabili dichiarate con `let` sono block-scoped, vale a dire che si comportano come in Java. Appartengono allo scope in cui sono state dichiarate e per fortuna non sono visibili fuori.

In nessun caso, tuttavia, possiamo dichiarare il tipo della variabile. In Javascript il tipo della variabile si desume dal suo contenuto. Se in `a` trovo un numero, la variabile è numerica. Se ci trovo un oggetto è un oggetto. Se ci trovo del codice, è una function (un metodo). E Javascript non si fa neanche troppi scrupoli a mischiare "mele e pere". Vediamo un esempio:

```
a = "Pippo";  
b = 5;  
c = a+b;  
console.log(c) // stampa c. Stamperà Pippo5, e c è chiaramente una stringa  
a = "5";  
c = a * 4;  
console.log(c); // stamperà 20. c è un numero, mentre a era una stringa. Javascript la ha convertita esplicitamente in un numero per la  
moltiplicazione  
a = "5";  
b = "1";  
c = a+b;  
console.log(c); // "51", una stringa
```

Javascript è disinvolto nel gestire le stringhe, e i tipi in generale. Se qualcosa può "sembrargli" un numero, e lo usiamo in operazioni algebriche, proverà a convertirlo in un numero. Al contrario, alcuni dei dati che gli arrivano dagli utenti e che dovrebbe vedere come numeri saranno visti come stringhe, e lui li tratterà di conseguenza, come abbiamo visto nell'ultimo esempio.

Ciò nonostante è sbagliato dire che Javascript non ha i tipi.

18.3 Tipi primitivi in Javascript

Javascript dispone di alcuni tipi predefiniti, piuttosto diversi dalla tradizione Java. E' possibile riconoscere il tipo di una variabile utilizzando l'operatore `typeof`, che restituisce una stringa. Vediamo una serie di esempi:

```
let a = 5;
console.log(typeof a); // "number", con o senza virgola
console.log(typeof typeof a); // typeof "number" -> string
a = true;
console.log(typeof a); //boolean
```

Questi sono tre tipi primitivi. In Javascript i tipi primitivi sono "tipi senza metodi", o "tipi non oggetti". A questo dobbiamo aggiungere il tipo "undefined", che è il valore di default per le variabili ancora non popolate, e il tipo null.

undefined è il valore predefinito delle variabili vuote, mentre null serve a indicare un vuoto "deliberato". Javascript tuttavia ha un bug che riporta come "object" il tipo di null se usiamo `typeof`. Per approfondire rimandiamo a questo ottimo articolo: <https://2ality.com/2013/10/typeof-null.html>. Resta il fatto che null e undefined siano due tipi primitivi

Serve una nota sulle stringhe. In Javascript sono primitivi, non oggetti, ma si possono usare come oggetti, facendo Javascript un lavoro di "boxing". Il loro stato è ambiguo come in Java. In effetti possiamo usare metodi sulle stringhe:

```
"Javascript".indexOf("a"); // stampa 1, la posizione della prima a in questa stringa
```

Sempre parlando di stringhe, queste possono essere specificate tramite virgolette doppie (come in Java) o singole, quindi come ' o ". Si può inoltre usare il backtick (`) per ottenere stringhe particolari, in cui possiamo stampare delle variabili senza bisogno di concatenarle. Un esempio chiarirà questo metodo:

```
let name = 'John';
let surname = 'Smith';
let welcome = `Hello mr ${name} ${surname}`;
console.log(welcome);
```

La sintassi \${v} dentro una stringa con i backtick si legge "stampa la variabile v dentro questa stringa". E' particolarmente utile perchè possiamo andare a capo nel codice mentre scriviamo. Ad esempio:

```
let welcome2 = `Hello Dear Sir ${name} ${surname}
please welcome to our humble abode, enter and leave behind some of
the happiness you bring`;
console.log(welcome2);
```

Questa flessibilità ci permetterà di incorporare con maggiore facilità Javascript in HTML, nel suo ambiente naturale.

18.4 Il tipo function

Il tipo function è un tipo a sé che identifica una funzione. Una funzione è un blocco di codice che può o meno restituire un valore. Potete pensarla come a un metodo che non abbia necessariamente un oggetto o una classe a cui appartenere.

Segnaliamo le principali differenze con Java:

- un metodo Java appartiene a una classe o a un oggetto. Una funzione può appartenere "all'intero sistema"
- un metodo Java ha un ritorno dichiarato. Una funzione può restituire qualcosa o non restituirlo, restituire numero o una stringa o un oggetto o un'altra funzione (!) a seconda dei casi. Diciamo che il tipo di ritorno è "misto", in questo caso. In Javascript puro non abbiamo modo di forzarlo.
- un metodo in Java può specificare i suoi parametri e avrà sempre la certezza di riceverli. A una function Javascript possiamo passare più o meno parametri di quelli che si aspetta, e lei può decidere di gestirli come preferisce.
- in Java abbiamo la certezza del tipo dei parametri, in Javascript no.

Affronteremo tutti questi problemi in seguito. Per ora notiamo che possiamo creare una funzione con due sintassi diverse, la sintassi di metodo e quella di variabile:

```
function double(x){return x*2;}  
var double = function(x){ return x*2;}
```

In entrambi i casi, double(4) restituirà 8.

Notiamo che si tratta comunque di una variabile, e come tale potrà essere trattata. Ad esempio, possiamo passare una funzione a un'altra funzione, che la richiamerà:

```
var double = function(x) {return x*2;}  
var triple = function(x) {return x*3;}  
var poly = function(f1, f2, x){ return f1(x) - f2(x);}  
poly(triple,double,3);  
// 3  
poly(double,triple,3);  
// -3
```

Questo ci permette di passare codice al codice, ed è estremamente importante nella pratica, come vedremo in seguito.

Impariamo a distinguere fra l'atto di richiamare la funzione e l'atto di passarla o assegnarla. Ad esempio:

```
var f = double;  
f(5); // restituisce 10  
typeof f; // "function"
```

In questo caso ho assegnato una funzione (double) a una variabile f. Ora f "punta" a una funzione, e richiamare f significa richiamare double. Se invece avessi scritto var f = double(5); f sarebbe stato uguale al numero intero 10.

Una funzione f può essere usata come variabile (quindi assegnata, passata come parametro, ecc...) o come codice, quindi eseguito. Potente pensare le parentesi tonde come l'operazione di invocazione per la funzione f. f() equivale a "EXEC f".
Ora andiamo a esplorare la natura "anarchica" di Javascript:

```
double(5,15, "pippo"); // restituisce 10. I parametri extra vengono ignorati, ma non vengono forniti errori di compilazione  
double("pippo"); //NaN, un tipo specifico che indica "not a number". Succede quando il risultato dell'operazione non è un numero, come invece ci aspettavamo fosse. Abbiamo passato un parametro di tipo sbagliato, ma Javascript non ha battuto ciglio  
double("5"); // number, 10  
double(); //NaN. In questo caso non abbiamo passato proprio il parametro. Javascript ci ha restituito il valore come se niente fosse.
```

La rigidità di Java ci costringeva se non altro a fornire i parametri del tipo corretto (NullPointerException permettendo). Javascript non offre garanzie sul tipo o sulla presenza dei parametri. Ne approfittiamo per imporre dei controlli alla function:

```
function double(x)  
{  
    if(x!=undefined || isNaN(x))  
        throw "A valid number is required to work within function double";
```

```
    return x*x;
}
```

isNaN si legge " is not a number", e restituisce true se il valore dentro x non è, neanche con parecchia fantasia, considerabile un numero. isNaN("5") è false, isNaN("pippo") è true.

throw è qualcosa a cui siamo abituati, ma in Javascript possiamo fare throw di stringhe o oggetti. Il concetto di eccezione è meno formalizzato di quanto non sia in Java. In generale, se vogliamo interrompere l'esecuzione e restituire un qualche messaggio di errore, possiamo fare throw. Esiste il corrispondente try-catch, che vedremo in seguito.

Abbiamo poi un ultimo modo di definire una funzione, con una sintassi ridotta che offre anche dei vantaggi in fase di programmazione a oggetti. Questa tecnica viene detta "arrow function", e sintatticamente può essere espressa come (i1,i2,i3...in)=>{blocco di codice}, oppure, nel caso di funzioni estremamente semplici, come (i1,i2,i3...in)=>(espressione di ritorno). Vediamo qualche esempio:

```
var f = (a,b)=>(a+b);
f(5,6); // 11
var fib = (n) => (n<2 ? 1 : fib(n-1) + fib(n-2));
fib(5);
var greeter = (name) => {console.log("Hello"); console.log(name);}
greeter("James");
```

Vedremo in seguito come questa sintassi velocizzi di molto alcune operazioni comuni.

18.5 Gli oggetti in Javascript

In Java un oggetto è un caso particolare di una classe. Non esistono oggetti senza classe, mentre possiamo avere classi senza oggetti corrispondenti.

In Javascript, ogni oggetto è potenzialmente un mondo a sè. Possiamo crearlo semplicemente tramite la sintassi:

```
o = {chiave:valore, chiave:valore...}  
a = {k:5}
```

L'oggetto a ha una proprietà k che contiene il valore 5. L'espressione a.k viene valutata a 5. Notiamo che "a" non ha una classe di riferimento. E' un oggetto con una proprietà, k, e può anche averne altre. Possiamo aggiungerne in itinere.

```
a.j= 10;  
console.log(a);  
// stampa : {k: 5, j: 10}  
typeof a; // "object"
```

Questo è impensabile in Java, dove ogni oggetto deve rispettare una classe, che è in effetti un "contratto" che garantisce che quell'oggetto avrà un dato funzionamento. In effetti, gli oggetti Javascript sono in realtà mappe con chiave string e valore arbitrario. Le chiavi non si possono ripetere, mentre i valori sì, come in tutte le mappe. Un metodo di un oggetto non è altro che una chiave associata a una function:

```
a.x = function(){return "ciao";}  
a.x(); // restituisce "ciao"
```

E siccome i tipi delle variabili sono dinamici, un "metodo" può diventare una proprietà: a.x = 5;

Gli oggetti senza classi sono comuni in Javascript e spesso ci permettono di semplificare il ritorno di un metodo, scrivendo un oggetto con le chiavi necessarie per rappresentare il ritorno. Gli oggetti si possono vedere anche come "vettori associati" (un altro nome per le mappe), ed è possibile accedere alle loro proprietà (o metodi, visto che la differenza è sottile) tramite la sintassi vettoriale:

```
a.k; // valore di k  
a["k"] // valore della chiave k  
a["x"](); //invoca il metodo "x" dell'oggetto a, ammesso che alla chiave "x" corrisponda una funzione
```

18.6 Vettori

Un vettore in Javascript è solo un tipo particolare di oggetto. Per essere precisi, è un oggetto derivato da un particolare prototipo (riporteremo documentazione in seguito), e non corrisponde al concetto di vettore di Java.

I vettori in Javascript sono in realtà più simili a delle liste di oggetti. Possono aumentare di dimensione, ridursi, possiamo estrarre elementi o inserirne in punti specifici. Una serie di esempi serviranno a chiarire:

```
var a = [] ; // vettore vuoto
a.push(5); // aggiungere 5 alla fine di a.
console.log(a) ; // [5]
a[2]=4; ;
console.log(a) ; // [5, empty, 4]; il vettore ha creato la posizione 1 quando noi abbiamo specificato la 2. Non ci possono essere buchi di posizione, ma possiamo avere buchi di contenuto
a.push("Pippo") ;
console.log(a) ; // [5, empty, 4, "Pippo"]
// abbiamo oggetti diversi!
a.pop(); // estrae e restituisce l'ultimo elemento, "Pippo"
a.splice(1,1) ; // elimina un elemento partendo dalla posizione 1, togliendo quell'undefined
console.log(a) ; // [5,4]
a.length ; vale 2
```

18.7 Condizioni

Come abbiamo detto prima, le condizioni in Javascript non sono chiare come possono esserlo in Java. In Java, una condizione è una espressione che restituisce un booleano. Lo stesso vale per Javascript, ma Javascript è meno schizzinoso relativamente a quello che ritiene essere "booleano".

Javascript cercherà di interpretare tutto quello che mettiamo al posto di una condizione (quindi in un if, un while, un do-while, la condizione di reiterazione di un for) come fosse un booleano. Questo ci porta ad avere valore "truthy" e "falsy".

Una espressione "conta" come falsy (e viene valutata a false in una condizione) quando produce la stringa vuota (" o ""), 0, il valore false, null, undefined o NaN. Tutto il resto conta come true, o meglio, truthy. Questo ci porta ad avere risvolti interessanti:

```
if(1) console.log("vera");
if(0) console.log("falsa");
if("") console.log("falsa");
```

Vedremo che dovremo fare attenzione al verificare di avere ricevuto i parametri corretti nelle funzioni. Un parametro col valore 0 verrebbe valutato come falso scrivendo if(param), e la funzione potrebbe dare una eccezione non voluta in questo caso.

Un altro elemento da considerare è l'operatore di uguaglianza. Javascript ha due tipologie di uguaglianza, == (usato per l'uguaglianza di valore, molto loose) e === (uguaglianza stretta, pari valore e contenuto). Vediamo qualche esempio:

```
"1"==1 ; // true
"1"===1; // false
"1"==[1] // true
```

Nella pratica spesso ci basterà l'uguaglianza loose (==), che possiamo applicare senza problemi alle stringhe, ma di quando in quando vorremo verificare che anche il tipo sia quello previsto, nel qual caso === potrebbe essere la scelta giusta.

Valgono per il resto le forme e gli operatori logici a cui siamo abituati: !, ||, &&, con le stesse meccaniche viste in Java.

18.8 Selezione

Le forme di selezione sono analoghe a quelle di Java. Disponiamo del consueto if con o senza else, identico a quello di Java con le problematiche relative a truthy / falsy, dello switch e del ternario.

Vale la pena di notare che lo switch utilizza l'operatore di uguaglianza stretta (`==`), e quindi il valore "checkato" deve essere dello stesso tipo e dello stesso tipo per essere riconosciuto. Per il resto, non si comporta diversamente rispetto a quello di Java.

Vale anche il discorso relativo alla visibilità delle variabili nei blocchi di codice degli if/else o in quelli annidati negli switch, a patto di usare let per dichiararle.

Riportiamo di seguito tre esempi, sostanzialmente identici a quanto visto in Java:

```
if(age>=18)
    status = "adult";
else
    status = "not an adult";

status = age>=18 ? "adult" : "not an adult";

switch(profession)
{
    case "teacher":
        discount = 2;
    break;
    case "student":
        discount = 1;
    break;
    default:
        discount = 0;
}
```

18.9 Iterazione

L'iterazione in Javascript non è molto diversa da quella vista in Java.

Disponiamo dei consueti while e do-while, sostanzialmente identici a quanto visto in Java, e del for classico in tre parti (inizializzazione, condizione di ripetizione e istruzione di avanzamento).

Il for nella forma for-each, usato in Java sugli Iterable, non è presente in tutte le versioni di Javascript, ma disponiamo di un comodo for-in per scorrere le proprietà di un oggetto:

```
a = {a:4, b:3}
for(var k in a) console.log(k+":"+a[k]);
// a:4
// b:3
```

Notiamo che nel for abbiamo usato la notazione vettoriale. a[k] significa "il valore della proprietà di nome k in a". k vale "a" e "b" rispettivamente nelle due ripetizioni del for.

18.10 Ereditarietà classica

Javascript simula l'ereditarietà classica, per quanto il concetto di "classe" in Javascript sia "debole" e il suo meccanismo di ereditarietà reale sia prototipale. E' possibile definire delle classi ed estenderle con una sintassi analoga a quella che abbiamo visto in Java.

```
class Person
{
    constructor(name,surname,age)
    {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }
    adult()
    {
        return this.age>=18;
    }
}

let james = new Person("James", "Watson", 44);
james.adult(); //true
james.hobby = "Chess"; //nonostante James sia stato creato a partire da una classe, possiamo sempre aggiungere nuove chiavi, o cambiare il
valore delle vecchie. james è sempre una mappa con chiave stringa e valore arbitrario
```

A differenza che in Java, l'uso della parola this per indicare le proprietà è obbligato. Se avessimo scritto age invece di this.age, age sarebbe risultato undefined.

Notiamo anche che non abbiamo dovuto dichiarare le proprietà dell'oggetto Person. E' stato sufficiente assegnarle per crearle. Le proprietà sono tutte pubbliche e visibili, non essendo disponibile un encapsulamento paragonabile a quello di Java.

Per fine, costruttore si chiama sempre e comunque "constructor". Non riprende il nome della classe.
Vediamo ora un esempio di ereditarietà:

```
class Worker extends Person
{
    constructor(name,surname,age,job)
    {
        super(name,surname,age);
        this.job = job.toLowerCase();
    }
    adult()
    {
        return this.age>=18;
    }
    hasDiscount()
    {
        return this.job=='teacher' || this.job=='student';
    }
}

jill = new Worker("Jill", "Smith", 28, "Police");
jill.hasDiscount(); // false

//Javascript dispone anche dell'operatore instanceof:
```

```
jill instanceof Person; // true  
jill instanceof Worker; // true
```

`instanceof` tornerà comodo per verificare che ci sia arrivato un oggetto della "classe" corretta all'interno dei metodi. Le virgolette sono d'obbligo. Le classi non esistono davvero in Javascript, ma nascondono un meccanismo noto come ereditarietà prototipale.

Uno studio dell'ereditarietà prototipale esula dagli scopi di questo corso. Un buon link, in lingua inglese, è il seguente: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain, che ripercorre i concetti visti fino a ora e spiega il meccanismo di prototipazione. Difficilmente ci troveremo a usarlo nella pratica.

18.11 Il DOM

Javascript non è pensato per vivere "nel vuoto". E' pensato per essere integrato nelle pagine Web, nei files HTML. In effetti, possiamo immaginare Javascript come lo "spirito nella macchina", il programma che governa la pagina web.

Quelli che per HTML sono elementi ed attributi per Javascript sono oggetti. Javascript "vede" la pagina Web (e può modificarla) come un albero costituito da oggetti, ciascuno con metodi e proprietà specifici.

Cominciamo con un esempio pratico:

```
//HTML
<form name="form1">
    <input type="text" name="a" />
</form>
<script>
    //Javascript
    form1.a.value = "Hello Javascript World";
</script>
```

Le prime tre righe (la form) hanno creato degli oggetti HTML. Le ultime tre righe (senza considerare i commenti) hanno usato gli oggetti (perchè di quello si tratta) in Javascript, all'interno del tag SCRIPT. Per la precisione, cosa vuol dire form1.a.value?

form1 identifica la form che abbiamo creato. In questo caso la identifichiamo tramite il nome, ed è un "trucco" che funziona bene per le form, meno bene per altri elementi.

form1.a è la consueta sintassi oggetto.proprietà. "Prendi la proprietà a della form form1". La proprietà "a" corrisponde all'input di tipo text posizionato dentro la form (name="a").

La casella di testo è a sua volta un oggetto, e ha una proprietà "value", corrispondente al suo contenuto. Scrivendo form1.a.value = "Hello Javascript World" stiamo modificando il contenuto della casella di testo.

Questo modo di lavorare viene detto "modifica del DOM". DOM sta per "Document Object Model", vale a dire "un modello standard per leggere e modificare un documento". Una maniera più informale di definire il DOM potrebbe essere "la pagina Web vista come un albero di oggetti".

Nella pratica abbia identificato un oggetto nella pagina (form1), un oggetto dentro il primo oggetto (la casella di testo di nome a), e la proprietà value (questa volta un primitivo, una stringa) dentro quell'oggetto, con una sintassi e delle regole standard che sono il fulcro del lavoro di Javascript.

Avgremmo potuto farlo in un altro modo, meno "furbo": document.forms[0].a.value = "Hello Javascript World";

document è un oggetto predefinito, corrispondente grossomodo all'intera pagina web. Selezioniamo all'interno di document un vettore, anch'esso predefinito, contenente tutte le form della pagina. Il vettore si chiama forms, e di questo vettore prendiamo il primo elemento (la nostra unica form).

Di quell'elemento prendiamo la proprietà di nome a, che corrisponde a una casella di testo. Di quella casella di testo prendiamo la proprietà value, e la impostiamo.

L'oggetto document dispone di diversi metodi pensati per selezionare, cercare e modificare elementi nella pagina, essendo document un "riflesso" programmatico della stessa. Quello che Javascript fa' a DOM si riflette sulla pagina.

In questo momento potrebbe sembrare poco utile. Potevamo ottenere lo stesso risultato in questo modo:

```
<form name="form1">
    <input type="text" name="a" value="Hello Javascript World" />
</form>
```

In effetti, le modifiche (o le letture) del DOM di solito avvengono in seguito a eventi. Anticipiamo un esempio di evento:

```
<form name="form1">
    a <input type="text" name="a" value="0" />
    b <input type="text" name="b" value="0" />
    <input type="button" value="calcola" onClick="somma.value = parseInt(a.value) + parseInt(b.value)" />
    risultato <input type="text" name="somma" readonly />
</form>
```

onClick è un attributo dell'elemento button, e si aspetta di ricevere un blocco di codice Javascript o una funzione. onClick, per essere precisi, viene detto "gestore di evento".

Intuitivamente, quando cliccheremo sul pulsante "calcola" la pagina eseguirà il codice Javascript corrispondente: prenderà la casella di testo di nome somma, ne prenderà il valore, e lo imposterà alla somma dei valori delle caselle di testo a e b.

Noterete che non ho avuto bisogno di scrivere form1.a.value nè form1.b.value. Questo codice funziona all'interno della form, essendo a e b "vicine", cioè contenute nella stessa form. Si tratta di un modo di fare comodo in certi casi, ma non universale.

In questo caso abbiamo letto dal DOM (partendo dal valore inserito dall'utente nelle caselle di testo) e lo abbiamo modificato di conseguenza (facendo comparire il risultato nella casella di testo di nome somma).

Su onClick e in generale sugli eventi ci soffermeremo meglio dopo. Per adesso è necessario fissare alcuni concetti:

- Javascript può leggere o scrivere la pagina, cambiandone gli elementi, direttamente o in seguito ad eventi
- il DOM (Document Object Model) è il formalismo che usiamo per vedere e modificare gli oggetti presenti nella pagina
- ci sono diversi modi di vedere gli oggetti della pagina, diversi modi di riferirsi allo stesso oggetto (tag o attributo) e diversi oggetti predefiniti (document è il principale e più utile).

18.12 Metodi di uso comune del DOM

Il DOM ci offre una serie di metodi predefiniti per identificare gli elementi tramite i criteri più comuni. Il più classico è `document.getElementById("idelemento")`.

Questo metodo dovrebbe restituire un singolo oggetto, partendo dal suo id. Ad esempio, `document.getElementById('terms')` restituirà il DIV contenente i terms della nostra pagina per ordinare la pizza, e il seguente codice:

```
console.log(document.getElementById("terms").innerHTML);
```

Stamperebbe in console (Altri strumenti -> Strumenti per sviluppatore -> Console, nel browser) il contenuto HTML del div terms. Nel caso in cui non dovesse esistere un elemento con id terms il metodo restituirà una eccezione. `innerHTML` è una proprietà predefinita, specificata dal DOM per quasi tutti gli elementi.

Non avremmo invece avuto la proprietà "value", che è propria dei controlli (caselle di testo, select, textarea ecc...). E' una delle tante proprietà predefinite che lo sviluppatore Javascript impara a conoscere con la pratica, e su cui ci soffermeremo in seguito.

Un secondo metodo di uso comune è `document.getElementsByTagName`, che restituisce una collezione di oggetti del tipo specificato. Vediamo un esempio applicato al form del sommatore di prima (apriamo la console e facciamo copia-incolla):

```
let inputs = document.getElementsByTagName("INPUT");
for(let i=0;i<inputs.length;i++)
    console.log(inputs[i].name);
```

Il risultato è:

```
a  
b  
somma
```

La riga vuota non è un caso. L'input type=button non ha la proprietà name. La variabile input è un vettore di oggetti HTML che sono anche dei controlli (tag INPUT), e tipicamente hanno una proprietà name.

Isolare tutti i controlli di un documento è spesso utile, ad esempio per verificare che tutta la pagina sia stata compilata dall'utente e che non ci siano campi vuoti.

Ora lavoreremo in maniera meno brutale, approfittando anche di un altro concetto, quello di classe. Supponiamo di voler calcolare la media dei voti di uno studente. Per lo studente inseriremo il nome e i voti di italiano, matematica e inglese. La form potrebbe avere questo aspetto

```
<form name="form1">
    Nome <input type="text" name="nome" />
    Italiano <input type="number" name="italiano" class="voto" />
    Matematica <input type="number" name="matematica" class="voto" />
    Inglese <input type="number" name="inglese" class="voto" />
    <input type="button" value="calcola" />
    <input type="text" name="media" readonly placeholder="qui comparirà la media" />
</form>
```

Così com'è non è funzionante. Possiamo renderla funzionante definendo, di nuovo, l'evento click (onClick) sul pulsante calcola.

```
<form name="form1">
    Nome <input type="text" name="nome" />
    Italiano <input type="number" name="italiano" class="voto" />
    Matematica <input type="number" name="matematica" class="voto" />
    Inglese <input type="number" name="inglese" class="voto" />
    <input type="button" value="calcola"
          onClick="media.value = ((parseInt(italiano.value) + parseInt(matematica.value) +
                               parseInt(inglese.value))/3)"
        />
    <input type="text" name="media" readonly placeholder="qui comparirà la media" />
</form>
```

Utilizzando di nuovo il concetto di controlli vicini. Altrimenti, possiamo selezionare gli elementi marcati tramite una classe, in questo caso gli elementi con classe voto, tramite `document.getElementsByClassName`.

Abbiamo visto in precedenza che la forma tipicamente suggerisce la funzione. In effetti le caselle di testo con classe voto dovrebbero comportarsi nello stesso modo: dovrebbero avere valori compresi fra 0 e 10, e andranno sommati per calcolare la media. Rifattorizziamo l'esempio di prima:

```
<form name="form1">
    Nome <input type="text" name="nome" />
    Italiano <input type="number" name="italiano" class="voto" />
    Matematica <input type="number" name="matematica" class="voto" />
    Inglese <input type="number" name="inglese" class="voto" />
    <input type="button" value="calcola"
    onClick="
        let voti = document.getElementsByClassName('voto');
        let somma = 0;
        for(let i=0;i< voti.length;i++)
        {
            if(isNaN(voti[i].value) || voti[i].value<0 || voti[i].value>10)
                voti[i].value = 0;
            somma+=parseInt(voti[i].value);
        }
        media.value = somma / voti.length;
    "
    />
    <input type="text" name="media" readonly placeholder="qui comparirà la media" />
</form>
```

Il vettore voti contiene solo gli elementi con classe voto (italiano, matematica e inglese). Potrebbe sembrare più complesso rispetto alla soluzione precedente, ma provate a modificarlo aggiungendo una materia. Noterete che non dovete modificare di una riga il Javascript.

Notiamo che voti[i] è un controllo (la casella di testo che contiene il voto i-esimo), mentre voto[i].value è il contenuto della casella di testo (il voto vero e proprio).

18.13 Proprietà comuni degli elementi del DOM

Il DOM offre un numero elevatissimo di proprietà predefinite per gli oggetti creati, ma a noi ne serviranno poche. Andiamo a vederne rapidamente solo le basilari:

- **innerHTML**: restituisce o imposta il contenuto HTML di un elemento: `document.getElementById("p1").innerHTML = "<h2> Ciao mondo </h2>";`
Stamperà un h2 con "Ciao mondo" all'interno dell'elemento con ID p1 (un paragrafo in questo caso).
- **innerText** : restituisce o imposta il testo di un elemento. Il testo non contiene HTML. Dopo l'istruzione di prima, `document.getElementById('p1').innerText` vale "Ciao mondo" (senza h2). Se impostiamo tag HTML nel testo questi verranno stampati as-is, non interpretati.
- **value**: non presente sui paragrafi, ma su tutti i controlli. Vi si accede tipicamente con la sintassi `nomeform.nomecontrollo.value`, come abbiamo visto negli esempi precedenti: `form1.name.value = "Nick"; console.log(form1.surname.value);`
- **style**: è una proprietà complessa, un oggetto a sua volta, che contiene le informazioni di stile dell'elemento. Possiamo leggerle o impostarle. Vediamo il seguente esempio, applicato all'esempio della pagella dello studente:

```
let voti = document.getElementsByClassName("voto");
for(let i=0;i<voti.length;i++)
  if(voti[i].value <6)
    voti[i].style.color = 'red';
```

I voti non sufficienti compariranno in rosso dopo l'esecuzione di questo codice. La proprietà `color` è un attributo dell'attributo `style` che appartiene all'oggetto `voto[i]`.

18.14 Eventi in Javascript

Nell'esecuzione di un programma Java ci siamo abituati al concetto di sequenza, e di comando-esecuzione-comando-esecuzione... La pagina Web ragiona in termini differenti, "rispondendo" alle azioni dell'utente e anche, potenzialmente, a eventi esterni.

Questo modello viene detto "a eventi". Il codice non viene eseguito secondo un ordine pianificato dal programmatore ma in risposta a "eventi" verificatisi sulla pagina.

Un "evento" è, informalmente parlando, "qualunque cosa succeda alla pagina": uno spostamento del mouse, la pressione di un tasto, o perfino lo scadere di un timer precedentemente impostato. Di solito, è una interazione dell'utente con la pagina. Il programmatore è chiamato a definire la risposta della pagina tramite i gestori di evento.

Noi definiremo un evento in base al suo tipo (vedremo a breve una serie di tipologie diverse) e al suo target (l'elemento che lo ha scatenato o "ricevuto").

Abbiamo già visto un caso concreto: l'evento di tipo "click", gestito dal gestore di evento onClick e avente il bottone come target. L'evento viene scatenato dal click dell'utente sul pulsante.

Come per gli attributi, ci sono eventi che hanno senso solo su alcuni elementi. Tutti gli elementi possono definire un gestore di evento per l'evento click (l'utente può cliccare ovunque) e quindi avere onClick, ma ha poco senso definire "onChange" su un paragrafo (p), visto che l'utente non può cambiare il suo contenuto, e l'evento change si verifica al cambiamento del valore (value). I paragrafi non hanno l'attributo value, quindi ha poco senso definirlo.

Gli eventi sono particolarmente importanti per i controlli, come vedremo nell'esempio conclusivo per mettere assieme Bootstrap, Javascript e CSS.

18.15 Eventi comuni

Useremo Javascript per facilitare e validare l'input da parte dell'utente. A tale scopo vedremo alcuni eventi usati tipicamente con le form e con gli input.

- **click:** si verifica quando l'utente clicca su qualcosa, tipicamente un pulsante. Il suo gestore si chiama onClick, e lo abbiamo già visto in azione.
- **focus:** l'utente "entra" in un controllo, posizionandosi dentro una casella di testo, una select o una textarea, ma anche un pulsante. Ci può arrivare col mouse o con la pressione del tab, o in qualunque altro modo. Il controllo in questo caso viene detto "attivo". A volte ci interessa eseguire del codice quando l'utente "entra" nella casella di testo (ad esempio, mostrare un messaggio per chiarire l'input richiesto). Il suo gestore si chiama onBlur.
- **blur:** l'opposto di focus. L'utente ha lasciato la casella di testo, la select o la textarea, o qualunque altro controllo. Il gestore si chiama onBlur, ed è spesso il posto adatto per mettere controlli sul dato appena inserito.
- **change:** l'utente (o il programma stesso) ha cambiato il valore di una casella di testo, di una select o di una textarea. Il gestore è onChange. Anche onChange è un buon punto per mettere dei controlli sul dato inserito.

18.16 Un primo esempio

Ragioniamo sull'esempio della pagella dello studente. Vogliamo:

- che l'utente riempia completamente la form
- che inserisca voti compresi fra 0 e 10 nelle caselle di testo di classe voto. I dati errati dovranno essere segnalati cambiando il colore del testo (diventerà rosso)
- che la media venga calcolata automaticamente ogni volta in cui varieremo il voto, senza bisogno di premere su alcun pulsante

Ecco una soluzione ottenuta definendo degli eventi:

```
<form name="form1">
    Nome <input type="text" name="nome" />
    Italiano
    <input type="number" name="italiano" class="voto"
        value="0"
        onBlur="this.style.color = this.value>=0 && this.value<=10 ? 'green' : 'red'; ricalcolaMedia();"
    />
    Matematica <input type="number" name="matematica" class="voto"
        value="0"
        onBlur="this.style.color = this.value>=0 && this.value<=10 ? 'green' : 'red'; ricalcolaMedia();"

    />
    Inglese <input type="number" name="inglese" class="voto"
        value="0"
        onBlur="this.style.color = this.value>=0 && this.value<=10 ? 'green' : 'red'; ricalcolaMedia();"
    />
    <input type="text" name="media" readonly value="0" />
</form>
<script>
    function ricalcolaMedia()
    {
        let voti = document.getElementsByClassName("voto");
        let somma = 0;
        for(let i=0;i<voti.length;i++)
            somma+=parseInt(voti[i].value);
        form1.media.value = somma / voti.length;
    }
</script>
```

Ci sono diverse novità. Prima di tutto, cosa vuol dire quel "this" all'interno di onBlur? onBlur è un attributo della casella di testo, il gestore per l'evento blur, e "this" si riferisce all'elemento in cui siamo, in questo caso. Il controllo verrà eseguito ogni volta in cui l'utente esce dalla casella di testo, che abbia cambiato il valore o meno.

La seconda novità è il richiamo della funzione definita sotto. Non avrebbe avuto senso riscrivere quel codice tutte le volte, in tutte e tre le caselle di testo. Lo abbiamo riassunto in una function (una funzione senza ritorno, l'equivalente di un metodo void), che lavora sul DOM. Il DOM è sempre accessibile dalle funzioni, e in generale da tutto il codice Javascript.

Come soluzione è ancora piuttosto "rozza". Il codice Javascript è mischiato all'HTML, c'è comunque parecchia ripetizione. E' anche difficile da gestire, e dovendo aggiungere una nuova materia dovremo ricopiare tutto il codice. Ci sono soluzioni più interessanti:

```
<form name="form1">
    Nome <input type="text" name="nome" />
    Italiano
    <input type="number" name="italiano" class="voto" value="0"
    />
    Matematica <input type="number" name="matematica" class="voto" value="0"
    />
    Inglese <input type="number" name="inglese" class="voto"
```

```

        value="0"      />
<input type="text" name="media" readonly value="0" />
</form>
<script>
    function refreshForm()
    {
        this.style.color = this.value<0 || this.value>10 ? "red" : "green";

        let voti = document.getElementsByClassName("voto");
        let somma = 0;
        for(let i=0;i<voti.length;i++)
            somma+=parseInt(voti[i].value);
        form1.media.value = somma / voti.length;
    }
    // inizializzazione degli eventi
    let voti = document.getElementsByClassName("voto");
    for(let i=0;i<voti.length;i++)
        voti[i].onblur = refreshForm;

</script>

```

Meno immediato, ma più potente. Abbiamo separato Javascript ed HTML, come si usa fare in molti framework (non tutti). Abbiamo di nuovo definito una funzione, ma questa volta non si occupa solo di ricalcolare la media, ma controlla anche il valore di... this.

Ma chi è "this"? Dipende. Dipende dall'oggetto a cui abbiamo assegnato la funzione. this diventerà l'oggetto a cui abbiamo assegnato quella funzione. Il for alla fine del tag script si occupa di questo.

Assegna (non richiama, assegna. Mancano le ()) la funzione refreshForm all'evento blur di tutti i voti. In questo caso, gli oggetti italiano, matematica e inglese ricevono, per la propria chiave onblur, lo stesso codice, e ciascuna lo esegue come se fosse proprio. Quindi, quando in italiano viene eseguito il codice this.style.color =..., questo si riferisce all'oggetto form1.italiano, la casella di testo in cui si trova.

Questo meccanismo ci permette di assegnare lo stesso codice a intere famiglie di oggetti (in questo caso, a tutte le caselle di testo con classe voto).

C'è un particolare importante da notare: Javascript è stato scritto sotto HTML. Il codice non avrebbe funzionato altrimenti. In Javascript stiamo impostando una chiave (onBlur) di alcuni controlli (italiano, matematica, inglese) che sono stati CREATI alle righe precedenti. Scrivere input type="text" name="italiano" corrisponderebbe in Java a qualcosa di simile a HTMLControl italiano = new HTMLInput("italiano", "text"); (sto inventando le classi).

Senza il costruttore prima, non avremmo avuto gli oggetti su cui lavorare. Di conseguenza, quando vogliamo impostare gli eventi in questo modo, Javascript deve essere scritto dopo l'HTML da governare.

18.17 jQuery

jQuery è una libreria Javascript pensata per velocizzare una serie di operazioni comuni nella pratica. Riprende tutti i concetti visti fino ad ora in Javascript e li arricchisce con una nuova sintassi, ma resta in tutto e per tutto backward compatible.

Tutto ciò che valeva per Javascript vale anche per jQuery, mentre jQuery offre un nuovo oggetto (l'oggetto \$, o "jQuery") contenente tutta una serie di utilities (metodi), pensati principalmente per la definizione rapida degli eventi e per la modifica e la navigazione del DOM.

Essendo una libreria, jQuery va installata o importata. Lo abbiamo già visto trattando Bootstrap, in realtà, e possiamo di nuovo appoggiarci all'importazione via web:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
```

Da riportare in head, come si usa in generale per gli import. A questo punto, anche se non è obbligatorio, si passa a definire un evento specifico, \$(document).ready.

18.18 L'oggetto \$, \$(document).ready e la definizione di eventi in jQuery

L'importanza di jQuery si traduce nell'avere a disposizione un oggetto che contiene la totalità delle funzioni offerte dalla libreria. Questo oggetto, per brevità, ha il nome di "\$", ma ha anche come alias "jQuery". Si può anche leggere come "selettore", e ora vedremo perchè.

Quasi tutti i progetti jQuery partono definendo gli eventi di interesse per la pagina all'interno di un altro evento, l'evento "ready".

L'evento ready si verifica quando l'intera pagina è stata caricata e tutti gli elementi sono stati creati. Questo elimina il bisogno di chiedersi se disponiamo degli elementi su cui lavorare, come accadeva invece nel caso di Javascript "liscio".

Il gestore dell'evento ready, il cui target è il document, si scrive in questo modo:

```
$(document).ready(function()
{
    // codice da eseguire
});
```

Si tratta di una sintassi barocca ed elegante allo stesso tempo. Si legge come \$ ("selettore", quindi "seleziona") il document (\$(document)), e invoca il metodo ready (corrispondente all'evento ready, pagina caricata) a cui passeremo come parametro una function anonima che verrà eseguita al caricamento della pagina.

Questo è un primo esempio pratico di una funzione che riceve in ingresso un'altra funzione. Sarà il caso comune con jQuery.
Noi passiamo la funzione, ma non sappiamo precisamente quando sarà eseguita. Sappiamo che verrà eseguita "quando la pagina sarà stata caricata per intero". Questo meccanismo, per cui passiamo funzioni che verranno eseguite "in un secondo momento" viene detto meccanismo delle callback.

Callback è anche il nome tecnico della funzione che passiamo, e in effetti è il meccanismo alla base di tutti gli eventi: specifichiamo o passiamo una funzione senza sapere quando (o se) sarà eseguita.

\$(document).ready è il posto ideale per definire gli eventi che governano la pagina. Potete pensarlo come un "init" della pagina, in cui definiamo il comportamento della stessa. Riscriviamo l'esempio precedente (la pagella dello studente) e approfittiamone per introdurre alcuni dei vantaggi di jQuery:

```
<script>
$(document).ready(function()
{
    //questa funzione è assegnata all'evento blur (è sufficiente scrivere blur, non onblur)
    //a "tutti gli elementi di classe voto"
    // quello che prima facevamo con Javascript e un for è solo una riga con jQuery
    $(".voto").blur(function()
    {
        //$(this) seleziono la casella di testo
        //che ha scatenato l'evento
        //$(this).val() -> invoco il metodo val() senza parametri
        //è come dire casellaDiTesto.getValue(); E' un getter.
        //salvo il valore in v.
        let v = $(this).val();

        //questa volta il metodo val() viene usato come un setter
        //un valore anomalo viene resettato a 0
    });
});
```

```

//$(this).val funziona da setter o getter a seconda che venga passato o meno un parametro.
//E' un trucco comune in jQuery, e non solo
if(v<0 || v>10 || isNaN(v)) $(this).val(0);

let somma = 0;
//un foreach in forma infissa.
//per ogni elemento di classe voto, esegui la funzione
// che passiamo come parametro di each
$(".voto").each(function(){somma+=parseInt($(this).val());});
//il $(this) dentro questo each è la singola casella di testo

//ma siccome siamo sempre in Javascript, possiamo usare la sintassi form.controllo.proprietà
form1.media.value = (somma / $(".voto").length);
});

});

</script>

<form name="form1">
  Nome <input type="text" name="nome" />
  Italiano
  <input type="number" name="italiano" class="voto" value="0"
  />
  Matematica <input type="number" name="matematica" class="voto" value="0"
  />
  Inglese <input type="number" name="inglese" class="voto"
           value="0"      />
  <input type="text" name="media" readonly value="0" />
</form>

```

Andiamo a spiegare i punti chiave:

`$(".voto")` restituisce una famiglia di oggetti "jQueryizzati".
`$(".voto").blur` è un metodo invocato sulla famiglia e permette di definire l'evento blur per TUTTI gli elementi della famiglia allo stesso tempo.
`$(".voto").blur(function(){...});` si legge "assegna questa funzione come gestore dell'evento blur a tutti gli elementi con classe .voto".

All'interno di quella function, troviamo scritto `$(this)`. Per quanto la regola sia stata applicata a una famiglia, a scatenare l'evento può essere stato un solo controllo per volto (non posso uscire da tutte le caselle di testo allo stesso tempo). `$(this)` è il nostro target, sempre "jQueryizzato".

`$(this).val()` è il metodo val() applicato al target. val() è un metodo particolare, che utilizza un trucco caro a jQuery. Se invochiamo val() senza argomenti, viene restituito il valore corrente della casella di testo da cui siamo usciti. Se invochiamo `$(this).val(n)`, impostiamo il valore della casella di testo da cui siamo usciti. In questo caso, sto usando val() per garantire che la form rimanga in uno stato coerente (voti compresi fra 0 e 10, niente valori non numerici).

Resta poi da calcolare la media. Questo viene fatto con un ennesimo metodo di jQuery, .each. `$(".voto").each(function(){})`; che si legge come "esegui ORA questa funzione su tutti gli elementi con classe voto". Non sorprende nessuno che vengano sommati i valori delle caselle di testo. Notiamo solo che `$(this).val()` dentro .each è diverso da `$(this).val()` fuori. `$(this).val()` dentro l'each prende i valori di tutte le caselle di testo di classe voto. `$(this).val()` fuori è il valore della sola casella di testo da cui siamo usciti.

Se provate a cancellare i commenti, noterete che le righe di codice sono pochissime. jQuery è estremamente sintetico e permette di separare ottimamente Javascript da HTML. Questo risulterà molto importante nei progetti di maggiori dimensioni.

Il selettore, che abbiamo usato diffusamente ma per ora solo per selezionare in base alla classe, può selezionare in base a qualunque criterio CSS di nostro gusto: `$("#id")`, `$(".classe")`, `$("P")`, `$("INPUT[type=text]")`, restituendo tipicamente collezioni di elementi su cui possiamo invocare i metodi di jQuery.

Ricapitolando:

- abbiamo definito document.ready, che definirà poi tutti gli altri eventi della pagina
- possiamo definire eventi per intere famiglie di elementi, senza bisogno di un for
- `$(this)` ha un significato radicalmente diverso in base all'evento in cui ci troviamo
- in jQuery alcuni (molti) metodi possono essere usati sia come setter che come getter di proprietà. Vedremo bene a breve.

18.19 Metodi di manipolazione del DOM in jQuery

jQuery offre una serie di metodi per manipolare il DOM (in sostanza la pagina) ragionando per famiglie di oggetti più che per oggetto singolo. Li riportiamo di seguito con una serie di esempi:

\$(selettore).val(), \$(selettore).val(x)

Restituisce il valore di un controllo o imposta il valore di uno o più controlli. Come abbiamo visto può essere usato come getter o setter. Vediamo qualche esempio:

```
$(".voto").val(0); // Imposta tutti i voti a 0  
$("input[type=text]").val(""); //svuota tutte le caselle di testo  
$("#media").val(); // restituisce il valore dell'elemento con id media. In questo caso è usato come getter
```

\$(selettore).html(), \$(selettore).html(htmlcontent)

Restituisce il contenuto HTML di un elemento o imposta il contenuto html di più elementi.

```
$(“BODY”).html(); // restituisce il codice di tutta la pagina  
$(“P”).html(“test”); // tutti i paragrafi ora conterranno “test” in grassetto  
/E’ equivalente a .innerHTML in Javascript “liscio”.
```

\$(selettore).text(), \$(selettore).text(textcontent)

Vale il discorso fatto per html, ma in questo caso impostiamo o ricaviamo il valore testuale, rimuovendo l'html (o rifiutandoci di interpretarlo)

\$(selettore).each(function(){})

Questo è più interessante. Possiamo applicare una stessa funzione a tutti gli elementi di una collezione, e si può vedere come la forma infissa di un foreach. Lo abbiamo visto in precedenza, nell'esempio sulla pagella dello studente, ma vediamone un altro:

```
let budget = $("#budget").val();  
$(".spesa").each(function()  
{  
    budget-=$(this).val();  
});  
$("#available").val(budget);
```

Questo semplice programma legge da una casella di testo con id budget (o comunque da un qualche elemento con id budget e una proprietà value) e poi procede a scorrere tutte le caselle di testo (o altri controlli) di classe "spesa", sottraendone i valori al budget.

Per terminare, fuori dal metodo (dal ciclo), imposta il valore di un controllo con id available al residuo.

Possiamo usare \$.each anche per ciclare e controllare famiglie di oggetti, verificando che il loro valore sia conforme alle aspettative (come visto in precedenza, nella pagella), e in generale per eseguire la stessa operazione su sottinsiemi del DOM.

\$(selettore).hide(), \$(selettore).show(), \$(selettore).toggle()

A volte vorremo mostrare o nascondere degli elementi in base alle scelte dell'utente. Ad esempio, potremmo decidere di mostrare un messaggio di errore se l'utente fa qualcosa che non ci piace (inserire un voto negativo, una spesa negativa, lasciare vuoto un campo quando ne esce).

jQuery mette a disposizione tre metodi: \$(selettore).hide() nasconde tutti gli elementi identificati dal selettore. \$(selettore).show() li mostra, mentre \$(selettore).toggle() modifica il proprio comportamento in base allo stato dell'elemento. Se l'elemento era nascosto, lo mostra, altrimenti lo nasconde. Potete pensarlo come un "interruttore". Vediamo qualche esempio:

```
$("#userguide").hide(); // nasconde la guida utente  
$("#userguide").toggle(); // nasconde o mostra la user guide  
$(".comment").hide(); // nasconde tutti gli elementi com classe “comment”  
$("#userguide").toggle("slow"); // mostra o nasconde la guida utente con una animazione “lenta”
```

Vediamo una applicazione pratica:

```
<form>  
Inserire il codice IBAN - <span onClick="$(‘#userguide’).toggle(‘slow’);”> Vedi / nascondi guida </span> <br />  
<input type=“text” name=“iban” />  
<div id=“userguide”>  
    Lorem ipsum sit dolorem,  
    Lorem ipsum sit dolorem,
```

```
        Lorem ipsum sit dolorem,  
        Lorem ipsum sit dolorem,  
        Lorem ipsum sit dolorem,  
        Lorem ipsum sit dolorem,  
    </div>  
</form>  
<script>  
    // la guida, contenente lorem ipsum a indicare la mia completa  
    // ignoranza in materia bancaria e fiscale, nasce "nascosta"  
    // cliccando su "Vedi / Nascondi guida" il div sottostante verrà visualizzato o nascosto  
    $("#userguide").hide();  
</script>
```

`$(selettore).addClass(x), $(selettore).removeClass(x), $(selettore).toggleClass(x)`

Aggiunge, rimuove o "toglia" classi di stile su elementi o famiglie di elementi, stilizzando interi pezzi di DOM con una sola riga:
`$("#input").addClass("error");`

19 - XML

19.1 Introduzione

XML (eXtensible Markup Language) non è un linguaggio di programmazione ma è un linguaggio pensato per trasmettere e conservare lo stato di oggetti sotto forma di un albero di elementi (tag) con attributi e contenuti.

La sensazione di déjà-vu è giustificata. XML è un caso più generale di HTML. La struttura del documento è la stessa, ma cambiano le finalità e i tag predefiniti: XML non ha tag predefiniti. Possiamo inventare tutti i tag che vogliamo, e tutti gli attributi che vogliamo, e il file XML resterà comunque "ben formato" (che è un concetto molto diverso rispetto a quello di validità).

Allo stesso tempo, XML non è pensato per avere una traduzione grafica (non è pensato per essere "renderizzato"). Aprire un file XML nel browser produrrà un albero esplorabile di cui sarà possibile ispezionare ogni ramo e ogni elemento, ma non produrrà immagini, corsivi, grassetti, icone ecc...

XML è pensato per esprimere il valore del dato (se avete già letto o pensate di leggere a breve MVC, lo stato del Model, il valore delle entities), non la struttura di una pagina. Il documento può essere visto come un insieme di oggetti privi di rappresentazione sotto forma di albero.

Oltre che per esprimere lo stato di un oggetto, XML viene usato anche per la scrittura dei files di configurazione per programmi complessi, soprattutto in ambito Java. Moltissimi framework cambiano radicalmente il proprio funzionamento sulla base di una riga di XML.

Un documento XML di prova potrebbe essere il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<employee id="1">
    <name>Jill</name>
    <surname>Smith</surname>
    <surname>Valentine</surname>
    <dateofbirth format="ddmmyyyy">01011986</dateofbirth>
</employee>
```

La prima riga viene detta "preambolo", e per quanto non strettamente necessaria per il funzionamento, è richiesta formalmente. Identifica il file come "file XML" e specifica la versione del linguaggio (con varianti non rilevanti per i nostri scopi).

La seconda riga è il tag radice, o elemento radice. Un documento XML deve avere un unico elemento radice che contenga tutto il resto. Gli elementi si possono ripetere: l'elemento employee contiene due volte il tag surname (come lo possiamo interpretare? non importa, è comunque permesso), mentre gli attributi non possono ripetersi all'interno dello stesso elemento, come in HTML, ma mentre HTML ce lo permetterà, XML segnalerà un errore.

Sempre in tema di differenza, in HTML sarebbe ammesso (ma criminale ed eticamente abominevole) chiudere i tag in ordine sparso, ad esempio:

```
<i><b>a</i></b>
```

Mentre in XML questo genererà un errore. I tag devono essere chiusi in ordine inverso di apertura: l'ultimo tag aperto è il primo a dover essere chiuso.

In merito al contenuto informativo del file, XML non è tipizzato e non ci sono vincoli. Avremmo potuto scrivere "ieri" come contenuto di dateofbirth e il file XML sarebbe stato comunque "ben formato". Vediamo un altro esempio:

```
<?xml version="1.0" encoding="UTF-8"?>
<employees>
    <employee id="1">
        <name>Jill</name>
        <surname>Smith</surname>
        <surname>Valentine</surname>
        <dateofbirth format="ddmmyyyy">01011986</dateofbirth>
    </employee>
    <employee id="2">
        <name>Jack</name>
        <surname>Winston</surname>
        <job>Engineer</job>
    </employee>
</employees>
```

Anche questo è un file XML "ben formato". Rispetta le regole di base: preambolo, una sola radice, tag chiusi nell'ordine giusto, niente attributi ripetuti, eppure dobbiamo notare la sua natura "anarchica". Il secondo impiegato ha un valore per "lavoro", il primo no. Il primo ha due cognomi, il secondo uno. Il secondo manca della data di nascita.

In generale, XML da solo non ci garantisce la possibilità di definire i dati che avremo a disposizione, né il loro tipo, e tanto meno ci permette di imporre dei controlli di validità.

Visto così potrebbe sembrare solo una versione verbosa dei nostri files CSV, ma non è questo il caso. XML è lo standard de facto per la trasmissione di informazioni industriali su internet, e lo rivedremo con un ruolo centrale parlando di web services.

XML in effetti serve come base per la creazione di linguaggi specializzati per la trasmissione di informazioni. Non avendo tag predefiniti, possiamo definire un nostro lessico e una serie di vincoli per decidere cosa sia valido (ora possiamo usare questo termine) per i nostri XML.

Questo viene ottenuto definendo un file di specifica che dettagli cosa è ammesso e cosa è vietato nei nostri documenti. Un linguaggio XML-derivato è definito a partire da un file di specifica che, da almeno dieci anni, è un XML egli stesso: un XML - Schema.

19.2 XML-Schema e il concetto di validità di un XML

XML da solo non può mai dirsi valido. Può dirsi al massimo "ben formato", se rispetta le regole che abbiamo visto sopra. Un XML può dirsi valido se rispetta un secondo file contenente delle regole. Questo file è scritto esso stesso in XML, e per essere precisi in un derivato di XML che prende il nome di XML-Schema.

XML-Schema, a differenza di XML puro, ha dei tag predefiniti e degli attributi predefiniti che servono a governare un numero arbitrario di files XML. Vediamo un esempio, partendo dal file XML (employee.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<employee>
    <name>John</name>
    <surname>Smith</surname>
    <job>Clerk</job>
    <salary>1400</salary>
</employee>
```

Potremmo voler definire le nostre regole di validità, le nostre regole di "linguaggio", a partire da questi dati. Ci aspettiamo di avere un unico tag employee come radice, composto da nome, cognome, mansione e salario. Il salario sarà un numero (magari intero?), mentre per il resto avremo stringhe. Potremmo decidere di rendere tutto obbligatorio.

Per essere sicuri che il file XML sia "valido", vale a dire rispetti queste regole, scriviamo di seguito un file XSD:

```
<xss:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  xmlns:xss="http://www.w3.org/2001/XMLSchema"
>
<xss:element name="employee">
    <xss:complexType>
        <xss:sequence>
            <xss:element type="xss:string" name="name"/>
            <xss:element type="xss:string" name="surname"/>
            <xss:element type="xss:string" name="job"/>
            <xss:element type="xss:short" name="salary"/>
        </xss:sequence>
    </xss:complexType>
</xss:element>
</xss:schema>
```

Si tratta sempre di un file XML, e ne segue le regole, ma i tag (che cominciano con xs:, non un caso) hanno un significato specifico. Questo XML-Schema, che potremmo chiamare employee.xsd, stabilisce che un file XML debba cominciare con un xs:element di nome employee, che contenga a sua volta una sequenza di elementi name, surname, job e salary. Per ogni elemento possiamo anche stabilire il tipo: notiamo che per salary è stato specificato il tipo "short", intero corto. Questo file è stato generato automaticamente a partire dall'XML da un generatore online gratuito: <https://www.freeformatter.com/xsd-generator.html#ad-output>, che offre per altro molte altre utility comode per gli sviluppatori web, ma può essere modificato agevolmente a mano.

Ad esempio, potremmo definire che salary debba essere positivo:

```
<xss:element type="xss:positiveInteger" name="salary"/>
```

In questo caso disponevamo di un tipo predefinito, ma siamo anche in grado di fornire restrizioni specifiche (da 2 a 5, o una scelta fra un elenco di valori possibili), che corrispondono al consueto lavoro di validazione che siamo abituati a fare sui file CSV.

A questo punto decidere se un file XML è valido significa "confrontarlo" con un XSD. Ci sono strumenti automatici che prendono in input un file XML e un file XSD e restituiscono un boolean: "sì o no", il file XML rispetta o meno quelle regole.

A livello teorico stiamo chiedendo se il file XML appartiene alla categoria specificata dal XSD. E' come dire XML instanceof XSD. Il file XML non può mai essere definito valido da solo, ma è valido in base a un XSD, e potrebbe non essere valido per una versione successiva del XSD.

I file XML validati tramite XSD a questo punto possono essere *parsati* (attraversati, scansionati) tramite librerie e metodi simili a quelli visti in Javascript per la manipolazione del DOM (si tratta sempre di un albero di elementi), e trasformati in oggetti o righe, con automatismi più o meno pronunciati.

XSD non viene tipicamente scritto a mano, ma creato tramite strumenti automatici, come quelli linkati in precedenza o quelli grafici presenti in Eclipse, e poi eventualmente personalizzati. Il programmatore ha bisogno più che altro di saper leggere e interpretare gli XSD, in quanto potrebbe essere chiamato a scrivere XML XSD-compliant, vale a dire che rispettino le regole di questo quel XSD, magari specificati da una grande azienda o un ente pubblico per essere sicuro di ricevere tutti i dati necessari a ogni comunicazione.

20 - Servlet e Web App

20.1 Introduzione ad HTTP

HTTP (Hyper Text Transfer Protocol) è il protocollo di rete alla base di quello che chiamiamo Web. Volendo semplificare si usano i termini web e internet come sinonimi, ma strettamente parlando il web è tutto ciò che si muove su HTTP, e che tipicamente viene fruito tramite browser.

Come si evince dal nome, HTTP nasce per spostare documenti. E' un protocollo, vale a dire un insieme di regole, piuttosto semplice, basato su due concetti: l'invio di una Request dal client a un server e la produzione di una Response che il server restituirà al client. La request corrisponde al nostro input, e la response è l'output. La response, in particolare, tende a essere una pagina web, uno dei documenti HTML / CSS / Javascript che abbiamo visto in seguito, per quanto questo non sia sempre vero.

HTTP nasce per trasferire ipertesti, vale a dire il tipo di documenti che abbiamo visto in precedenza, ma si è evoluto per trasferire qualunque tipologia di informazione, o quasi. Nella sua forma basilare, si limita a inviare un file HTML da un computer a un altro a seguito della request.

Una delle request più basilari potrebbe essere *GET /index.html*. La forma è verbo (get) - oggetto (index.html). E' quello che succede quando clicchiamo su un link: il browser invia qualcosa del tipo GET /indirizzodellink a un dato server.

La response "è" (in realtà contiene) il documento index.html, posto nella *radice* del sito web, o in una qualche altra posizione specificata dalla request o dalle configurazioni del server. Il file index.html viene consegnato al richiedente, che di solito è un browser e si occupa di renderizzarlo secondo le procedure che abbiamo visto nei capitoli su HTML e CSS.

Per HTTP, la request e la response sono due file di testo strutturati secondo determinate regole. Per Java, Request e Response saranno due oggetti specifici appartenenti a classi apposite.

Questo funzionamento elementare (richiedi file - fornisci file) è stato alla base dei primi giorni del web ed è ancora alla base di buona parte delle interazioni HTTP, ma non è più esclusivo. L'evoluzione successiva di HTTP è stata fornire accesso a programmi remoti, "facendo finta" che fossero documenti.

Vediamo un esempio di request di questo tipo: *GET /sommatore?a=5&b=10*. Questa request, incompleta, suggerisce che ci sia un documento di nome sommatore, ma non è questo il caso: sommatore è un programma. Il server HTTP (concetto analogo a quello di server MySQL visto in precedenza) non rimanderà al client il codice del programma sommatore, ma il suo *output*.

GET /sommatore?a=5&b=10 significa "passa al programma sommatore sul tuo server i parametri a e b, con valori 5 e 10 rispettivamente, e mandami l'output prodotto da quel programma". Questo avviene senza che HTTP se ne renda conto: per HTTP sommatore è un output che viene inviato al client, ma noi sappiamo che in realtà è un programma. Ora spiegheremo gli attori coinvolti in questo meccanismo e ne identificheremo i ruoli.

20.2 Elementi del protocollo

Per capire il funzionamento di HTTP, e di conseguenza delle web app, dobbiamo capire quali sono gli elementi che costituiscono il protocollo, e come variano nel loro comportamento.

Il primo elemento da definire è il *client*. Il client è un attore del sistema, ed è un programma che necessita di un server ubicato, potenzialmente, su un'altra macchina. Il client software, che abbiamo già visto in MySQL, in questo caso comunica col server software tramite una Request, e si aspetta di ricevere una Response.

In molti casi il client è un browser, e si aspetta una response renderizzabile o comunque interpretabile in qualche modo.

La *Request* è un file di testo per HTTP, un oggetto per Java e magari solo un click su un link per l'utente. Noi la ricorderemo come "tutte le informazioni di cui il server ha bisogno per fornirci il servizio".

Nella sua forma basilare, la request contiene almeno un verbo (ad esempio GET) e un oggetto (index.html), specificato tramite il suo indirizzo, assoluto o relativo. Spetterà al server decidere come interpretarla, e come rispondere, ma in generale il verbo ci dice cosa fare, e l'indirizzo su cosa farlo. Si tratta quasi sempre di operazioni di lettura o esecuzione, almeno per le web app.

In termini pratici, distinguiamo fra request verso contenuti *statici*, vale a dire documenti HTML (o di un qualche altro tipo) che vengono inviati tali e quali, e verso contenuti *dinamici*, vale a dire verso programmi, che vengono eseguiti e di cui ci viene restituito l'output. Il client non si rende conto della differenza: sarà il server a interpretare diversamente le request sulla base dell' *indirizzo* richiesto.

A un indirizzo (*URL*) può quindi corrispondere un documento (ad esempio la lista dei dividendi delle azioni svizzere di Aprile 2021) o un programma (un esempio su tutti: <https://www.google.it/search?q=http>).

Il *server*, o server web per essere precisi, riceve la request e decide cosa farne. Potrebbe decidere di rispondere con un "accesso vietato", a prescindere, con un "non trovato", con il documento che abbiamo richiesto o con un altro documento non correlato. Il client non ha controllo sulla risposta del server, solo sulla request.

Un server web può essere solo statico (quindi incapace di eseguire programmi e di restituirne l'output) o dinamico (il caso comune). Un server web dinamico ragiona sulla base dell'indirizzo che gli è stato richiesto: se è l'indirizzo di un documento restituisce il documento tale e quale. Se è un programma, lo esegue e restituisce il suo output (tipicamente sotto forma di HTML, CSS e JS).

In ambo i casi, parliamo di una *Response*. Una response contiene al suo interno il documento richiesto o l'output del programma di cui si era richiesta l'esecuzione (sommatore?a=5&b=10 probabilmente restituirà qualcosa del tipo 15), oltre a una serie di informazioni pensate per facilitarne la gestione e l'interpretazione da parte del client, come vedremo meglio nella parte sui web services.

Per il browser che la richiede la response è un documento, e può generare altre request a catena. Ad esempio, il tag script che richiede di includere un file js genera una request verso quel file js. Per noi sarà un oggetto da riempire gradualmente.

20.3 Servlet

Una *servlet* è un programma, ospitato su un *container* (un server web dinamico, nel nostro caso Tomcat). La servlet riceve la request che arriva dal client tramite Tomcat, e produce la response che tomcat invierà al client (tipicamente un browser). Prima di poter lavorare sulle servlet dovremo quindi configurare Tomcat e creare un progetto adatto a "girare" su Tomcat, vale a dire un Dynamic Web Project. Gli step sono dettagliati nell'appendice apposita, a cui rimandiamo lo studente. Proseguendo, supporremo di avere Tomcat installato e configurato.

Siamo evidentemente nel caso delle richieste di contenuti dinamici. La servlet dovrà occuparsi della creazione dei contenuti che verranno inviati al browser. Vediamo un esempio di una servlet minima:

```
package web;
// progetto TestSV
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

//mappatura: "dove" troverò questo programma
@WebServlet("/salutatore")
public class Test extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    public Test()
    {
        //costruttore. Per ora non ci interessa
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        response.getWriter().append("Hello world");
    }
}
```

Analizziamo una serie di punti chiave.

Prima di tutto, non abbiamo il main. Una servlet non ha un punto di partenza predefinito: il punto di partenza dipenderà dal verbo HTTP utilizzato per chiamare il programma. Se utilizziamo GET (come abbiamo visto fino ad ora), verrà eseguito il metodo doGet. Se utilizziamo POST (il secondo verbo più comune, come vedremo a breve) verrà eseguito un metodo doPost che per ora non è definito: *una servlet può scegliere a quali verbi rispondere e rispondere in maniera differente a seconda del verbo*.

Successivamente notiamo che la servlet viene messa a disposizione dei client a un certo indirizzo. Questo indirizzo viene detto *mappatura* della servlet, ed è specificato tramite una annotation apposita: @WebServlet("/salutatore").

Notiamo che la servlet si chiama Test, mentre la mappatura è "salutatore". Partendo da questi punti, analizziamo due request. GET /Test restituirà un "codice 404", vale a dire pagina non trovata. GET /salutatore invece troverà la servlet, ed eseguirà il metodo doGet (GET -> doGet).

Qui c'è un elemento contro-intuitivo da considerare. doGet è un metodo void. Non dovrebbe avere un ritorno, eppure cosa succede se andiamo col browser a <http://localhost:8080/TestSV/salutatore>? Vedremo "hello world". Il metodo doGet ha prodotto un ritorno di qualche tipo, in maniera implicita.

20.4 I gestori di verbo e la produzione della response nelle servlet

Le servlet hanno una serie di metodi (doGet, doPost, doPut, doDelete, per citare i quattro più usati in ordine di uso) che potremmo definire gestori di verbo. All'arrivo di una request di un dato verbo viene eseguito il metodo corrispondente (get -> doGet, post -> doPost, ...). Il metodo è incaricato di produrre (in realtà riempire) un oggetto request.

Rivediamo il codice visto sopra e analizziamolo:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    response.getWriter().append("Hello world");
}
```

Il metodo doGet riceve due oggetti: una HttpServletRequest (leggermente la request vista come oggetto) e un HttpServletResponse (letteralmente, la response vista come oggetto).

Potremmo dire che la request è tendenzialmente piena (contiene tutto quanto necessario al server per capire i bisogni del client), e la response è vuota. Spetta al metodo(doGet in questo caso) riempire la response, e in effetti è quello che fa: response.getWriter().append(). Letteralmente: prendi la response, apri il writer della response (immaginate la response come un file su cui si possa scrivere e non vi sbaglierete di molto) e invoca il metodo append del writer.

Se immaginiamo la response come un foglio bianco, possiamo leggerlo come "scrivi sul fondo della response", sotto tutto quanto possiamo avere scritto in precedenza. In questo caso siamo fortunati: dovevamo scrivere solo una riga.

Alla fine del metodo, la response viene "ritradotta" in un documento e inviata al browser, che la renderizza. Questo è il funzionamento di base di tutte le web app: richiesta, elaborazione, produzione della risposta, invio della risposta.

Notiamo che il meccanismo di traduzione è implicito e il programmatore non se ne rende conto. Dobbiamo sapere noi, a priori, che ciò che avviene sulla response verrà inviato al client. Allo stesso modo, la request ci arriva già "popolata", vale a dire già contenente tutti i dati necessari per lavorare e produrre un risultato.

20.5 Uso basilare dell'oggetto HttpServletRequest: i parametri

A meno di casi estremi e patologici di solitudine le web applications (i programmi che utilizziamo tramite il browser, senza bisogno di installazione) non si usano per salutare.

Di solito si usano per condividere informazioni (i vari social) o male che vada per svolgere calcoli avanzati. Pensiamo anche semplicemente alla calcolatrice scientifica di Google.

Supponiamo di voler fornire il calcolo dei primi n numeri della sequenza di Fibonacci. Può essere un compito computazionalmente intenso, e magari vogliamo affidarlo a un computer centralizzato più potente, non a uno "sussidiario". Una soluzione servlet-based per i primi 100 numeri potrebbe essere la seguente:

```
package web;
// progetto TestSV
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

//mappatura: "dove" troverò questo programma
@WebServlet("/fib")
public class Fibonacci extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    public Fibonacci()
    {
        //costruttore. Per ora non ci interessa
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        int n = Integer.parseInt(request.getParameter("n"));
        int n2 = 1;
        int n1 = 1;
        response.getWriter().append(n2+"");
        response.getWriter().append(n1+"");
        for(int i=2;i <= n;i++)
        {
            int current = n1+n2;
            n1 = n2;
            n2 = current;
            response.getWriter().append(current+"");
        }
    }
}
```

La servlet è Fibonacci, il progetto è TestSV, la mappatura è fib. L'unico verbo definito è il get (doGet). Ma cosa è quel `request.getParameter?` Lo capiremo vedendo il modo in cui richiamare la servlet: GET /fib?n=10.

n=10 è parte del nostro input. Il nostro input, la nostra Request, è composto da verbo, indirizzo e ora una *querystring*. La querystring è tutto quello che troviamo dopo il ? nell'indirizzo, ed è un modo abbastanza comune di passare *parametri*.

Possiamo pensare ai parametri come agli argomenti di un metodo. In questo caso passiamo un solo parametro (n) con valore 10. Possiamo accedere ai parametri che ci arrivano tramite la request tramite (fra gli altri) il metodo `request.getParameter()`. In questo caso, `request.getParameter("n") = "10"`. Le virgolette non sono un caso: i parametri sono visti come stringhe, e per usarlo come intero sono stato costretto a parsarlo.

Formalmente, i parametri sono una mappa con chiave stringa e valore vettore di stringhe (< String, String [] >), ma "degenerano" spesso in una mappa < String, String >, (il vettore di stringhe contiene solo un valore e viene visto come una stringa). Pensiamoli come una mappa String->String e non sbagliheremo più di tanto.

Cosa sarebbe successo se avessimo inviato come request solo GET /fib ? Errore: `NullPointerException`. Il parametro n ha valore null se non specificato. GET /fib?n=pippo avrebbe generato una `NumberFormatException`. E così via: questo ci costringe a validare gli input che ci arrivano dal client, di cui non ci fidiamo mai.

Notiamo che abbiamo inviato (append) alla response stringhe in linguaggio HTML. È la scelta di default. La servlet pensa di stare parlando con un browser, e di conseguenza sceglie il linguaggio che probabilmente sarà capito.
E se avessimo avuto più di un parametro?

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
```

```

//ora voglio i numeri di fibonacci compresi fra a e b
int a = Integer.parseInt(request.getParameter("a"));
int b = Integer.parseInt(request.getParameter("b"));
int n2 = 1;
int n1 = 1;
response.getWriter().append(n2+"");
response.getWriter().append(n1+"");
for(int i=2;i <= b;i++)
{
    int current = n1+n2;
    n1 = n2;
    n2 = current;
    if(i>= a)
        response.getWriter().append(current+"");
}
}

```

Mi servono due parametri, due limiti: a e b. La request corretta sarebbe la seguente: GET /fib?a=3&b=10. La & commerciale serve a congiungere parametri. In questo caso la request presenta verbo GET, indirizzo fib, corrispondente alla servlet Fibonacci, a cui arriva una mappa con due parametri (a="3", b="10"). A questo codice di base possiamo aggiungere un controllo di validità:

```

// doGet
int a=0;
int b=0;
try
{
    a = Integer.parseInt(request.getParameter("a"));
    b = Integer.parseInt(request.getParameter("b"));
    if(a <= b) throw new Exception("Invalid parameters");
}
catch(Exception e)
{
    response.getWriter().append("Bad parameters a and b: expected integers with a <b");
    return;
}

```

Il return termina prima il doGet, e la response conterrà solo il nostro messaggio di errore.

Questo metodo di passaggio dei parametri viene detto "per querystring", ed è usato soprattutto quando vogliamo nascondere i parametri nei link, ma ci sono maniere più eleganti. La principale è l'uso delle form.

20.6 Le form e le request

Analizziamo la seguente form, che supponiamo essere nel file "fibform.html", sullo stesso server Tomcat della servlet Fibonacci, mappata a fib:

```
<form method="get" action="fib">
    Limite inferiore <br />
    <input type="text" name="a" />
    Limite superiore
    <input type="text" name="b" />
    <input type="submit" value="send" />
</form>
```

Ora possiamo ricollegare i fili. Ogni controllo genererà (verrà tradotto in) un parametro. La prima casella di testo genererà il parametro a, la seconda la b. *Nel momento in cui premiamo su submit, la form verrà trasformata in una request e inviata a un programma.* A quale programma? Lo leggiamo nella form stessa: l'action (il programma) è quello mappato a fib (la servlet Fibonacci). Il metodo ("verbo") è get. Questa form esegue il metodo doGet passando i parametri a e b, inseriti dall'utente in comode caselle di testo, non in una querystring.

Ciò nonostante, premendo su "send" (il submit), vedremo la querystring ricostruita come se la avessimo scritta a mano. In effetti, una form con method=get è sempre e comunque una request GET. Le request GET inviano i propri parametri tramite querystring, che essi arrivino da un link:

```
<a href="fib?a=3&b=10"> Numeri di fibonacci da 3 a 10 </a>
```

o da una form come abbiamo appena visto.

Resta il meccanismo di base: una form *diventa* una request nel momento in cui l'utente preme su submit. I suoi dati vengono tradotti in un oggetto request, che viene inviato al programma specificato tramite l'attributo action, e viene invocato il metodo corrispondente all'attributo method specificato sempre nel tag form.

Le form sono lo strumento principe per la comunicazione con le web application, e ne usiamo principalmente due tipologie: le form get e le form post.

Abbiamo appena visto un esempio di form get, che genera una querystring. Una form POST non attiverà il metodo doGet, ma la sua request andrà gestita nel doPost (ma ci sono metodi per ridirigerla al doGet, e li vedremo in seguito). La principale differenza è che non vedremo i parametri nella querystring, ma saranno "nascosti" all'interno della request, e non visibili nella barra degli indirizzi.
In termini pratici, cambia solo l'attributo method: form action="fib" method="post".

Quando si tratta di form, si preferisce usare il post al get, e questo porterebbe a dover scrivere il codice in due metodi diversi (ad avere due "main", con molte virgolette). Spesso lo si evita in questo modo:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //redirigo le richieste post al get
    doGet(request,response);
}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //uguale a prima
}
```

Questa tecnica viene detta redirezione della request. Una invocazione del metodo doPost si traduce in una al metodo doGet, cui vengono passati i nostri due oggetti chiave (request e response). Useremo una tecnica molto simile in seguito, per strutturare meglio il nostro sistema delle viste.

Per le web app ci limiteremo a usare questi due verbi (get e post), e quindi a dover scrivere al massimo due metodi (doGet e doPost) per la gestione delle request relative.

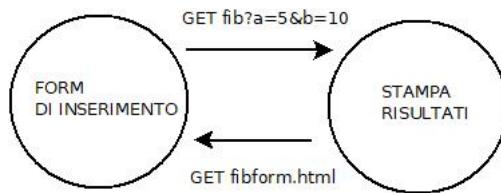
Come funziona la nostra applicazione, quindi?

Prima l'utente dovrà caricare la form, andando su <http://localhost:8080/TestSv/fibform.html> dal suo browser. Questo genererà una prima request get verso un contenuto statico (fibform.html), che verrà restituito tale e quale. Il browser lo renderizzerà in una form.

L'utente a questo punto compilerà la form inserendo i parametri a e b, e cliccherà su "send". A questo punto il browser eseguirà una seconda request (GET o POST, con parametri a e b) verso il programma mappato dall'indirizzo fib, vale a dire la servlet Fibonacci. La servlet produrrà il risultato e noi lo vedremo a schermo.

Potremmo dire che la nostra applicazione ha due stati: la form di inserimento e la produzione dei risultati. Sarebbe carino poter tornare alla form dopo la stampa dei risultati, e sarebbe anche piacevole non avere questo mixto di contenuto statico e dinamico, magari rendendo anche la form

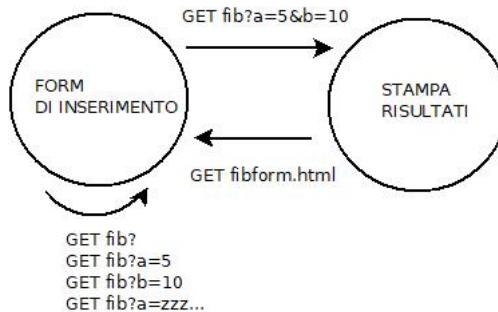
"dinamica". Sarebbe bello poter tornare alla form quando a e b non sono validi.
Graficamente, potremmo vedere l'applicazione come qualcosa di simile:



Ogni ovale rappresenta uno *stato dell'applicazione*, e ogni freccia una *transizione* fra stati. Le frecce equivalgono alle nostre request, e mappano il comportamento della nostra applicazione. Quando arriva un dato comando dobbiamo spostarci nello stato corrispondente. Se arriva GET /fibform.html andiamo alla form (richiesta di contenuto statico, per ora). Se arriva GET /fib?a=5&b=10 abbiamo una richiesta di contenuto dinamico, eseguiamo un programma e restituiamo un output. Il primo GET arriverà da una form (fibform.html), il secondo da un link. Ogni stato corrisponde a una schermata, vale a dire a una *vista* dell'applicazione.

20.7 Java Server Pages

Partendo dallo schema visto prima potremmo immaginare una applicazione più completa con questa struttura:



Abbiamo sempre due stati, ma c'è una transizione in più, per essere precisi una *auto-transizione*, una transizione che resta sullo stesso stato. Se la form invia dati corretti, stampiamo i risultati, altrimenti torniamo sulla form e, idealmente, dovremmo stampare un messaggio di errore. La servlet quindi dovrebbe comportarsi in questo modo.

```
//prima versione: pseudo codice
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    // 1 - prendo i parametri a e b
    // 2 - se a e b sono validi stampo i numeri di fibonacci per (a,b)
    // se non ci sono a e b, o non sono validi, torno alla form che dovrà stampare messaggio di errore ricevuto da me
}

//seconda versione: semi-funzionante
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    String aString = Integer.parseInt(request.getParameter("a"));
    String bString = Integer.parseInt(request.getParameter("b"));
    if(!_invalid(aString,bString))
    {
        //torna alla form e inviale un messaggio da stampare.
        //Questo metodo non esiste e non può essere realizzato con le nostre attuali conoscenze
        _tornaA("formfib.html", "Dati non validi");
    }
    else
    {
        List <Integer> fibs = _fib(Integer.parseInt(a), Integer.parseInt(b));
        for(Integer i:fibs)
            response.getWriter().append(i);
    }
}
```

Con l'eccezione del metodo `tornaA`, questo codice è abbastanza semplice da leggere. Se alla servlet arrivano i parametri, produci la lista (blocco `else`), altrimenti torna alla form mandandole un messaggio di errore. I metodi `_fib` e `_invalid` sono lasciati allo studente come esercizio di ripasso, ma crediamo che il concetto sia chiaro.

Adesso il problema è passare un messaggio di errore ("Dati non validi") a un file HTML.

Questo non è ottenibile in HTML puro. HTML è un documento statico che non gestisce i parametri, tanto meno messaggi arbitrari che gli arrivano dalla servlet, *che è invece il caso più comune*. Vorremo che HTML stampi i dati che gli arrivano dalla servlet.

Disponiamo di uno strumento specifico per questo scopo, per scrivere "HTML intelligente", che possa stampare dati ricevuti da altre sorgenti. Si chiama JSP(Java Server Pages), e somiglia, ma non lo diremo troppo forte, a un incrocio bizzarro fra HTML, Java e il linguaggio di programmazione orientato al web PHP.

JSP è il nome di una tecnologia e di un linguaggio che mischia Java ed HTML in un "documento intelligente", che potrà far di calcolo, gestire i parametri in ingresso e anche altri dati che gli arriveranno da altre sorgenti. Le pagine JSP risiedono sul server assieme alle servlet, e in effetti, come vedremo a breve, sono servlet travestite. Vediamo un esempio, il file `fibform.jsp`:

```
<%@
page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
%>
<!DOCTYPE html>
<html>
<head>
```

```

<meta charset="ISO-8859-1">
<title>Fibonacci Form</title>
</head>
<body>
    <form method="get" action="fib">
        Limite inferiore <br />
        <input type="text" name="a" />
        Limite superiore
        <input type="text" name="b" />
        <input type="submit" value="send" />
    </form>
</body>
</html>

```

Può sembrare un documento HTML, ma in realtà è un programma. Le varie righe (ad esempio la stampa di body) sono in realtà esecuzioni di response.getWriter().append(), che creano la pagina. Sono le prime righe a tradire la differenza: quella sintassi fornisce alla pagina le *direttive* di esecuzione.

Quale linguaggio devo usare per eseguire eventuali blocchi di codice? Java (attributo language dell'elemento page, che è una direttiva). Che tipo di documento devo produrre? Un HTML (attributo content-type, stessa direttiva). Che charset devo usare? E così via...

Per il resto, è tutto e solo HTML. In effetti, HTML è un sottoinsieme di JSP: tutto ciò che è valido in HTML (e quindi anche CSS e JS) è valido in JSP. Se posso scriverlo in HTML, posso scriverlo in JSP, mentre l'opposto non è sempre vero.
La differenza è che posso incorporare anche Java, in questo modo:

```

<%@
page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
%>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="ISO-8859-1">
        <title>Fibonacci Form</title>
    </head>
    <body>
        <form method="get" action="fib">
            Limite inferiore <br />
            <input type="text" name="a" />
            Limite superiore
            <input type="text" name="b" />
            <input type="submit" value="send" />
        </form>
        <%=request.getAttribute("errormsg")!=null ?request.getAttribute("errormsg").toString() : ""%>
    </body>
</html>

```

Perchè il codice funzioni, assicuriamoci di salvare le .jsp nella cartella WebContent del Dynamic Web Project.

Il tag %lt;%= si legge "stampa". La riga subito sotto la form si legge come "stampa il metodo `toString` dell'oggetto corrispondente alla chiave "errormsg" della mappa attributes dell'oggetto request, ammesso che il valore corrispondente a quella chiave esista.

E' un bel boccone da digerire tutto assieme, ma molti avranno capito che quel "errormsg" probabilmente conterrà il messaggio "Dati non validi", ottenuto in qualche modo dalla servlet.

Un programma JSP è in grado di recepire dei dati che le arrivano dalla servlet, e in effetti il codice del doGet, con l'ausilio di JSP, sarebbe il seguente:

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    String aString = Integer.parseInt(request.getParameter("a"));
    String bString = Integer.parseInt(request.getParameter("b"));
    if(!_invalid(aString,bString))
    {
        request.setAttribute("errormsg", "Dati non validi");
        request.getRequestDispatcher("formfib.jsp").forward(request,response);
    }
    else
    {
        List <Integer> fibs = _fib(Integer.parseInt(a), Integer.parseInt(b));
        for(Integer i:fibs)
            response.getWriter().append(i);
    }
}

```

```
    }  
}
```

Il punto chiave è `request.getRequestDispatcher("formfib.jsp").forward(request,response);`, e si legge "inoltra". Per la precisione, "inoltra la request (piena) e la response (vuota) al programma `formfib.jsp`", che si occuperà di terminare la creazione della response.

Potete immaginarlo come una chiamata a metodo. L'esecuzione passa dal `doGet` alla JSP, che è in realtà una servlet essa stessa: una servlet pensata per generare le viste e produrre principalmente HTML, e che si può scrivere con una sintassi particolare.
Non ci limitiamo neanche a passare la request così come ci arriva dal browser: la arricchiamo. Impostiamo un *attributo* della request.

Gli attributi sono una mappa `< String, Object >`, creata dal server (dalla servlet in realtà) per essere poi graficati dalle JSP. Sono il modo in cui comuniciamo alle JSP i dati da stampare. In questo caso, tornando alla form, avremo un attributo di nome "errormsg", contenente una stringa avente valore "Dati non validi".

`request.getAttribute()` restituisce il valore della chiave (lo abbiamo usato nella JSP, per stampare), mentre `request.setAttribute()` imposta il valore di una chiave nella mappa. In questo modo abbiamo potuto caricare una vista (`formfib.jsp`) e passarle dei valori calcolati altrove. E' il caso comune, e rispetta il pattern MVC: la vista riceve dei dati e li stampa. E' simultaneamente input e output, e funge da interfaccia verso l'utente, nascondendo tutto il resto.

A questo punto abbiamo gli strumenti necessari per scrivere una web application MVC, quindi possiamo cominciare a studiarne la struttura generale, applicando anche un altro pattern: Front Controller.

20.8 Struttura generale di una Web Application MVC Front Controller

Cominciamo dal caso visto in precedenza, generalizzandolo e dividendo come si deve le parti in un model, una vista e un controller.

Partiamo dal model. Il model dovrebbe essere qualcosa di autonomo, separato dal resto e riutilizzabile in altri programmi, anche non-web. Dovrebbe contenere l'accesso al dato (i vari DAO, ove presenti) e il generale le routine di calcolo. In questo caso possiamo cavarcela tranquillamente con una classe singleton:

```
package com.generation.fibonacciweb.model;

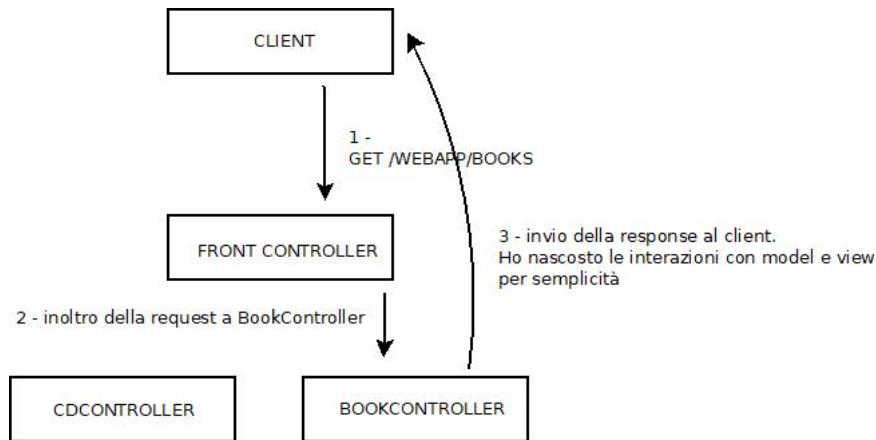
public class FibonacciSequence
{
    private FibonacciSequence instance = new FibonacciSequence();
    private FibonacciSequence()
    {
    }
    public static FibonacciSequence getInstance()
    {
        return instance;
    }
    public int getValue(int n)
    {
        return n>1 ? getValue(n-1)+getValue(n-2) : 1;
    }
    public List <Integer> getValues(int from, int to)
    {
        List <Integer> values = new ArrayList <Integer> ();
        for(int i=from; i<= to; i++)
            values.put(getValue(i));
        return values;
    }
}
```

Notiamo la nomenclatura del package: com.generation, il sito di generation invertito, per definire l'azienda di appartenenza, seguito dal nome del progetto (fibonacciweb), seguito dalla sezione del programma (model). Questa nomenclatura permetterà in seguito, quando useremo lo strumento Maven, di poter condividere i nostri package senza timore di duplicazione dei nomi.

Ora abbiamo un componente che produce i nostri risultati, non in maniera eccessivamente ottimizzata, ma comunque funzionante. Ora bisogna collegarlo al mondo: a un controller e indirettamente a una vista.

Il controller presenterà la principale novità. Per come abbiamo strutturato l'applicazione prima, avevamo due punti di entrata: la form (formfib.jsp) e la servlet vera e propria. Sarebbe bello avere un unico punto di accesso, un unico file centrale per controllare tutti gli accessi. Che io voglia la form o i risultati, o qualche altra pagina spuria, punterò comunque allo stesso indirizzo, fornendo eventualmente parametri diversi.

Questo concetto è quello di SAP, single access point, o di "Front Controller", se innestato sopra una struttura MVC. Il Front Controller sarà l'unico punto di entrata per la nostra applicazione, che poi delegherà l'esecuzione ad altri metodi o ad altri controller di "secondo livello". Nei casi più semplici si limiterà a eseguire codice diverso a seconda della request.



Il front controller in questo caso delega al book-controller, che poi invierà la request al client. Abbiamo nascosto per brevità le interazioni fra BookController e i vari DAO e viste.

Il Front Controller è un pattern architettonale innestato su MVC, e garantisce diversi benefici. Tutti i controlli di sicurezza potranno essere effettuati dentro il Front Controller, prima di passare ai controller di secondo livello o ai metodi di gestione, e lo stesso varrà per le operazioni di logging, di filtraggio e in generale di tutte quelle operazioni che vogliamo eseguire potenzialmente per ogni request.

Un altro grosso vantaggio è che ci fornisce un unico punto per definire il comportamento generico di tutta l'applicazione. Il front controller "tradurrà" o "implementerà" lo schema che abbiamo visto in precedenza (stati e transizioni: qualcosa di simile a una *macchina a stati finiti*). In pratica, finisce per somigliare molto ai nostri main:

```

package com.generation.fibonacciweb.controller;
// progetto "fibonacciweb"
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

//mappatura: "dove" troverò questo programma
@WebServlet("/index")
public class Fibonacci extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    public Fibonacci()
    {
        //costruttore. Per ora non ci interessa
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        String cmd = request.getParameter("cmd");
        if(cmd==null) cmd = "form";
        switch(cmd.toLowerCase())
        {
            case "result":
                _produceResults(request,response);
                break;
            default:
                _produceForm(request,response);
        }
    }
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        doGet(request,response);
    }
    private void _produceResults(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        //questo metodo dovrà produrre la lista dei numeri di fibonacci
    }
    private void _produceForm(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        //questo metodo dovrà produrre la form
    }
}

```

Per ora abbiamo riportato la struttura ma omesso l'implementazione dei due metodi di "gestione". Come per i programmi da console, stiamo lavorando in base a una variabile "cmd", che questa volta è un parametro.

Se non arriva un comando, o arriva un comando non riconosciuto, passeremo l'esecuzione a un metodo che crea una form. Se arriva il comando "result", passeremo l'esecuzione a un metodo che stampa i risultati. La servlet è com.generation.fibonacciweb.controller.Fibonacci, ed è mappata su server:8080/fibonacciweb/index, e funge da Front Controller. I due metodi privati (_produceResult e _produceForm) servono da "controller di secondo livello". Non è la soluzione più elegante, servirebbe avere dei controller (degli oggetti) separati, ma per un esempio così facile sarebbe stato uno spreco.

Nella pratica, il Front Controller (nel nostro caso in realtà il solo metodo doGet) smista le chiamate (le request) verso altri controller (in questo caso dei metodi, non degli oggetti, ma rende l'idea). Abbiamo uno smistatore, un front controller, che è *impropriamente* doGet (un metodo, non un oggetto) e degli esecutori (di nuovo dei metodi, non degli oggetti...).

Vedremo una struttura simile, ma molto più elegante e complessa, studiando Spring MVC. Ma ora vediamo di completare l'esempio:

```

private void _produceForm(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    request.getRequestDispatcher("formfib.jsp").forward(request,response);
}

private void _produceResults(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //mi aspetto di avere i dati necessari: un parametro a e un b, validi.
    //Se così non fosse, dovrei tornare alla form dando un messaggio di errore
    try
    {
        int from = Integer.parseInt(request.getParameter("a"));
        int to = Integer.parseInt(request.getParameter("b"));
        if(from <= to) throw new Exception("Bad pair");
    }

```

```

        request.setAttribute("list", FibonacciSequence.getInstance().getValues(to,from));
        request.getRequestDispatcher("results.jsp").forward(request,response);
    }
    catch(Exception e)
    {
        request.setAttribute("errmsg", "Invalid bottom and top. Expected two numbers with a <b>");
        _produceForm(request,response);
    }
}

```

Da questo codice lo studente ha un'idea di cosa succederà, ma ci sono ancora dei punti da chiarire. Cosa conterranno le viste (formfib.jsp e results.jsp)? Come si comporta la request e quali sono i suoi *rimbalzi*?

Ipotizziamo la seguente request: <http://localhost:8080/fibonacciweb/index?cmd=results&a=5&b=10>. Stiamo chiamando il metodo doGet della servlet Fibonacci mappata al percorso index del progetto fibonacciweb, il nostro Front Controller. Gli stiamo passando una request valida: c'è la variabile cmd, che servirà a dirgli cosa fare, e ci sono i due parametri a e b da cui dipende per capire quale sequenza di Fibonacci calcolare.

La request arriva a Tomcat e viene passata alla servlet. La servlet esegue il metodo doGet, e in particolare esegue il primo caso ("results") dello switch. Gli oggetti request (piena) e response (vuota) vengono passati al metodo _produceResults.

Il metodo _produceResults verifica che a e b siano valide (lo sono), e arricchisce la request con un attributo ("list", avente come valore la lista di interi prodotta dall'oggetto singleton di tipo FibonacciSequence). A questo punto passa la request (arricchita) e la response (ancora vuota) a un altro programma: results.jsp. results.jsp è una JSP, vale a dire una servlet travestita, e dovrà occuparsi di stampare i risultati, in questo modo:

```

<%@
page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
import="java.util.List"
%>
<%
//la servlet mi ha inviato una request arricchita, con questo attributo
//lo "prendo"
List< Integer > list =
    (List < Integer > ) request.getAttribute("list");
%>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="ISO-8859-1">
        <title>
            Fibonacci sequence for <%=request.getParameter("a")%> to <%=request.getParameter("b") %>
        </title>
    </head>
    <body>
        <h2> Fibonacci sequence for <%=request.getParameter("a")%> to <%=request.getParameter("b") %> </h2>
        <% for(Integer i:list){ %
            <%=i %><br />
        <% } %
        <a href="index"> Back to form </a>
    </body>
</html>

```

Ora cominciamo a credere al fatto che sia davvero un programma. Analizziamo i punti chiave.

Prima di tutto, abbiamo un import, nella prima direttiva. Ci serviva una interfaccia, List, e la abbiamo dovuta importare da java.util. Senza la JSP non avrebbe saputo usarla.

Successivamente, abbiamo creato una variabile (list), andando a recuperare il valore di una chiave dalla mappa attributes (< String, Object >) della request.

Quel valore è stato passato unidirezionalmente dalla servlet alla JSP, perchè la JSP lo possa stampare. Per usarlo come lista siamo stati costretti a castarlo (a List < Integer >) perchè la mappa degli attributi contiene Object. Dovremo sempre castare per usarli, salvo rarissimi casi. Notiamo che questo blocco non è di stampa (< % =), ma di codice puro (< %). Quello che mettiamo fra < % e % > è Java puro. Possiamo fare dei calcoli, e volendo perfino accedere ai database, ma viola qualunque idea anche solo recondita di MVC ed è punibile per legge in alcuni stati.

Le JSP possono contenere Java, ma dovrebbero usarlo solo per prendere i dati da stampare ed eventualmente per formattarli, come vedremo a brevissimo .

Sotto stampiamo dei parametri. Niente di nuovo: la JSP ha accesso alla request, quindi anche ai parametri, che sono dati di *input* che arrivavano dal browser. Li stampiamo con la consueta sintassi di stampa veloce (< % =). La sorpresa arriva sotto: apriamo un blocco di codice generico per scrivere un for, apriamo una graffa, vi inseriamo dentro un blocco di stampa e poi apriamo un blocco di codice Java per chiudere la graffa. All'interno del for stampiamo un risultato (un numero della sequenza di Fibonacci fra a e b) con un br (a capo) di fianco.

La sintassi può essere poco chiara. Ci sono maniere più eleganti di scriverlo, ma è anche piacevolmente sintetico. Questo è più o meno tutto l'uso di Java che faremo nelle JSP: cicli per stampare oggetti che arrivano dalla servlet (attributi).

In fondo alla pagina abbiamo il solito HTML, un link. Il link viene stampato una volta sola, non tante quanti sono i risultati: perché? Quello che abbiamo visto è un caso comune, che potremmo generalizzare in questo modo:

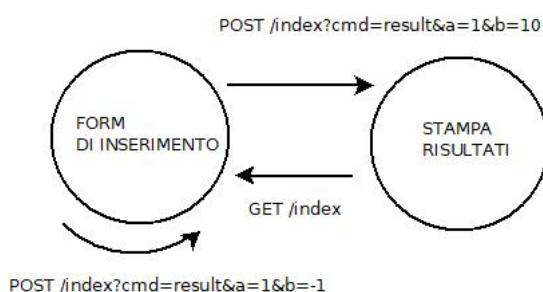
- Il browser invia una request (URL e parametri, vale a dire Map < String, String [] >) al server.
- La request viene intercettata dalla servlet e viene eseguito un metodo di gestione del verbo HTTP corrispondente (tipicamente doGet).
- La servlet si interfaccia col model per eseguire i calcoli, tipicamente passando alcuni dei parametri della request come argomenti dei metodi del model, e ottiene dei risultati. La servlet è il nostro *controller*.
- La servlet, ricevuti i risultati, li associa ad attributi della request (Map < String, Object >) e li invia alle JSP perché gli grafichino.
- La JSP grafica i risultati e genera la response che verrà inviata al browser, e attendiamo una nuova request.

Cosa sarebbe successo con questa request invece: <http://localhost:8080/fibonacciweb/index?cmd=results&a=x&b=10>? La request sarebbe passata dal metodo doGet al metodo _produceResults, ma lì avrebbe generato una eccezione. Saremmo finiti nel blocco catch di quel metodo, che *arricchisce la request* con un messaggio di errore, e poi invoca il metodo che produce la form. C'è differenza fra *invocare _produceForm con o senza attributi*. Lo vedremo presentando, come ultimo pezzo, il codice di formfib.jsp:

```
<%@  
page language="java"  
contentType="text/html; charset=ISO-8859-1"  
pageEncoding="ISO-8859-1"  
%>  
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="ISO-8859-1">  
        <title>  
            Fibonacci sequence for <%=request.getParameter("a")%> to <%=request.getParameter("b") %>  
        </title>  
    </head>  
    <body>  
        <form method="post" action="fib">  
            Lower limit <br />  
            <input type="text" name="a" />  
            Higher limit  
            <input type="text" name="b" />  
            <input type="hidden" name="cmd" value="results" />  
            <input type="submit" value="send" />  
        </form>  
        <%= request.getAttribute("errmsg")!=null ? request.getAttribute("errmsg").toString() : "" %>  
    </body>  
</html>
```

La form è, come la pagina dei risultati, una view. Uno strumento di input e di output. Potremmo arrivarci direttamente (/index, senza bisogno di parametri) o indirettamente (/index?cmd?result&a=1&b=-1, con una request errata). Nel primo caso non avremo la chiave errmsg, e non verrà stampata. In caso contrario, avremo un messaggio di errore che verrà stampato.

C'è una grossa differenza rispetto a prima: la form ha un campo nascosto, cmd. Serve a dire alla servlet cosa vogliamo ottenere mandandole questi dati. I campi nascosti sono molto usati nella pratica, e vengono valorizzati tipicamente dal programmatore, non dall'utente. Noi non inseriremo manualmente cmd=, a=, b= nella queryString. Verranno passate dall'form, e la nostra applicazione avrà quindi questa struttura:



20.9 Un esempio di web application CRUD: CensusWeb

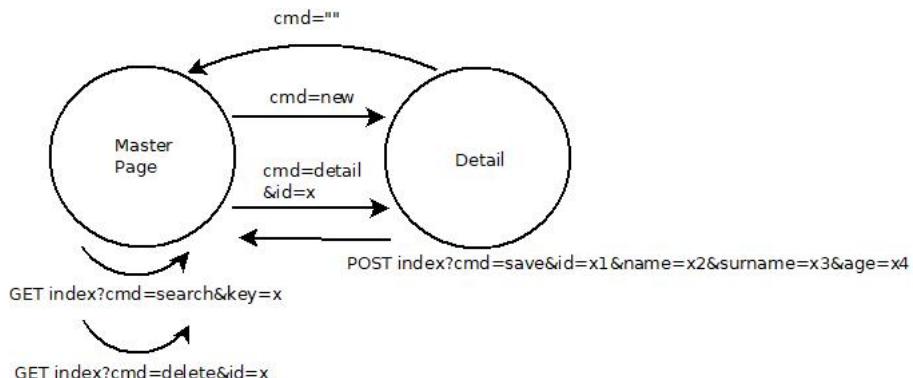
Prepariamo una semplice web application per la gestione di persone. Dovremo poter consultare un elenco di persone, svolgere ricerche basilari (operazioni di filtro), inserire, cancellare o modificare persone.

Potremmo avere una home page con un elenco di persone, ricercabile, e cliccando sul nome della persona potremmo andare a modificarla, o anche a cancellarla direttamente. Sempre dalla home page, potremmo creare nuove persone da gestire.

Potremmo chiamare la nostra applicazione CensusWeb, e dividerla in "caso", che corrisponderanno ai case dello switch principale sul parametro cmd del doGet, secondo l'esempio visto prima.

- **default:** pagina principale. L'elenco di tutte le persone, come da master page del mock up. Un esempio di request che porti a questa pagina potrebbe essere semplicemente /index, supponendo che il nostro front controller sia la index. Corrisponde a una operazione di read.
- **search:** pagina principale filtrata. Solo le persone che presentino, nel nome o nel cognome, il termine di ricerca. Serve anche il filtro, che potrebbe essere il parametro key. /index?cmd=search&key=Rossi. E' una operazione di read, seppur filtrata.
- **detail:** pagina di dettaglio della persona, modificabile, corrispondente al secondo mockup. Bisogna indicare l'id della persona. /index?cmd=detail&id=1 produrrà una pagina per modificare la persona. Si parla sempre di lettura (read).
- **update :** il comando save viene attivato premendo sul tasto save dalla schermata di detail, dovrà ricevere i dati per aggiornare una persona dalla rispettiva form . Aggiorerà la persona (o almeno ci proverà) e poi ci riporterà a detail. E' una operazione di update, che termina sempre sul dettaglio della persona. /index?cmd=update&id=1&name=John&surname=smith&age=35
- **delete :** cliccando sul bottone "x" accanto alla persona nella lista master (prima vista) invieremo il comando delete, che cancellerà la persona corrispondente e ci riporterà alla vista master. Sarà necessario inviare l'id: /index?cmd=delete&id=1 cancellerà la prima persona. E' la D di crud.
- **new :** la pagina master ha un pulsante "+", che invierà il comando new. new creerà una nuova persona con dati di default e ci porterà alla pagina di dettaglio per sostituirli con i dati corretti. E' la C di crud. /index?cmd=new

L'applicazione quindi potrebbe essere riassunta così:



E da questo schema è piuttosto semplice scrivere un controller e poi le relative viste, così come anche i DAO. Questo procedimento è top-down: sto lavorando immaginando di avere già i componenti, e poi andrò a scriverli a seconda delle mie necessità, di cosa dovranno fare.

```
package com.generation.censusweb.controller;
// progetto "censusweb"
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.generation.censusweb.model.entities.Person;
import com.generation.censusweb.model.dao.*;

@WebServlet("/index")
public class PersonController extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    //supponiamo di avere un DAO standard
    PersonDAO dao = PersonDAOFactory.getInstance().make();
    private final static String MASTERPAGE = "main.jsp";
    private final static String DETAILPAGE = "detail.jsp";
    public PersonController()
    {
        //costruttore. Per ora non ci interessa
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        String cmd = request.getParameter("cmd");
        if(cmd==null) cmd = "form";
        switch(cmd.toLowerCase())
        {
```

```

        case "detail":
            _detailPage(request,response);
        break;
        case "save":
            _savePerson(request,response);
        break;
        case "new":
            _newPerson(request,response);
        break;
        case "delete":
            _deletePerson(request,response);
        break;
        case "search":
            _searchPerson(request,response);
        break;
        default:
            _masterPage(request,response);
    }
}
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    doGet(request,response);
}
// restano da implementare i vari gestori dei comandi
}

```

Adesso analizziamo uno a uno i vari comandi e relativi gestori. Partiamo dal default, che deve corrispondere alla master page con tutte le persone a video. Il gestore potrebbe essere quello che segue:

```

private void _masterPage(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //carico tutte le persone
    request.setAttribute("list", dao.list());
    //le passo alla vista request.getRequestDispatcher(MASTERPAGE).forward(request,response);
}

```

La ricerca è estremamente simile.

```

private void _searchPerson(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //carico tutte le persone che rispettano il criterio (key) cercando per cognome. Se manca key le carico tutte
    if(request.getParameter("key")==null)
        request.setAttribute("list", dao.list());
    else
        request.setAttribute("list", dao.list("surname like '%"+key+"%'"));
    //UNSAFE!! attenzione, solo a scopo didattico
    //le passo alla vista request.getRequestDispatcher(MASTERPAGE).forward(request,response);
}

```

L'operazione di dettaglio è l'ennesima operazione di lettura, ma questa volta focalizzata solo su una singola persona:

```

private void _detailPage(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    try
    {
        int id = Integer.parseInt("id");
        request.setAttribute("person", dao.load(id));
        request.getRequestDispatcher(DETAILPAGE).forward(request,response);
    }
    catch(Exception e)
    {
        //in caso di errore per adesso torniamo semplicemente alla pagina principale
        _masterPage(request,response);
    }
}

```

Passiamo ora alla creazione di una persona, ricordando che il metodo save dovrebbe restituirci un id ove questo non fosse presente in precedenza:

```

private void _newPerson(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    Person emptyPerson = new Person();

```

```

        emptyPerson = dao.save(emptyPerson);
        request.setAttribute("person", emptyPerson); request.getRequestDispatcher(DETAILPAGE).forward(request,response);
    }
}

```

L'aggiornamento è più complesso. Riceveremo i dati dalla form creata in detail (come vedremo a breve) o da qualche altra sorgente e procederemo a salvarli:

```

private void _savePerson(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    Person person = new Person();
    person.setId(Integer.parseInt(request.getParameter("id")));
    person.setName(request.getParameter("name"));
    person.setSurname(request.getParameter("surname"));
    person.setAge(Integer.parseInt(request.getParameter("age")));
    // per il funzionamento di save, se la persona esiste già viene sovrascritta
    dao.save(person);
    _masterPage(request,response);
}

```

Abbiamo omesso il controllo degli errori per brevità. L'idea è che ci arrivano i dati da una form e li usiamo per ricostruire la persona. Terminiamo con la cancellazione.

```

private void _deletePerson(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    try
    {
        int id = Integer.parseInt("id");
        dao.delete(id);
        request.getRequestDispatcher(MASTERPAGE).forward(request,response);
    }
    catch(Exception e)
    {
        //in caso di errore per adesso torniamo semplicemente alla pagina principale
        _masterPage(request,response);
    }
}

```

Resta l'ultimo livello. Ci servono le due viste. Cominciamo dalla master page, main.jsp. La pagina riceve un elenco di persone, nell'attributo list, e deve graficarle. Una soluzione rossa ma efficace potrebbe essere la seguente:

```

<%
page language="java"
contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"
import="java.util.List,com.generation.censusweb.model.entities.*"
%>
<%
List< Person > list =
(List <Person > ) request.getAttribute("list");
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title> Census Web Master Page </title>
</head>
<body>
<h1> Census Web Master Page </h1>
<a href="index?cmd=new"> New </a> <br />
<form action="index" method="post">
    Look for <input type="text" name="key" />
    <input type="hidden" name="cmd" value="search" />
    <input type="submit" value="search" />
</form>
<% for(Person p:list){ %>
<div class="person">
    <%=p.getName() %> <%=p.getSurname() %> - <%=p.getAge() %>
    <a href="index?cmd=detail&id=<%=p.getId() %>"> Edit </a>
    <a href="index?cmd=delete&id=<%=p.getId() %>"> Delete </a>
</div>
<% } %>
<%=list.size() %> people found.
</body>
</html>

```

Per ogni persona generiamo un div di classe person (la classe di stile potrà essere definita in seguito) contenente i dati della persona, e due link (equivalenti a due request get quando cliccati), uno dei quali invierà il comando detail e l'id da modificare, mentre l'altro attiverà la cancellazione.

A questo si aggiungono due elementi. Il primo è la form di ricerca posta in cima alla pagina, che non viene ripetuta (è fuori dal for) e che, alla pressione di "search" (controllo di tipo submit) invierà una request post col parametro key e il parametro nascosto cmd=search.

Il secondo è il link "new", che attiva il comando "new" della servlet (il nostro front controller), che crea una persona per poterla poi modificare.

In totale la main page può generare $2n+2$ request diverse, dove n è il numero di persone passate dalla lista. Se arriviamo alla main page senza una selezione vedremo tutte le persone, altrimenti vedremo solo le persone che rispettano il criterio di ricerca. A tale scopo abbiamo stampato il numero di risultati sul fondo.

La pagina di dettaglio è raggiungibile solo tramite il comando detail presente nella main page, e dovrebbe permettere di modificare una persona già esistente (creata dal comando new). A differenza della master, riceve una sola persona, in un attribute (Object) che dovrebbe contenere una person. Dovrà usarla per riempire una form, che l'utente potrebbe volerci rimandare modificata.

```
<%@  
page language="java"  
contentType="text/html; charset=ISO-8859-1"  
pageEncoding="ISO-8859-1"  
import="com.generation.censusweb.model.entities.*"  
%>  
<%  
Person person = (Person) request.getAttribute("person");  
%>  
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="ISO-8859-1">  
        <title> Census Web Master Page </title>  
    </head>  
    <body>  
        <h1> Census Web Detail Page </h1>  
        <form method="post" action="save">  
            <input type="hidden" name="id" value="<%=person.getId() %>" />  
            Name  
            <input type="text" name="name" value="<%=person.getName() %>" /> <br />  
            Surname  
            <input type="text" name="surname" value="<%=person.getSurname() %>" /> <br />  
            Age  
            <input type="number" name="age" value="<%=person.getAge() %>" /> <br />  
            <input type="hidden" name="cmd" value="save" />  
            <input type="submit" value="save" />  
        </form>  
        <a href="index"> Back to main page</a>  
    </body>  
</html>
```

Questa form stampa i dati della person *dentro* gli input. L'utente a questo punto potrà modificarli, e premendo su "save" invierà i dati dell'utente, così come cmd=save, al front controller (la servlet mappata a index). Notiamo che invia anche l'input (è richiesto, come tutto il resto), ma lo lascia nascosto, non permettendo all'utente di modificarlo. E' un caso tipico. L'id *dovrebbe* restare stabile per non complicare la vita al db.

Sul fondo della form abbiamo un link piuttosto semplice, corrispondente a una request get che ci rimanda alla master page con l'elenco delle persone, nel caso in cui non si volessero modificare i dati.

Notiamo che questa sintassi ha dei problemi. I dati stampati in questo modo nelle caselle di testo non sono affidabili, come si vede con la pratica. Questo termina questo esempio minimo di applicazione. e' possibile rendere il tutto meno brutale utilizzando una estensione di JSP che prende il nome di JSTL.

20.10 JSTL

JSTL (JSP Standard Tag Library) è una libreria di tag di uso comune in JSP. Definisce una serie di tag utili che facilitano la scrittura delle JSP, e rivela la natura programmatica di JSP in generale: si parla sempre di programmi, quindi possiamo importare delle librerie.

Potremo scaricare il JAR (la libreria vera e propria) da questo indirizzo (fra gli altri) : <https://tomcat.apache.org/taglibs/standard/>, scegliendo JSTL 1.2. Ottenuto il JAR lo copieremo in WEB-INF/lib della cartella del Dynamic Web Project. A questo punto sarà possibile importare le componenti di JSTL dentro le nostre JSP.

Prima di procedere all'importazione, notiamo che JSTL è modulare. Non è una sola famiglia di tag, ma una collezione di famiglie. Noi ci concentriamo sulla famiglia core (prefisso c:).

Cominciamo con l'importazione:

```
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>
```

Queste righe, posto in cima nella JSP, importano i tag che ci interessano. Ora vediamoli in azione sulla pagina list di Census:

```
<%@  
page language="java"  
contentType="text/html; charset=ISO-8859-1"  
pageEncoding="ISO-8859-1"  
import="java.util.List,com.generation.censusweb.model.entities.*"  
%>  
<%@ taglib prefix = "c" uri = "http://java.sun.com/jsp/jstl/core" %>  
  
<%  
List< Person > list =  
    (List <Person > ) request.getAttribute("list");  
%>  
  
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="ISO-8859-1">  
        <title> Census Web Master Page </title>  
    </head>  
    <body>  
        <h1> Census Web Master Page </h1>  
        <a href="index?cmd=new"> New </a> <br />  
        <form action="index" method="post">  
            Look for <input type="text" name="key" />  
            <input type="hidden" name="cmd" value="search" />  
            <input type="submit" value="search" />  
        </form>  
        <!--  
            un forEach JSTL, appartenente alla famiglia di tag con namespace c:.  
-->  
        <c:forEach items="${list}" var="p">  
            <div class="person">  
                ${p.name} ${p.surname} - ${p.age}  
                <c:if test="${p.age>18}">  
                    <b>Adult</b>  
                </c:if>  
                <a href="index?cmd=detail&id=${p.id}"> Edit </a>  
                <a href="index?cmd=delete&id=${p.id}"> Delete </a>  
            </div>  
        </c:forEach>  
        <%=list.size() %> people found.  
    </body>  
</html>
```

In questo esempio vediamo un tag dedicato equivalente al foreach della famiglia c di JSTL (c:foreach), un tag dedicato a una stampa condizionale (c:if) e una sintassi di espressione \${}.

La sintassi \${qualcosa} si legge "espressione", e cela un trabocchetto. \${p.id} si traduce con <% = p.getId() %>: quando provo a stampare una proprietà la JSP proverà a richiamare il getter. Può essere stampata direttamente o usata all'interno di altri tag, come c:if.

Il tag c:if è una stampa condizionale. Valuta una condizione (attributo test) e visualizza il proprio contenuto solo quando la condizione è vera. La condizione è una espressione, e viene fornita con la sintassi {condizione}.

Il tag c:foreach, di contro, è una maniera comune di stampare liste o comunque contenuto iterato. Il suo uso più comune è quello appena visto: partendo da una sorgente di informazioni (oggetto iterabile {list}) definisce una variabile di scorrimento (attributo var, uguale a p) e ne stampa le proprietà.

Per i nostri scopi sarà sufficiente utilizzare c:foreach e c:if, ma JSTL offre molto di più e lo studente è invitato ad approfondire al link https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm, per avere una idea delle varie possibilità.

Altri tag verranno studiati e incorporati volta per volta a seconda delle necessità del progetto, ma il grosso del lavoro lo faremo con questi due, che

rendono meno brutale la sintassi di JSP standard, soprattutto quando si tratta di mischiare codice Java e HTML. Notiamo infatti che non c'è Java all'interno di JSTL, se non forse per la sintassi delle espressioni, quindi in misura molto limitata, e che JSTL si sposa in maniera naturale con HTML.

20.11 Session

Per completare il discorso sulle web-application è necessario parlare delle Session.

HTTP è, di suo, stateless. Questo vuol dire che ogni request è a sé: non esiste memoria delle request precedenti, e ogni request è usa-e-getta. Viene ricevuta da un metodo di gestione, processata e scartata.

Questo comportamento non ci dispiace in generale, ma a volte è limitante. Immaginiamo di doverci loggare su un sito, per eseguire operazioni riservate (ad esempio movimentare il conto in banca). Essendo le request prive di memoria, dovremo reinviare nome utente e password con ogni request, e questo è scomodo e potenzialmente rischioso.

Ci interessa mantenere uno stato, uno *scope di sessione* che superi lo stato di request (o scope di request), che duri nel tempo e che sia *diverso per ogni utente*. Ci interessa cioè creare una *sessione*, un'area di memoria temporanea che possa mantenere attive delle variabili che coprano diverse request. Nella pratica, una Session è un oggetto contenente una mappa di attributi (Map < String, Object >) analoghi a quelli che inviamo alle JSP.

A differenza di quelli che usiamo come variabili di output, gli attributi di session non sono usa-e-getta, ma permangono nel tempo (per un certo tempo, dipendente dalle impostazioni del server e dalla volontà del programmatore) e sono accessibili dai vari doGet e doPost in sequenza.

Una prima request potrebbe *impostare* un attributo di sessione, mentre una successiva potrebbe trovarla già impostata e comportarsi di conseguenza. Vediamolo con un esempio concreto di uso tipico delle session: l'autenticazione presso un sistema di posta online.

Supponiamo di essere uno studente generation (studente@generation.com) e di voler inviare due email: una al vostro docente, (ilvostrodocente@generation.org) e una al vostro tutor (ilvostrotutor@generation.org), con testo "ciao".

Senza session, questa operazione richiederebbe di autenticarsi ogni volta. Dovremo inviare le seguenti request: /posta/mail?cmd=send&myemail=studente@generation.org&mypassword=miapassword&to=ilvostrodocente@generation.org&text=ciao e /posta/mail?cmd=send&myemail=studente@generation.org&mypassword=miapassword&to=ilvostrotutor@generation.org&text=ciao.

E' assurdo chiedere all'utente di inserire ogni volta la propria email e password (così come usare la query string per inviarle...), come è assurdo chiedergli di specificare ogni volta in che lingua preferisce vedere l'interfaccia, o in generale di ripetere le sue preferenze una volta espresse. Dobbiamo *ricordarci di lui*, almeno per un periodo.

Questo viene ottenuto con l'uso della servlet. Vediamo il seguente esempio:

```
// metodo doGet della servlet mappata a posta/mail

//Verifico di avere un utente. Se non c'è
//avrò null
User user = request.getSession().getAttribute("user");
// getSession restituisce un oggetto di tipo HttpSession
// che conterrà una mappa con chiave string e valore object
// per quanto getSession() appartenga a request, conviene
// immaginarlo come qualcosa di indipendente, che vive per più di una request

switch(cmd)
{
    case "formlogin":
    {
        request.getRequestDispatcher("formlogin.jsp").forward(request,response);
    }
    break;
    case "login":
    {
        User newuser = userdao.login(request.getParameter("email"), request.getParameter("password"));
        // il metodo login restituirà un oggetto user
        // corrispondente a quella email e password o null, se non ne trova di corrispondenti
        if(newuser!=null)
            request.getSession().setAttribute("user",newuser);
        break;
    }
    case "send":
    {
        if(user==null)
            request.getRequestDispatcher("forbidden.jsp").forward(Request,response);
        else
            sendMail(user, request.getParameter("to"), request.getParameter("text"));
    }
}
```

La servlet si comporta in maniera diversa a seconda che sia presente in session un oggetto User. Alla sua prima esecuzione sicuramente non ci sarà (request.getSession().getAttribute("user") sarà null), ma dopo il primo login riuscito (dopo il primo invio di email e password corrette) la session terrà in memoria la variabile user, per quella request e le successive.

Le chiamate per inviare email sarebbero quindi:

- /post/mail?cmd=formlogin, per caricare una form di login con caselle di testo per email e password, e il parametro nascosto cmd=login.
- /post/mail?cmd=login&email=xxx&password=yyy, arrivata dalla form precedente.
- /post/mail?cmd=send&to=zzz1&text=ttttt1 ,
/post/mail?cmd=send&to=zzz2&text=ttttt2 ,
/post/mail?cmd=send&to=zzz3&text=ttttt3 ,
/post/mail?cmd=send&to=zzz&4text=ttttt4

L'oggetto user è stato creato e messo nella session alla seconda request, e sarà presente nelle successive, senza bisogno di doverlo ricreare tramite invio di email e password.

21 - Web Service

21.1 Il concetto di Web Service - due definizioni

Il concetto di Web Service presenta due definizioni differenti, una "classica", accademica e formale, e una più pratica, industriale, che è quella con cui lavoraremo.

La definizione formale è quella del W3C (World Web Wide Consortium, una organizzazione che si occupa fra l'altro di definire gli standard per il web), che definisce un web service come un sistema software pensato per supportare l'interazione machine-to-machine e la cui interfaccia è specificata a sua volta in un formato interpretabile dalle macchine, per la precisione con un documento in formato WSDL, che non approfondiremo in questo momento, e usando la tecnologia SOAP (un altro standard per definito) per trasferire i dati.

Nella pratica, molti programmatori preferiscono evitare SOAP se appena possono e il documento WSDL spesso e volentieri non è presente. Le ragioni di queste scelte progettuali esulano dallo scopo di questo documento.

La definizione di web service che useremo per i nostri scopi è la seguente: un sistema software (programma), risiedente su un server e pensato per fornire servizi (nella pratica, *dati*) non direttamente a esseri umani ma ad altri programmi, attraverso il Web (quindi su HTTP) e in formato *machine ready*.

Non si tratta di una definizione inoppugnabile de jure, ma copre la quasi totalità di quello su cui andrete davvero a lavorare, e somiglia molto a una web-app. C'è un server, ci sono diversi client, e il server gestisce le request HTTP dei client. La differenza è che il client stasera non è, tipicamente, un browser, ma un altro programma, e non gli interessa ricevere pagine web ma dati grezzi.

In una web-app un browser potrebbe inviare la request GET /Index al server, e ricevere una pagina web graficata (HTML, CSS, Javascript) pronta a essere renderizzata e usata da un essere umano.

In un web-service, un client HTTP (che non sarà per forza e neanche tipicamente un browser) potrebbe inviare la request GET /shares al server e ricevere un file .CSV con le quotazioni delle azioni della Borsa di Milano in quel dato istante.

Il formato CSV non è un caso. I web service parlano machine-to-machine, e quindi forniscono dati che dovranno essere interpretati facilmente da altri programmi. CSV, XML, JSON, sono tutte scelte comuni. HTML molto meno.

A questo punto ci si potrebbe chiedere: perché far parlare due macchine? E' un caso estremamente comune in realtà. Potremmo avere una app sul cellulare che invia request http a dieci web service diversi, tutti implementati la stessa *interfaccia*, per trovare il prezzo migliore per un dato articolo. E' il caso degli aggregatori: è raro ormai scorrere i siti degli hotel uno per uno, mentre è normale avvalersi degli aggregatori. Un aggregatore sarà solo un client per n web service: interrogherà i web service dei vari alberghi e presenterà al cliente finale le varie possibilità.

Il cliente non interagisce col web-service, ma con l'app, magari sul suo telefonino. E' l'app che interagisce col web-service, quindi la comunicazione è machine to machine.

Un altro modo di immaginare un web service, non strettamente corretto in termini formali ma molto pratico, è la *condivisione* dei dati di un database (o se preferite dei suoi oggetti) sul web, utilizzando HTTP secondo una qualche interfaccia. Abbiamo il nostro DB e vogliamo che mille programmi diversi, scritti potenzialmente con tecnologie diverse (Java, C, C#, PHP, Objective C) ed eseguiti su piattaforme diverse (PC, cellulari, tablet, ma anche sensori intelligenti per la domotica) possano accedere secondo le regole che decidiamo.

Questo è il caso comune: gli alberghi condividono le stanze perché altri programmi possano prenotarle per gli utenti finali, i mercati azionari condividono gli asset che potrebbero essere acquistati da esseri umano o anche (e sempre più spesso) da bot. E così via.

Chi scrive un web service tipicamente non sa chi userà i servizi, né gli interessa. Scrivere un web service significa *implementare un'interfaccia*, una metodica di accesso al dato tramite chiamate HTTP. Chiunque rispetti quelle regole e sia connesso al web avrà gli accessi che abbiamo preventivato per lui.

Queste regole di accesso non sono necessariamente specificata in un WSDL (un documento formale di specifica, piuttosto complesso per quanto affascinante). Il formato in cui noi restituiremo i dati è tipicamente arbitrario. Da principio doveva essere XML, mentre ora sono comuni altri formati, primo fra i quali JSON. Il caso comune al momento della scrittura di questo libro in effetti non è una tecnologia standardizzata (WSDL e SOAP) ma uno *stile architettonico*, REST.

21.2 Introduzione a REST

REST (Representational state transfer), è uno stile architetturale estremamente popolare per la scrittura dei web service. Non si parla di una tecnologia vera e propria, quanto di un modo diverso di usare vecchie tecnologie, e in particolare HTTP.

REST è espresso tramite una serie di vincoli (constraints) che servono a definire se una architettura è REST o meno. Essendo uno stile più che una tecnologia, è aperto a interpretazioni entro certi limiti. Noi ci concentreremo sugli aspetti più comuni delle architetture REST, rimandando agli approfondimenti per quanto non trattato. Noi rispetteremo queste regole:

- **Struttura client-server:** è quella a cui siamo abituati. Abbiamo un server che gestisce molti client, che gli chiedono o inviano informazioni.
- **Senza-stato (statelessness):** è la natura di HTTP, e viene rispettata da REST. Le comunicazioni REST, che lavorano su HTTP, partono dall'idea che sia il client a dover inviare tutte le informazioni necessarie per l'esecuzione della request, senza che il server debba ricordare niente. In pratica, in REST non usiamo le Session o equivalenti. Ogni client si farà carico di memorizzare il proprio *stato*, e invierà ogni volta tutte le informazioni al server.

Questo permette la scalabilità del sistema. Un server faticherebbe a memorizzare milioni di session, e creerebbe un collo di bottiglia nell'architettura.

Se il client ha bisogno di autenticarsi o di ricordare qualcosa al server dovrà allegare tutto nella request.

- **Uniformità di interfaccia :** probabilmente il vincolo più importante. REST definisce uno stile univoco per rappresentare le risorse, uno stile univoco per definire la loro interfaccia al mondo, le loro API. In particolare, una risorsa è definita tramite il suo URL, il suo indirizzo, in maniera univoca. <http://www.generation.com/books/1> potrebbe essere l'indirizzo della risorsa che rappresenta questo testo: il sito è generation.com, la *collection* è books e la singola risorsa è books/1.

La prassi è distinguere fra risorse singole (che nella pratica terminano con un numero o un identificatore singolare) e collezioni di risorse (collection, tipicamente equivalenti a liste o set di oggetti), che terminano col nome della collezione, al plurale. Il numero è tipicamente l'id, ma un URI valido in REST potrebbe essere /books/javacompendium, per quanto gli id restino dominanti e diffusissimi.

E' anche tipico assegnare ai diversi *verbi* HTTP funzioni diverse per la manipolazione della risorsa. Mentre nelle web-app facciamo tutto con GET e POST indifferentemente, una API REST prevede di assegnare funzioni diverse a verbi diversi. Approfondiremo il concetto di interfaccia uniforme e la sua manipolazione nella prossima sezione.

Ci sono almeno altri due vincoli da rispettare: la cacheability (una response in REST può essere cachabile o non cachabile, e deve dichiararlo) e il vincolo di sistema a strati (layered system), che prevede che un sistema REST debba funzionare ugualmente con o senza intermediari (proxy), ma esulano dalla trattazione del corso e rimandiamo agli approfondimenti.

21.3 Manipolazione delle risorse e i verbi di HTTP

Il punto chiave di REST è la rappresentazione di un oggetto sul server tramite un URI. Il primo libro della collezione books del web service "courses" sul sito di generation.com potrebbe essere <http://www.generation.com/courses/books/1>, e sarà un indirizzo univoco e universale per identificare la risorsa, vale a dire un oggetto in un database sul server di generation.com, probabilmente.

I client potranno richiedere la risorsa, o più correttamente il *trasferimento di una rappresentazione del suo stato* (da cui l'acronimo). E' una maniera spaventosamente formale per dire "mandami le informazioni di questo oggetto".

La rappresentazione non è vincolata al formato con cui l'oggetto è memorizzato sul server. Potrebbe trattarsi di una riga su un database (il caso comune), di un oggetto salvato in un database NoSQL (MongoDB?), o perfino di un oggetto temporaneo in memoria (un oggetto Java senza persistenza). Il client ne riceverà una rappresentazione in un formato concordato col server ("content negotiation").

Tutto questo rischia di essere molto astratto, ma un esempio dovrebbe chiarire. Supponiamo di avere un web service REST di nome census sulla porta 8080 di localhost, che esponga una collection di nome people, e i cui singoli elementi siano disponibili tramite id. Ipotizziamo la seguente request:

```
GET /census/people/1 HTTP/1.1
Host: localhost:8080
Accept: application/json
Cache-Control: no-cache
Postman-Token: ff1f7a9d-6e11-b281-bff5-b3839907d091
```

Questa request, generata non dal browser ma da un tool molto utile per testare i web service, che prende il nome di Postman (<https://www.postman.com/downloads/>), è abbastanza esplicativa, e mostra HTTP sotto il cofano. E' una request GET a un URI, su un server specifico (localhost:8080), non cacheabile, e che specifica un header : Accept. Accept indica la disponibilità del client a "capire" quel formato. In questo caso, il client pretende di ricevere una risposta in formato json (un oggetto Javascript). Si tratta di uno di molti header predefiniti, noti ad HTTP, a cui possiamo accedere tramite la request.

Il corpo della response potrebbe essere il seguente:

```
{"name": "John", "surname": "Smith", "age": "41"}
```

Noteate che abbiamo parlato di "corpo" della response. La response non è formata solo dal suo corpo, ma anche da header e in particolare da uno *status code*. Sono tutti elementi importanti nelle web-app ma fondamentali nei web services, come vedremo a breve.

L'indirizzo (URL) della risorsa identifica l'oggetto, ma a questo punto in REST è convenzione mappare i quattro verbi fondamentali di HTTP (POST, GET, PUT, DELETE) alle quattro operazioni fondamentali di CRUD (Create, Read, Update e Delete rispettivamente). Questa è la prima grossa differenza rispetto alle web app, in cui i verbi sono indifferenti.

Partiamo con un esempio di POST (C in CRUD):

```
POST /census/people/1 HTTP/1.1
Host: localhost:8080
Accept: application/xml
Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Postman-Token: 55f1b043-349d-458a-6cc7-1eae349c159b

name=John&surname=Smith&age=41
```

Questa request post dovrebbe inviare tutti i dati per creare una nuova risorsa, e in effetti lo fa. Il server, ricevendo questi dati, dovrebbe creare la risorsa con id=1 *se non esiste ancora*, e fornirne una rappresentazione in formato XML. Se la risorsa con id=1 esiste già, il server dovrà restituire un codice HTTP di errore convenzionale, come vedremo nel codice.

Una maniera più comune di creare risorse (entità) è la seguente:

```
POST /census/people HTTP/1.1
Host: localhost:8080
Accept: application/xml
Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Postman-Token: 55f1b043-349d-458a-6cc7-1eae349c159b

name=John&surname=Smith&age=41
```

Il POST sulla collezione viene tipicamente inteso come "crea una risorsa in quella collezione", per quanto non sia strettamente corretto dal punto di vista formale. Noteate che in questo caso non stiamo specificando l'id tramite l'URL (l'URI, in realtà), e la rappresentazione XML che ci verrà restituita presenterà un collegamento al nuovo id. La response potrebbe avere questo aspetto:

```
<person>
```

```

<id>6</id>
<name>John</name>
<surname>Smith</surname>
<age>35</age>
</person>

```

Questo tipo di request POST è pensata per essere multipotente. Eseguendola ancora e ancora e ancora creeremo ogni volta una nuova persona, di cui cambierà solo l'ID.

Il verbo GET è utilizzato per leggere (R in CRUD), e lo abbiamo visto in precedenza. GET è nullo-potente, vale a dire non deve mai cambiare lo stato del database remoto. Invocare GET n volte sullo stesso dato produrrà la stessa risposta.

GET su una collezione produrrà un elenco di elementi invece di uno solo. GET /census/people potrebbe restituire la lista di tutte le persone nel nostro database in qualunque formato accettato dal client (header Accept), mentre GET /census/people/Milano potrebbe produrre una lista di milanesi.

E' possibile, e comune, specificare parametri di filtri per le operazioni di GET, sia tramite l'URI (come sopra, che è la scelta preferenziale) che tramite parametri (GET /census/people?city=Milano).

PUT viene utilizzato per aggiornare una risorsa già esistente (U in CRUD), ed è idempotente. Inviare una request PUT con gli stessi dati n volte sarà equivalente a inviare un'unica request PUT. Nel caso in cui la risorsa non dovesse esistere, questa non verrà creata ma verrà restituito un codice di errore. Di seguito un esempio di request PUT:

```

PUT /census/people/1 HTTP/1.1
Host: localhost:8080
Accept: application/xml
Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Postman-Token: 0679c75f-d22e-d31c-d2e8-ac047f0e07f4
name=John&surname=Smith&age=36

```

Teoricamente possiamo fare PUT di intere collezioni, ma è insolito. Si preferisce lavorare risorsa per risorsa.

Per terminare, il verbo DELETE (D in CRUD) fa esattamente ciò che il nome suggerisce: cancella una risorsa. E' estremamente semplice da illustrare:

```

DELETE /census/people/1 HTTP/1.1
Host: localhost:8080
Accept: application/xml
Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Postman-Token: 093c6432-3bf2-b35a-ecea-5d053bcb1297

```

Anche DELETE è idempotente: cancellare una risorsa mille volte è come cancellarla una sola, ma è buona prassi restituire un codice HTTP di errore se si prova a cancellare una risorsa già rimossa.

I codice di errore HTTP, gli header e la content negotiation verranno spiegati in dettaglio nella prossima sezione.

21.4 Un esempio di Web-Service REST per una risorsa semplice

Mettiamo assieme quanto visto fino ad ora. Ci troviamo nella situazione di voler condividere i dati relativi a delle persone via web, con attori ancora non identificati. Di conseguenza decidiamo di approntare un web-service.

Cerchiamo di venire incontro ai nostri utenti, producendo dati almeno in tre formati (CSV, XML, JSON), ed esponiamo una interfaccia uniforme secondo i vincoli REST. Per fornire questo servizio avremo bisogno di analizzare, nella request, le seguenti parti:

- **Lo URI (Uniform Resource Identifier)** .

Lo URI è l'ultima parte della URL, tutto ciò che viene dopo il nome del server e l'eventuale porta. Se l'URL è `http://localhost:8080/census/people/1`, la URI è `/census/people/1`. È accessibile tramite il metodo `String request.getRequestURI()`, e lo useremo per ricavare la risorsa su cui operare (o la collezione).

- **I parametri.**

Sono la consueta mappa `< String, String [] >`, tipicamente degenerata in `< String, String >`, che abbiamo imparato a conoscere nelle web-app. Ci serviranno per i verbi (o metodi) POST e PUT. Sono accessibili, fra le altre cose, tramite `String request.getParameter(String key)`

- **Gli header**

Questi sono una novità. Sono una mappa `< String, String >` di informazioni che vengono tipicamente decise dal programmatore, non dall'utente, e che servono a integrare la request con informazioni di configurazione. Ci interesserà in maniera particolare l'header con chiave `Accept`, che potrebbe avere valore `text/json`, `text/xml`, `application/csv` (tutti valori convenzionali, definiti in documenti industriali standard) per capire quale tipo di documento, di *rappresentazione* produrre. In base al valore di `Accept` decideremo quale *vista* usare per formattare il dato. Sono accessibili tramite `String request.getHeader(String key)`.

Allo stesso modo, dovremo fare di più che generare il *corpo* della response. Sarà necessario integrarlo con degli *header* (sempre `Map < String, String >`, ma questa volta inviati dal server al client per permettergli di capire meglio la response) e con uno *status code* .

In alcuni casi non avremo davvero un corpo della response, ma solo uno status code, vale a dire un numero convenzionale, il più famoso dei quali è 404. GET `/census/people/67` produrrà una response contenente solo uno status code:

```
response.setStatus(404);
```

404 è il codice convenzionale per "non trovato", ed è un codice della famiglia 400 che indica "errore del client", o comunque una request non esaudibile. Altri due codici molto utili della famiglia 400 sono il 400 stesso (errore generico del client, utile quando non abbiamo un codice specifico) e 403 ("Forbidden", vietato, come quando cerchiamo di eseguire operazioni per cui non abbiamo fornito gli accessi).

Di contro, quando le cose vanno bene, restituiamo il codice 200. 200 significa letteralmente "OK", quindi request andata a buon fine, e indica successo come tutti i codici della famiglia 200. È anche il codice predefinito che inviamo quando non specifichiamo nessun altro codice (c'è *sempre* un codice HTTP di stato, ed era 200 nelle nostre web app). Noi useremo 200 e 203 ("creazione avvenuta con successo") per i nostri programmi.

I codici della famiglia 500 indicano invece "errore del server", o errore di programmazione, e vengono forniti automaticamente da Tomcat quando il nostro programma crasha o si verificano altri eventi similmente edificanti. Possiamo anche impostarli noi, ma è insolito. Ci sarebbe da parlare anche della famiglia 300, ma non la useremo per i nostri scopi.

In termini di header di response, invece, useremo principalmente Content-Type, per indicare il tipo di documento prodotto:

```
response.setHeader("Content-Type", "text/json");
```

Ora vediamo in pratica una servlet che si occupi di implementare i quattro verbi fondamentali, di capire quali risorse le vengono chieste e di produrre le relative response (body, status code e header). Ometteremo i DAO per semplicità, mentre ci fermeremo sulle viste in seguito. Cominciamo con la struttura di base:

```
package com.generation.censusws.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

//questa servlet non è mappata a un solo indirizzo, ma a una famiglia di indirizzi
//è mappata a /people/ seguita da qualunque altra cosa
@WebServlet("/people/*")
public class Index extends HttpServlet
{
    private static final long serialVersionUID = 1L;
    PersonDAO dao = PersonDAOFactory.make();
    public Index()
    {
        super();
    }
    // lettura
```

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
}
//creazione
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
}
//aggiornamento
protected void doPut(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
}
//cancellazione
protected void doDelete(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
}
}

```

Per ora ignoriamo la content negotiation e restituiamo come "rappresentazione" il `toString` degli oggetti. Ignoriamo anche il controllo degli errori e tutto quanto non sia funzionamento di base. Cominciamo con la lettura.

Il `doGet` potrebbe ricevere chiamate a `/people/1` (singola risorsa) o a `/people` ("voglio vederli tutti"). Dobbiamo capire se stiamo lavorando su un singolo o una collezione. Poi dobbiamo capire se quel singolo esiste. Per finire dobbiamo generare una response. Sempre per semplicità, supponiamo che tutti gli accessi (lettura, scrittura, cancellazione) siano non autenticati.

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    String URI = request.getRequestURI();
    String parts[] = URI.split("/");
    //l'ultima parte può essere people oppure numerica.
    String last = parts[parts.length-1];
    if(last.equals("people"))
    {
        //collezione:
        response.getWriter().append(persondao.list().toString());
    }
    else
    {
        //singolo
        response.getWriter().append(persondao.load(Integer.parseInt(last)).toString());
    }
}

```

E' una versione idealizzata. E se qualcuno eseguisse GET `/census/people/pippo?` `Integer.parseInt` darebbe errore. Inoltre, non stiamo controllando che la risorsa esista. Stiamo stampando i `toString` e non in formato machine-ready (standard industriali: CSV, JSON, XML...), non stiamo fornendo codici di errore e in generale non stiamo gestendo gli errori.

Rimediamo:

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //prima di tutto il resto: "parlo la lingua del client"?
    String accept = request.getHeader("Accept");
    PersonView view = PersonViewFactory.make(accept);
    if(view==null)
    {
        //mi dispiace, non so eseguire la tua request
        //non supporto quel formato
        response.setStatus(415);
        //415: UNSUPPORTED MEDIA TYPE. USATO IN MANIERA IMPROPRIA QUI
        //ma utile.
        return;
        //non proseguo. Se non ho una vista mi rifiuto di andare avanti.
        //la vista qui serve per formattare i dati
    }
    //la vista sa dirmi in che formato lavora. Prima di qualunque altra cosa
    //imposto l'header Content-Type della response. Dirà al server che formato stiamo producendo
    response.setHeader("Content-Type", view.getContentType());
    String URI = request.getRequestURI();
    String parts[] = URI.split("/");
    //l'ultima parte può essere people oppure numerica.
    String last = parts[parts.length-1];
    if(last.equals("people"))
    {
        response.getWriter().append(view.render(persondao.list()));
    }
    else
    {
        try

```

```

    {
        int id = Integer.parseInt(last);
        Person p = persondao.load(id);
        if(p==null)
        {
            //non c'è la persona richiesta. Errore NOT FOUND
            response.setStatus(404);
            return;
        }
        //ok, ci siamo
        response.getWriter().append(view.render(p));
    }
    catch(NumberFormatException e)
    {
        //non era un id: errore del client
        response.setStatus(400);
    }
}
}
}

```

Decisamente meno banale. Questo metodo potrebbe generare: una lista di oggetti in un qualche formato specificato da Accept con codice di stato 200, un singolo oggetto con codice di stato 200, una response vuota con codice di stato 415 (vedere sopra) se non riconosciamo il formato richiesto, una response vuota con codice 404 se è stato richiesto un id non presente in database (se persondao.load() ci ha restituito null), una response vuota con codice 400 se è stato specificato un id non numerico dopo /people.

In tutto questo la view (interfaccia PersonView) svolge un ruolo da leone, ed è anche una dimostrazione da manuale del pattern Factory, ma non è immediatamente chiaro se non diamo almeno una implementazione di base di tre "pezzi" del motore: PersonView (interfaccia), PersonViewJSON (classe) e PersonViewFactory (classe):

```

public interface PersonView
{
    //come da regole MVC, la View si occupa di graficare il dato, di creare l'interfaccia grafica
    //una view in un web service si occupa di stampare il dato ma anche di impostare il Content-Type, che prende anche il nome di MIME-TYPE
    //deve quindi avere tre metodi: render di un singolo, render di una lista e getContentType
    String render(Person person);
    String render(List <Person> people);
    //restituirà un MIME TYPE
    String getContentType();
}

// quello che abbiamo visto sopra era il contratto, vale a dire la responsabilità di una vista. Ora vediamo un caso concreto di vista per un web
service, che tradurrà le persone in JSON

public class PersonViewJSON implements PersonView
{
    public String getContentType()
    {
        //fisso. Sempre uguale.
        return "text/json";
    }
    public String render(Person p)
    {
        return "{\"name\":\""+p.getName()+"\",\"surname\":\""+p.getSurname()+"\",\"age\":\""+p.getAge()+"}";
    }
    public String render(List <Person> people)
    {
        String res = "[";
        for(int i=0; i<people.size(); i++)
        {
            res+=render(people.get(i));
            if(i < people.size()-1) res+=",";
        }
        res+="]";
        return res;
    }
}
//Il codice di render ( List ) è lasciato come esercizio di deduzione. Basi sapere che creerà qualcosa del tipo [{};{};{};...]

//E ora, la factory. Per ora può produrre solo una PersonViewJSON o null. Il nostro web service per ora gestisce solo request in JSON
public abstract class PersonViewFactory
{
    private static PersonView json = new PersonViewJSON();
    public static PersonView make(String mimetype)
    {
        return mimetype.equals("text/json") ? json : null;
    }
}

```

```
// come modificheremo la factory per introdurre XML, CSV ecc...?
```

Ora dovrebbe essere più chiaro. Il client invia la sua request, completa di Accept. PersonViewFactory vede se abbiamo un pezzo adatto a soddisfare la request (a graficare le persone nel formato richiesto). Se non lo abbiamo, restituisce null, e la servlet restituirà un codice http di errore. Altrimenti potremo provare a graficare qualcosa, se c'è qualcosa da graficare. La view si occupa anche di impostare il content-type di risposta (header di response, metodo view.getContentType).

Partendo da questi elementi, potremmo passare a POST. Scegiamo di creare una nuova risorsa facendo POST sulla collezione, quindi senza passare un id ma ottenendolo automaticamente. POST /people andrà bene, POST /people/1 no.

```
//creazione
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    String accept = request.getHeader("Accept");
    PersonView view = PersonViewFactory.make(accept);
    if(view==null)
    {
        response.setStatus(415);
        return;
    }
    response.setHeader("Content-Type", view.getContentType());
    String URI = request.getRequestURI();
    String parts[] = URI.split("/");
    //l'ultima parte può essere people oppure numerica.
    String last = parts[parts.length-1];
    if(last.equals("people"))
    {
        try
        {
            //provo a ricreare la persona dai parametri passati
            //se ci sono eccezioni o i dati non sono validi finisco nel blocco catch
            Person p = new Person
            (
                request.getParameter("name"), request.getParameter("surname"),
                Integer.parseInt(request.getParameter("age"))
            );
            if(!p.isValid())
                throw new Exception("BAD DATA");
            p = persondao.save(p);
            //se è andato tutto bene lo stampo
            response.getWriter().append(view.render(p));
        }
        catch(Exception e)
        {
            //houston: dati errati
            response.setStatus(400);
        }
    }
    else
        response.setStatus(403);
    //accetto solo request su /people. altrimenti, FORBIDDEN
}
```

PUT non è molto diverso, ma occorre prima caricare la persona, e verificare che esista. Nel caso di PUT vogliamo la mappatura /people/id.

```
//aggiornamento
protected void doPut(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    String accept = request.getHeader("Accept");
    PersonView view = PersonViewFactory.make(accept);
    if(view==null)
    {
        response.setStatus(415);
        return;
    }
    response.setHeader("Content-Type", view.getContentType());
    String URI = request.getRequestURI();
    String parts[] = URI.split("/");
    //l'ultima parte può essere people oppure numerica.
    String last = parts[parts.length-1];
    if(!last.equals("people"))
    {
        try
        {
```

```

Person p = persondao.load(Integer.parseInt(request.getParameter("id")));
if(p==null)
{
    //non la ho trovata. non posso aggiornarla.
    response.setStatus(404);
    return;
}
p.setName(request.getParameter("name"));
p.setSurname(request.getParameter("surname"));
p.setAge(Integer.parseInt(request.getParameter("age")));
if(!p.isValid())
    throw new Exception("BAD DATA");
//sovrascrivo la persona
p = persondao.save(p);
//la restituisco salvata
response.getWriter().append(view.render(p));
}
catch(Exception e)
{
    //houston: dati errati
    response.setStatus(400);
}
}
else
response.setStatus(403);
//accetto solo request PUT su /people/id. altrimenti, FORBIDDEN
}

```

Per terminare, DELETE è piuttosto simile. Lavoro sulla singola risorsa, restituisco 404 nel caso in cui non ci sia, altrimenti cancello. In questo caso, non ci serve neanche la vista.

```

//cancellazione
protected void doDelete(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    //non mi serve neanche la vista. non ci sarà corpo di risposta
    //basta lo status code
    String URI = request.getRequestURI();
    String parts[] = URI.split("/");
    //l'ultima parte può essere people oppure numerica.
    String last = parts[parts.length-1];
    if(!last.equals("people"))
    {
        try
        {
            Person p = persondao.load(Integer.parseInt(request.getParameter("id")));
            if(p==null)
            {
                //non la ho trovata. non posso aggiornarla.
                response.setStatus(404);
                return;
            }
            persondao.delete(p);
        }
        catch(Exception e)
        {
            //houston: è successo qualcosa di sbagliato
            response.setStatus(400);
        }
    }
    else
        response.setStatus(403);
    //accetto srequest DELETE su /people/id. altrimenti, FORBIDDEN
}

```

21.5 Approfondimenti

Abbiamo sorvolato su diversi aspetti di REST che sono meno centrali nella pratica ma comunque degni di attenzione. Il principale è HATEOAS (Hypermedia As The Engine Of The Application State), vale a dire l'idea di utilizzare collegamenti all'interno delle response per manipolare le risorse esposte.

Bisogna anche notare che in alcune trattazioni PUT viene usato per creare quanto per modificare, e POST è un verbo di uso generale per tutte le funzioni che non sono strettamente CRUD, ma preferiamo mostrare una versione "basilare" di REST, che coprirà la maggior parte delle applicazioni.

Per approfondire, raccomandiamo la documentazione Oracle sui web-service RESTFUL: <https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html>

22 - Spring

22.1 Introduzione a Spring

Spring è un framework di Java, uno dei più diffusi e più utilizzati nella pratica. E' modulare, composto di vari strumenti che si innestano su di un core. Il core implementa diversi pattern a cui siamo abituati in maniera trasparente e velocizza la scrittura delle applicazioni mantenendo una struttura elegante e coesa, facile da manutenere.

Per i nostri scopi ci concentreremo su Spring MVC, l'estensione di Spring per web application e i web services, che comunque prevede l'installazione e l'uso del core.

In termini di architettura, Spring MVC prenderà il posto delle servlet nella scrittura delle web app, e resterà indifferente al livello sottostante. Potremo usare Spring MVC con le sue librerie di persistenza specifiche, con JDBC liscio o con Eclipse Link, indifferentemente. Astrarremo da queste problematiche utilizzando le consuete interfacce (DAO, BL), e ci concentreremo su quella che invece è la struttura di Spring e di Spring MVC.

22.2 Installazione e configurazione con Maven

Spring va scaricato e collegato al path dell'applicazione non differentemente da quanto accade per EclipseLink. Una soluzione alternativa al download delle librerie è fornita dall'utilizzo di Maven.

Maven è un tool multiuso, difficile anche solo da definire. Maven è un termine ebraico che significa "accumulare conoscenza" e può essere considerato un gestore di progetti, in grado di automatizzare tutta una serie di compiti comuni nella gestione di un progetto Java. Un progetto Java con Maven attivo in effetti viene detto "mavenizzato", e Maven diventa una sorta di coordinatore per il progetto.

Noi seguiremo i seguenti step in Eclipse per ottenere un web application mavenizzata:

1. Creeremo un dynamic web project nella solita maniera.
2. Cliccheremo col pulsante destro sul progetto e sceglieremo Configure -> Convert to Maven Project.
A questo punto dovremo specificare un group name (il nome del nostro team di sviluppo) e un artifact name. Artifact è un termine di comodo per indicare "programma" o "libreria", ed è qualcosa che Maven sa mettere in comune fra progetti diversi, modularizzando il software e semplificandone la composizione.
Sceglieremo com.generation come nome del gruppo (tipicamente il sito della nostra azienda in forma infissa - estensione.dominio) e diamo un nome univoco, all'interno della nostra azienda, al progetto. Un esempio potrebbe essere SpringCensus.

Terminati questi passi noteremo che il progetto ha un aspetto diverso, e che c'è un nuovo file di configurazione: pom.xml. pom.xml è il file di configurazione del progetto per maven, e assolve a diverse funzioni. Vediamone un esempio:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
>
<modelVersion>4.0.0</modelVersion>
<groupId>com.generation</groupId>
<artifactId>SpringCensus</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <plugin>
            <artifactId>maven-war-plugin</artifactId>
            <version>3.2.1</version>
            <configuration>
                <warSourceDirectory>WebContent</warSourceDirectory>
            </configuration>
        </plugin>
    </plugins>
</build>
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.3.22</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.22</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>5.3.22</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.22</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.3.22</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/jsp-api -->
    <!-- ne parleremo... -->
    <dependency>
        <groupId>javax.servlet</groupId>
```

```

        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.6</version>
    </dependency>
</dependencies>
</project>

```

Essendo un file .xml deve avere una radice, e questa è project. Dentro project troveremo una serie di tag di interesse: groupId (il nome o identificativo del gruppo), artifactid (il nome o identificativo del progetto, o meglio dell'artifact) e un tag complesso dependencies, contenente a sua volta diversi tag dependency.

Il tag dependencies è il nostro principale interesse in Maven. E' possibile specificare una serie di artifact che *vogliamo importare automaticamente nel nostro progetto*. Maven andrà a cercare quei gruppi e quegli artefatti in dei repository online (qualcosa di simile a uno "store" online di applicazioni), li scaricherà e li collegherà automaticamente al nostro path. E in effetti, leggendo i tag dependency troviamo qualcosa di piuttosto rivelatorio:

```

...
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.22</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.3.22</version>
</dependency>
...

```

Maven ha cercato sui repository il gruppo org.springframework (i principali "manutentori" di Spring, gli autori, l'ente responsabile) e ha scaricato e installato i moduli di Spring di nostro interesse (non solo questi due, ce ne sono diversi). Notiamo anche che abbiamo potuto specificare la versione (5.1.3 RELEASE): Maven ci permette di scegliere fra varie versioni di un artifact, ed è parte della ragione per cui abbiamo dovuto versionarlo (oltre a dare group id e artifact id) in fase di mavenizzazione.

Per i nostri scopi sarà tipicamente sufficiente copiare questi tag dependency, integrandole eventualmente con quelle di EclipseLink (se lo desideriamo) o semplicemente col connettore MySQL. Potenzialmente potremmo anche convertire il progetto in progetto JPA (in realtà a questo punto si parlerebbe di progetto *faceted*) e lasciare che sia il wizard di Eclipse a cercare per noi le librerie.

A scanso di equivoci, queste dependency sono *dipendenze del progetto*, sono *classi e interfacce*, sono *librerie*. Non bisogna confonderle con le dipendenze di *oggetto* che vedremo in seguito, e che saranno alla base del meccanismo di *dependency injection di Spring*.

Per ora, ricordiamo questo: in Maven una dependency è una libreria, un insieme di tipi, del codice. In Spring con dependency si intende un oggetto. Questo è lo stato delle cose, e dobbiamo convivere con queste ambiguità terminologiche.

A ogni modo, na volta scaricato e collegato tramite Maven, Spring sarà disponibile per l'uso.

Ora bisogna passare a configurare Spring, e per essere precisi Spring MVC.

I dettagli cambiano a seconda della versione che andremo a usare: il nostro riferimento è la versione 5.1.3 che analogamente a EclipseLink avrà un file di configurazione centrale in formato XML e necessiterà poi di annotazioni nelle *classi gestite*.

Il primo passaggio di configurazione avviene fuori da Spring, al livello del Dynamic Web Project. Dobbiamo intervenire sul file di configurazione del DWP per dirgli che vogliamo delegare una parte, o tutto, il lavoro a Spring MVC.

Così come pom.xml è il file di configurazione di Maven e persistence.xml è il file di configurazione di EclipseLink, il DWP viene configurato nel file web.xml, che viene creato contestualmente al progetto.

Il file contiene una serie di informazioni sulla struttura del progetto web, e in particolare permette di definire delle servlet e di mapparle a uno o più indirizzi. Vediamo un esempio di un file web.xml che delega parte del lavoro a Spring:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0"
>
    <display-name>SpringCensus</display-name>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <!--
        sto creando una servlet. Il progetto, appena avviato
        creerà una servlet della classe org.springframework.web.servlet.DispatcherServlet
        abbiamo importato questa classe con Maven in precedenza.
        E' la classe che in Spring MVC ha la funzione di Front Controller e
        prende il nome di dispatcher servlet.
        L'oggetto si chiamerà "spring", ma il nome è arbitrario.
        L'oggetto spring delle classi data verrà mappato (tag servlet mapping) alla radice del DWP.
        Vale a dire che tutte le chiamate al DWP verranno dirottate alla Dispatcher Servlet!
    -->

```

```
-->
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

Ribadiamo che non è un file di configurazione di spring, ma del dynamic web project. Ne definisce il nome (SpringCensus), la pagina principale da caricare nel caso in cui non venga specificata una (index.jsp) e poi arriva alla parte che ci interessa.

Questo file XML dispone la creazione di una servlet di nome spring, che sarà configurata in un file .xml di nome spring-servlet.xml (per convenzione: se l'oggetto servlet si chiama a, il file di configurazione si chiama a-servlet.xml), posto nella stessa cartella di web.xml.

La servlet appena creata è il "cuore" di Spring MVC, e viene detta dispatcher servlet. Svolge le funzioni del front controller visto in precedenza: ridireziona le request verso controller di secondo livello, che vedremo a breve. Può risultare controintuitivo, ma questo file XML comporterà la creazione di un oggetto Java, che verrà poi configurato in un altro file XML. E' tutto implicito, e bisogna semplicemente esserne al corrente.

Andiamo quindi a esaminare il file spring-servlet.xml, che definirà il comportamento della dispatcher servlet (front controller) e in sostanza "configurerà Spring":

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- questa sarà la dispatcher servlet -->
    <!-- useremo le annotation per far funzionare tutto -->
    <mvc:annotation-driven />

    <!-- package che conterrà i controller. Attenzione a non sbagliare!
    Spring cercherà le classi di suo "interesse" in questo package.
    -->
    <context:component-scan
        base-package="com.generation.springcensus.controller" />

    <mvc:default-servlet-handler />

    <!--
        qui sto definendo un "bean", vale a dire un
        componente riusabili e condivisibile in Spring.
        e non è neanche un bean qualunque. E' il view
        resolver, il componente che si occuperà di decidere la logica di presentazione. Ci dirà
        dove troviamo le viste, le JSP (in /WEB-INF/jsp)
        e quale estensione avranno (.jsp, appunto).
        Anche in questo caso, sto usando XML per creare
        un oggetto. Dentro il file di configurazione di
        spring-servlet.xml, vale a dire nella crezione della dispatcher servlet, stiamo anche creando
        un oggetto di tipo ViewResolver
        (org.springframework.web.servlet.view.InternalResourceViewResolver, per la precisione).
    -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
    <!-- questo è il file di configurazione di spring mvc -->
</beans>
```

E' un altro file piuttosto facile da copiare e incollare. Di volta in volta cambierà solo il valore di context:component-scan, che dovrà puntare a un package che per convenzione indicheremo con nomedelprogetto.controller.

Quello sarà il "punto di partenza" per Spring, che scansionerà le classi di quel package alla ricerca di "bean", di oggetti da gestire e, in questo caso, da mappare verso indirizzi web.

Terminiamo con un esempio concreto: il file Test.java, in com.generation.springcensus.controller

```
package com.generation.springcensus.controller
```

```
@Controller
```

```
public class TestController
```

```

{
    @RequestMapping("/test")
    public String test()
    {
        return "test";
    }
}

```

Questa classe definisce un controller di secondo livello, attivato dalla dispatcher servlet, e lo fa tramite l'uso di due annotation. La prima è @Controller. Spring analizza i file di questo package e cerca l'annotation controller, analogamente a quanto viene fatto da EclipseLink nell'analizzare le classi cercando @Entity.

Una volta trovata l'annotation @Controller, Spring sa che quella classe sosterrà di essere un controller, e che alcuni dei suoi *metodi* saranno *mappati*.

Questa è una prima grossa differenza con le servlet. Le servlet mappano *per classe*, Spring mappa per metodo. In particolare, abbiamo mappato il metodo test() all'indirizzo /test tramite l'annotation @RequestMapping, ed è qui che le cose si fanno eleganti ma sottili.

Il metodo test() viene eseguito quando l'utente accederà all'indirizzo /test. Questo è abbastanza chiaro vista l'annotation RequestMapping posta appena sopra (ricorda la mappatura della servlet nei DWP classici), ma il lettore attento si starà chiedendo: perché String? Non dovremmo produrre una response?

Siamo nell'ambito di Spring MVC, non di Spring Core. Possiamo dire senza sbagliarci che stiamo lavorando per il web, quindi per HTTP, quindi Request e Response. E in effetti quel metodo produce una response. Vediamo come:

Prima di tutto, è un metodo mappato a un indirizzo dentro una classe marcata come @Controller. Quindi sa di dover produrre una response. Successivamente, SpringMVC è, appunto, un framework MVC. Questo è il controller, ma per stampare i dati useremo le viste. Le viste, come abbiamo detto prima, sono definite dal ViewResolver. La String di ritorno *non* è il ritorno, ma è il nome della vista. Dire return "test" è come dire "renderizza la vista (programma) test.jsp in WEB_INF/jsp".

Il valore di ritorno non è la response. E' *il nome della vista che produrrà la response*, a cui viene aggiunto un suffisso definito in spring-servlet.xml (.jsp), e che verrà cercata nella cartella definita sempre in spring-servlet.xml.

return "test" si tradurrebbe, nelle servlet, in request.getRequestDispatcher("WEB-INF/jsp/test.jsp").forward(request,response);. SpringMVC è estremamente più conciso, sfruttando la configurazione fatta a monte.

test.jsp potrebbe contenere semplicemente "Ciao mondo", e sarà quello che verrà inviato al client.

Adesso andiamo a simulare due passaggi:

1. l'avvio del progetto web con questa configurazione
2. l'esecuzione di GET /springcensus/test a progetto attivo

22.3 Step nell'avvio di un progetto web SPRINGMVC

Tenendo da conto quanto visto alla sezione precedente, premendo play sul progetto web vengono eseguiti i seguenti passaggi:

1. Tomcat parte e attiva il progetto. Il primo file che viene "eseguito", o meglio, che determina una esecuzione, è web.xml.
2. web.xml come abbiamo visto determina l'interpretazione di spring-servlet.xml. Il file spring-servlet.xml determina la creazione della dispatcher-servlet (il nostro front controller), che in questo caso viene mappato a /. Questo vuol dire che *tutte* le chiamate a questo progetto (tutto /springcensus/*) passeranno per la dispatcher (servlet di smistamento).
3. durante l'interpretazione ("esecuzione", con molte virgolette) di spring-servlet.xml, Spring cerca il package com.generation.springcensus.controller, e cerca "quello che è suo", i suoi Bean, i suoi "oggetti". Il concetto di Bean in Spring è differente rispetto al concetto di Javabean visto in precedenza, e lo approfondiremo fra poco. Per ora, diciamo che trova una classe annotata a controller, e la analizza.
Inoltre, la dispatcher crea una ViewResolver, un gestore delle viste, che useremo in seguito.
4. La classe Test.java è annotata a controller. Spring la registra come bean di tipo Controller, e la scansione per trovare le sue *mappature*. Registra quindi la mappatura /test e la fa corrispondere all'esecuzione del metodo test(). I nomi non devono per forza coincidere.

A fine avvio, il sistema è attivo, il DWP mappa tutte le chiamate alla dispatcher servlet, e la dispatcher servlet mantiene in memoria gli indirizzi mappati e i metodi corrispondenti.

22.4 Esempio di una request su un DWP SpringMVC

Terminato di configurare, chiediamoci cosa succede quando invio GET /springcensus/test al mio progetto. Gli step sono i seguenti:

1. Il client invia GET /springcensus/test a Tomcat
2. Tomcat smista la request al progetto springcensus, che interpreta il proprio web.xml. La request viene passata a spring-servlet.xml
3. spring-servlet.xml riconosce in /test una mappatura gestita. /test viene girato al metodo test(), che viene seguito.
4. test() è di una sola riga: return "test". L'esecuzione passa alla vista (test.jsp) che produrrà la response. La response verrà a questo punto inviata al client.

Dovrebbe essere familiare a tutti: è il meccanismo che abbiamo usato nelle servlet fino a ora, solo estremamente più sintetico. E' possibile fare tutto quello che facevo nelle servlet: passare attributi da graficare, reindirizzare da una pagina all'altra, utilizzare sistemi di rendering avanzato delle viste, ma lo vedremo in seguito.

Per ora ci siamo concentrati sull'estensione web di Spring, che è di gran lunga la più popolare e forse la più importante, ma è necessario capire il core sottostante, e il suo meccanismo fondamentale: la *dependency injection*.

22.5 Dependency e Dependency Injection

Parlando di Maven, abbiamo definito dependency una libreria (un JAR, un insieme di classi) che Maven ha scaricato e collegato al path per noi. In letteratura però c'è un altro significato per il termine "dependency", e lo abbiamo visto più o meno in tutte le salse: è il rapporto di uso in UML. Se un oggetto di una classe A usa uno o più oggetti di una classe B, A dipende da B, in senso lato.

Nel caso più comune, l'oggetto di classe A contiene nel suo stato uno o più oggetti di classe B (A ha una o più proprietà di tipo B).

Prendiamo l'esempio dei web service visti in precedenza. La servlet (oggetto di classe Index) era il controller, e per operare sul database aveva bisogno di un oggetto di tipo PersonDAO. In questo caso, diremo che PersonDAO è una dipendenza (dependency) della servlet. Dependency, in questo caso, è un nome altisonante per indicare "un oggetto di cui ho bisogno".

E ribadiamo che si parla di *oggetti*, non di classi.

Posto che Index deve avere (conoscere, in realtà) un PersonDAO per funzionare, in che modo se lo procurerà?

La prima soluzione, la più ovvia, è PersonDAO dao = new PersonDAOMySQL(...) nel costruttore della servlet. E' la soluzione delle "dipendenze fatte in casa". Quello che mi serve lo creo.

E' perdente per tutta una serie di ragioni. La prima è che vincola un tipo concreto (PersonDAOMySQL) a una interfaccia (PersonDAO), uccidendo completamente il polimorfismo di oggetto. PersonDAOMySQL può andare bene oggi, ma non andrà bene domani, e non vogliamo fare il giro di tutto il codice per cambiarlo.

Inoltre, ogni servlet o comunque ogni utente di PersonDAO si creerà il proprio PersonDAO. Spreco di memoria, ripetizione delle connessioni, in generale pessima prassi.

Una seconda possibilità, più sensata, è utilizzare una factory. PersonDAO dao = PersonDAOFactory.make();, usando un metodo statico. In questo modo garantiamo il polimorfismo (l'essenza delle factory) e possiamo anche fornire più volte lo stesso oggetto (la factory potrebbe sempre riutilizzare un PersonDAO già prodotto). E' l'equivalente ad entrare in un supermercato e chiedere un prodotto adatto alle nostre esigenze. C'è la "seccatura" di dover andare al supermercato, di dover conoscere la factory, ma è già una soluzione accettabile.

Spring tuttavia ci permette di fare di meglio: la consegna a domicilio. L'idea è che ogni classe (in realtà, ogni oggetto di una data classe) definisca le proprie dipendenze, cioè gli oggetti di cui ha bisogno. A questo punto Spring procederà a fornirglieli "per magia".

Questo meccanismo viene detto autowiring, e garantisce tutti i vantaggi delle factory, più alcuni aggiuntivi, e in effetti è il cuore di Spring: Spring (non solo Spring MVC) ha come cuore una factory per la dependency injection, vale a dire una fabbrica di oggetti che verranno passati (*iniettati*) in altri oggetti.

I dettagli implementativi sono vari. E' possibile ottenere questo meccanismo tramite diverse forme di iniezione, ma noi utilizzeremo le annotation, forse la forma più comune.

Partiamo dalla *dichiarazione di una dipendenza*. Questo avviene tipicamente sopra una proprietà di oggetto (come potrebbe essere un oggetto di tipo PersonDAO), con la notazione "@Autowired". Ne approfittiamo anche per dare un'idea di come passare delle informazioni alle viste, tramite il seguente esempio:

```
package com.generation.springcensus.controller

@Controller
public class CensusWebController
{
    @Autowired
    PersonDAO persondao;
    public void setPersonDAO(PersonDAO personDAO)
    {
        this.personDAO = personDAO;
    }
    @RequestMapping("/people");
    public String list(Model model)
    {
        model.addAttribute("list", persondao.list());
        return "list";
    }
}
```

C'è molto di implicito da dire. Cominciamo con l'argomento specifico: @Autowired. Posto sopra PersonDAO persondao, è una comunicazione per Spring: "ho bisogno di un oggetto di tipo PersonDAO". Quando parliamo di tipo potremmo parlare di una interfaccia o di una classe, non è importante.

Nella forma più basilare, e più comune, di dependency injection, Spring cercherà altrove nel progetto, e presto vedremo dove, un oggetto che *fornisca quel servizio*, che implementi quel contratto. Trovato un PersonDAO, lo collegare alla proprietà personDAO della classe CensusWebController.

L'oggetto di tipo PersonDAO collegato (un Bean, per dirla nella maniera corretta in Spring) potrebbe essere nuovo o "usato", e in effetti tipicamente è così. Nel caso più comune un PersonDAO viene creato solo al primo richiamo: diremo che lo *scope di default* dei Bean è singleton. Questo può essere cambiato in seguito.

Ora sorge la domanda: da dove prenderà questo oggetto? Come lo creerà per noi? Ci sono diverse risposte e possibilità, ma noi utilizzeremo una tecnica abbastanza basilare: un *Context* annotato.

Il concetto di Context è affine a quello di una scatola degli attrezzi da cui pescare, contenente oggetti. Gli oggetti, che in Spring chiameremo Bean e che non corrispondono strettamente ai Javabean, potranno essere richiesti da chiunque nel progetto, da qualunque soggetto, tramite la notazione @Autowired.

Possiamo avere diversi Context, diverse "cassette degli attrezzi", e non è detto che una classe sia solo quello. Di seguito riportiamo un caso base:

```
package com.generation.springcensus.controller

@Component
public class Context
{
    @Bean
    public PersonDAO getDAO()
    {
        //Ipotizziamo di avere un PersonDAOMySQL che crei la propria connection... cattiva idea, ma per chiarezza...
        return new PersonDAOMySQL("localhost", "root", "root", "census");
    }
}
```

L'annotazione Component (<https://www.baeldung.com/spring-component-annotation>) dichiara che questa classe sarà un "context", sarà un contenitore di oggetti da iniettare (fornire) a terzi che ne facciano richiesta (tramite @Autowired).

Nel corpo di questa classe ho definito, indirettamente, tramite un metodo, un @Bean, un oggetto condivisibile, di tipo PersonDAO. Il risultato di questo metodo verrà salvato nella "credenza" di Spring, pronto a essere riutilizzato in seguito.

Notiamo che sono necessarie entrambe le annotazioni (component e bean) perché il sistema funziona. Assieme, costituiscono l'offerta di oggetti, mentre autowired è la domanda.

Cosa succede nel momento in cui avviamo il progetto? In aggiunta a quanto visto prima (tomcat, dwp, spring) viene eseguita una fase di assembly e registrazione. Con "assembly" si intende la fase in cui a ogni autowired viene fatto corrispondere un bean. E' la fase in cui le dipendenze vengono soddisfatte: all'avvio del progetto il controller verrà creato, e gli verrà passato un riferimento all'oggetto di tipo PersonDAO gestito da Spring. Per la precisione, Spring cercherà di eseguire una setter injection, chiamando setPersonDAO e fornendogli il riferimento all'oggetto, collegando domanda e offerta.

Il controller non conosce il suo tipo concreto, nè gli interessa (come nel caso delle factory), e non ha neanche bisogno di sapere da dove venga. Se una qualunque dipendenza non viene soddisfatta (salvo quelle marcate come opzionali, come è possibile fare tramite annotazione), Spring restituirà una eccezione e il progetto non potrà partire. Questo elimina alla radice una buona parte delle odiose NullPointerException che sono la croce del programmatore Java disattento dai tempi di Java 1.

Questo illustra una versione estremamente basilare, ma funzionale, del meccanismo della dependency injection. Il bean PersonDAO è singleton di default: verrà creato una sola volta. Successive richieste di PersonDAO forniranno lo stesso bean, e quando cambieremo il metodo getPersonDAO in Context cambieremo l'oggetto prodotto per tutto il sistema. Ora possiamo affrontare l'elefante nella stanza: cosa è quel Model? Cosa vuol dire addAttribute? Non è difficile da intuire, ma in questo ambito usciamo da Spring core e torniamo a Spring MVC.

22.6 Passaggio di attributi alle viste: Model in Spring MVC

Spring MVC rispetta il meccanismo di passaggio di attributi visto per le servlet, ma a questo punto non ci sorprenderà scoprire che è meccanizzato e reso implicito. Rivediamo l'esempio precedente, in dettaglio:

```
@RequestMapping("/people");
public String list(Model model)
{
    model.addAttribute("list", persondao.list());
    return "list";
}
```

Da dove salta fuori quel model, e dove va?

Intendiamo con model un insieme di attributi che verranno passati alla vista in maniera completamente automatica. Se in model aggiungiamo la chiave "list", questa viene resa disponibile alla vista (la JSP) perché la graficherà. Possiamo immaginare il model, impropriamente, come una mappa < String, Object >, senza sbagliare troppo.

E' l'equivalente di request.setAttribute nelle servlet. La variabile model di tipo Model verrà passata per convenzione alla vista, che la graficherà. E' solo uno dei tanti modi di passare dati alle viste, ma sarà sufficiente per i nostri scopi. Per approfondire raccomandiamo <https://www.baeldung.com/spring-mvc-model-model-map-model-view>, che mostra altre maniere di inviare dati alle viste e illustra le differenze principali.

22.7 @RestController e la scrittura di un primo web service REST in Spring

Spring dispone di una pletora di annotazioni, molte delle quali ne "riassumono" altre. Noi vedremo che c'è una versione specializzata di controller, pensata per la scrittura di web services REST. E' @RestController, e funziona analogamente a quanto accade con @Controller, con una differenza: non utilizza il sistema delle viste. Rivediamo l'esempio di prima, con una variante:

```
package com.generation.springcensus.controller

@RestController
public class TestController
{
    @RequestMapping("/test")
    public String test()
    {
        return "test";
    }
}
```

La modifica dell'annotazione comporta una modifica sostanziale di comportamento. Con @RestController il return è il corpo della response. La chiamata HTTP a /test restituirà la stringa "test", in formato puramente testuale, nel browser. Questo è molto utile per la scrittura di servizi REST, che sarebbe inutile scrivere in JSP.

A questo punto dobbiamo ricordarci che in REST i verbi contano. @RequestMapping, senza ulteriori attributi, accetterà qualunque verbo diretto all'URI corrispondente. Noi vogliamo che verbi diversi verso lo stesso indirizzo abbiano funzioni diverse: GET per leggere, DELETE per cancellare, e così via. Spring ci offre diversi modi per ottenerlo: possiamo aggiungere attributi di configurazione a @RequestMapping (l'attributo method: @RequestMapping(value = "/list", method = RequestMethod.GET) o utilizzare @GetMapping("/list"), indifferentemente. @GetMapping riassume la prima annotazione.

Ora abbiamo gli strumenti necessari per cominciare a scrivere un web service REST in Spring:

```
package com.generation.springcensus.controller

@Component
public class Context
{
    @Bean
    public Gson getGson()
    {
        //cosa sono? vedere sotto, nel controller
        return new Gson();
    }
    @Bean
    public PersonDAO getDAO()
    {
        //Ipotizziamo di avere un PersonDAOMySQL che crei la propria connection... cattiva idea, ma per chiarezza...
        return new PersonDAOMySQL("localhost", "root", "root", "census");
    }
}
```

```
package com.generation.springcensus.controller

@RestController
public class CensusWebController
{
    @Autowired
    PersonDAO personDAO;
    //Gson: un tool di google, free
    //che "traduce" gli oggetti Java in oggetti Javascript
    //vale a dire JSON. Abbiamo importato la libreria in maven (potete controllare pom.xml), e l'oggetto è stato creato nel context sopra.
    //Pensatelo come un "traduttore" di oggetti, automatizzato e tendenzialmente chiavi-in-mano.
    @Autowired
    Gson gson;
    public void setGson(Gson gson)
    {
        this.gson = gson;
    }
    public void setPersonDAO(PersonDAO personDAO)
    {
        this.personDAO = personDAO;
    }
    @GetMapping("/people");
    public String list()
    {
        return gson.toJson(personDAO.list());
    }
}
```

E' la forma più semplice di un web service. Un solo indirizzo, che restituisce la lista di tutte le persone, in formato JSON. Per il processo di JSONizzazione ci siamo appoggiati a un tool di google, che avevamo precedentemente collegato al path tramite Maven. Per buona misura, lo abbiamo iniettato tramite dependency injection.

Notiamo che non è l'unica via. Spring è in grado di produrre automaticamente JSON con la configurazione corretta, ma serve da esempio di dependency injection e in generale di MVC.

Abbiamo un altro problema. Abbiamo mappato GET /list, ma GET /list/ seguito da un numero? Spring ci offre scorciatoie anche per questo, che si applicano anche alle web-application:

```
...
@GetMapping("/list/{id}")
public String person(@PathVariable int id)
{
    return gson.toJson(persondao.load(id));
}
...
```

Questo processo è di *binding*. Abbiamo collegato un pezzo di indirizzo (parte finale, dopo list) a un parametro del metodo, tramite l'annotation @PathVariable. E' qualcosa di equivalente alla mappatura /Index/*, ma ci permette di specificare una struttura riconoscibile per gli indirizzi (in gergo si direbbe un *pattern*, ma non facciamo confusione coi pattern di design)

22.8 Form in Spring MVC

Sarebbe scomodo, per quanto possibile, passare tutti i parametri di una form nell'indirizzo. /Index/send/teacher@generation.com/studente@generation.com/message avrebbe poco senso. Ha senso usare uno strumento predefinito per questo scopo, vale a dire le form.

Una maniera intuitiva è quella di *risalire* agli oggetti delle servlet, che sono comunque presenti e disponibili "sotto" Spring MVC. Vediamo un esempio concreto:

```
<form method="post" action="/person/save">
    Name
    <input type="text" name="name" />
    Surname
    <input type="text" name="surname" />
    <input type="submit" value="save" />
</form>

// siamo in PersonController
@PostMapping("/person/save")
public String savePerson(HttpServletRequest request)
{
    dao.save(request.getParameter("name"), request.getParameter("surname"));
    return "saved";
}
```

Questo è molto comodo, perché ci riporta a una tecnologia che conosciamo bene e che sappiamo controllare a livello granulare, ma disponiamo di strumenti migliori, e in realtà non solo in Spring. Esiste il concetto di *mappatura a oggetto* della form.

In effetti, le form sono spesso state rappresentazioni dello stato di un oggetto, di una entity. I vari campi di testo erano associati manualmente alle proprietà dell'oggetto, che poi veniva salvato nel database. Non dovrebbe stupirci il fatto che questo sia stato, col tempo, meccanizzato. Siamo in grado di dire a Spring che "quella form corrisponde a quell'oggetto", e Spring sarà abbastanza furbo da collegare i fili al posto nostro, trasferendo i dati da casella di testo a oggetto in maniera quasi automatica.

In primis, dovremo scrivere la form in un certo modo, utilizzando dei tag JSP che verranno poi tradotti in HTML in fase di rendering della pagina, e che permetteranno a Spring di riconoscere la form come una propria entità:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<body>
    <form:form method="post"
        action="/censusspring/newperson" modelAttribute="person">
        <table>
            <tr>
                <td><form:label path="name">Name</form:label></td>
                <td><form:input path="name"/></td>
            </tr>
            <tr>
                <td><form:label path="surname">Surname</form:label></td>
                <td><form:input path="surname"/></td>
            </tr>
            <tr>
                <td><form:label path="age">Age</form:label></td>
                <td><form:input path="age"/></td>
            </tr>
            <tr>
                <td><input type="submit" value="save"/></td>
            </tr>
        </table>
    </form:form>
</body>
```

Questa form utilizza una libreria di tag con prefisso form (prima riga) per eseguire un *binding* della form a un oggetto (indicato con ModelAttribute person). I vari form:input sono tag JSP che verranno poi tradotti in HTML standard (in particolare, name e id saranno uguali all'attributo path per ogni input), e la request inviata dalla form sarà riconosciuta da Spring come corrispondente a quell'oggetto, a patto che ci siano i getter e i setter per le proprietà (name e surname in questo caso) e che l'handler dell'indirizzo (/censusspring/newperson) lo specifichi. Vediamo come:

```
@Controller
public class PersonController
{
    // quando chiamo l'indirizzo col metodo GET
    // viene caricata la form
    @GetMapping("/newperson")
    public String formNewPerson(Model model)
    {
        // passiamo un valore iniziale per l'oggetto Person
        // la JSP ne ha bisogno! Lo deve legare alla form.
        Person person = new Person();
        person.setName("");
    }
}
```

```

        person.setSurname("");
        model.put("person", new Person());
    }
    @PostMapping("/newperson")
    public String save(@ModelAttribute("person") Person person, BindingResult result)
    {
        // Il binding potrebbe dare errore.
        // In questo caso, renderizzeremo la pagina di errore.
        // non siamo riusciti a trasformare la form in un oggetto
        if (result.hasErrors())
            return "error";
        // altrimenti...
        dao.save(person);
        // il dao sarà stato iniettato altrove
        return "success";
    }
}

```

In questo codice dobbiamo focalizzarci su due elementi: l'annotation di parametro `@ModelAttribute`, che collega il parametro `Person` alla form (avvisa il controller di aspettarsi un oggetto di tipo `Person` e di nome `person` in arrivo dalla form) e il `BindingResult`, che a differenza del precedente non è obbligatorio.

L'annotation `ModelAttribute` permette di ricevere i parametri della form sotto forma di un singolo oggetto, ma il processo di conversione potrebbe non funzionare. Non funzionerebbe, ad esempio, se mettessimo una stringa al posto dell'età.

L'oggetto `BindingResult`, che è opzionale, viene generato automaticamente da Spring e contiene gli eventuali errori di conversione. Possiamo gestirli e poi decidere cosa fare dell'oggetto, ma tipicamente in caso di errore si preferisce annullare l'operazione.

Per quanto `BindingResult` non sia obbligatorio, è almeno buona pratica utilizzarlo. In alternativa avremo come minimo errori di salvataggio, e nel peggior dei casi il salvataggio di dati assurdi o violazioni del database, per quanto si tratti di ipotesi piuttosto remote.

23 - AJAX

23.1 Definizione di AJAX

AJAX non è uno standard industriale, né un prodotto, ma un modo di lavorare.

Fino ad ora abbiamo lavorato con l'idea implicita di ricaricare l'intera pagina web a ogni request. Ogni request comportava l'esecuzione di una JSP e la produzione di una response, che sarebbe stata inviata al browser, sostituendo il documento (DOM) pre-esistente.

Questo pone dei problemi di prestazioni - dovremo ricaricare la pagina per cambiare anche solo un elemento - e rende praticamente impossibili alcuni siti moderni. Immaginate di dover ricaricare la pagina che state leggendo ogni volta in cui dovete mandare un messaggio sul vostro social preferito, o di dover ricaricare l'intera pagina della posta online a ogni invio di una lettera.

Ci sono applicazioni che lavorano ancora così, in parte o interamente, ma sono minoritarie e "superate".

La soluzione moderna è AJAX, e prevede di comunicare col server "dietro le quinte". Per la precisione, è Javascript a inviare o richiedere i dati al server a seconda della necessità, e ad aggiornare la pagina di conseguenza. Una applicazione AJAX si comporta quindi in questo modo:

- viene caricata una prima pagina, che fungerà da *client* per le richieste successi
- la pagina così caricata conterrà un *motore Javascript* che effettuerà richieste HTTP verso il server. Il server risponderà non con intere pagine web ma con pacchetti di dati di vario genere (tipicamente JSON, vale a dire oggetti Javascript), che il motore Javascript utilizzerà per aggiornare la pagina se necessario.
- La meccanica con cui questo accade dipende dalla libreria Javascript utilizzata. Possiamo utilizzare Javascript liscio, ma è più comune utilizzare librerie di terze parti. Noi useremo jQuery.

Nel caso in cui la pagina da caricare per intero sia solo una si parla di SPA : Single Page Application, ma AJAX (acronimo di Asynchronous JavaScript and XML, come vedremo studiandone il funzionamento interno) non è vincolato a questa struttura. Potremmo avere più pagine per una applicazione, ciascuna con o senza *request dietro le quinte*. AJAX è, in sostanza, *Javascript che esegue richieste HTTP al posto del browser*.

Conviene ricordarlo in questo modo.

Questo non esclude una applicazione fatta da più pagine AJAX.

23.2 Un caso concreto - Stock Market

Supponiamo di voler scrivere un'applicazione per monitorare il prezzo di alcune azioni o titoli.

Il prezzo varia in maniera più o meno continua, a volte ogni secondo, spesso nel giro di minuti. Ricaricare l'intera pagina ogni minuto sarebbe allo stesso tempo eccessivo per il server (in termini di carico) e insufficiente per gli utenti (che possono scegliere di vendere o comprare nel giro di secondi). E' un ambito time-critical, in cui le informazioni di due minuti prima sono già vecchie.

Posto il fatto che non possiamo ricaricare l'intera pagina, possiamo affrontare il lavoro in questo modo:

- Creiamo una SPA con un motore Javascript che si occuperà di ricaricare solo i dati delle azioni ogni 10 secondi. Notiamo che *non* è la soluzione migliore, ma la sceglieremo a scopo didattico.
- Creiamo una servlet (o un controller Spring) che produca i dati delle azioni in formato JSON
- Il motore Javascript interupperà la servlet e aggiornerà il DOM di conseguenza, a intervalli regolari.

Cominciamo dalla servlet. Per semplicità ipotizziamo di fornire accesso in sola lettura, e di avere solo un paio di titoli da gestire. La parte Java (backend) potrebbe essere la seguente:

```
// la nostra entity: un titolo in borsa, o impropriamente una azione
public class Stock
{
    int id;
    String name;
    double value;
    // omettiamo i getter e setter..
    public void change()
    {
        // il valore di una azione cambia il 50% delle volte in cui
        // invochiamo questo metodo
        if(Math.random()>0.5)
        {
            this.value*= (1 + ((Math.random()-0.5)*0.01));
            // un cambiamento minimo a ogni "tick", in positivo o negativo, completamente casuale e senza trend
        }
    }
}

// servlet mappata a Index: riporto solo il doGet
// la servlet può produrre due cose: la single page application (una JSP standard) o il flusso di dati grezzo in formato JSON
// la stessa servlet produce sia il client che i suoi dati

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    // ipotizzo di avere uno stockDAO
    // ipotizzo di avere un componente gson visto in precedenza
    String cmd = request.getParameter("cmd");
    if(cmd==null) cmd = "";
    switch(cmd)
    {
        case "data":
        {
            // simulo il cambiamento
            for(Stock s:stockdao.list()) s.change();
            //invio al client in forma di JSON.
            //questi dati non saranno letti dal browser, ma da Javascript! response.getWriter().append(gson.toJson(stockdao.list()));
        }
        default:
        {
            request.getRequestDispatcher("client.jsp").forward(request,response);
        }
    }
}
```

Questa servlet è emblematica. /Index genererà una pagina web, e sarà richiesta e gestita dal browser. /Index?cmd=data genererà un file JSON con i dati delle stock (id, name, value), che verrà gestito da Javascript. A brevissimo vedremo come.

23.3 Il cliente Javascript di Stock Market

Il client Javascript verrà "nascosto" dentro una JSP, ma non è obbligatorio. Sarebbe sufficiente un qualunque file HTML, o in generale un qualunque sistema (ad esempio NodeJS) in grado di eseguire Javascript e con gli oggetti giusti (XMLHttpRequest, ad esempio), per fare richieste HTTP. Nel nostro caso, la abbiamo nascosta dentro una JSP fornita da una servlet col consueto meccanismo di attributi e viste.

Per comodità, sceglieremo di costruire una "replica" lato client di quello che esiste lato server. Produrremo entità "Stock", per quanto in Javascript le classi non esistano davvero, munite di un metodo render. Una stock sarà in grado di renderizzare se stessa, e a ogni ricaricamento dei dati (in formato JSON), le stock si re-renderizzeranno per mostrare il proprio stato aggiornato.

Il codice Javascript è il seguente:

```
class Stock
{
    constructor(id, name, value)
    {
        this.id = id;
        this.name = name;
        this.value = value;
    }
    render()
    {
        return $("#stocktemplate")
            .html()
            .replace("[id]",this.id)
            .replace("[name]",this.name)
            .replace("[value]",this.value);
    }
}
```

Stiamo facendo ovviamente uso di jQuery (e ne faremo uso anche per caricare i dati Json) dalla servlet, e di un "trucco" abbastanza comodo. Abbiamo nascosto il template di uno stock dentro un tag HTML pensato per questo scopo, il tag template, in un'altra parte della pagina:

```
<template id="stocktemplate">
    <div>
        [name] - [value] $euro;
    </div>
</template>
```

Il template non viene renderizzato quando la pagina viene caricata, ma funge da "scorta", o da "modello" (template, appunto) per ricavare altri pezzi di HTML in seguito. In questo caso Javascript lo userà per avere un modello in cui sostituire i segnaposti con le variabili. In questo caso la entità contiene in sè una "parte" della vista, ma non è un problema, nè un peccato troppo grande contro MVC. Queste sono entities lato client, pensate per essere renderizzate e mostrate a schermo. Tipicamente avranno una sola rappresentazione. Vedremo in seguito che potranno contenere anche "pezzettini" di controller, che permetteranno di renderle quasi dei "componenti" indipendenti. Ma come scriveremo in controller principale della pagina?

Un controller in una applicazione Ajax non è troppo dissimile da uno di una applicazione non Ajax. Ha mansioni abbastanza simili, sincronizza vista e modello, o utilizzando la definizione formale "riceve i comandi e li trasforma in modifiche a modello e vista". La differenza nel nostro caso è la presenza di request http: il controller lato client si coordina col controller lato server (la servlet) per ricaricare i dati, e successivamente anche per le operazioni di scrittura.

Una versione elementare potrebbe essere la seguente:

```
var controller =
{
    // lista delle stocks
    // verrà caricata dal server
    stocks:[],
    reload:function()
    {
        //il cuore di tutto. Il controller ricaricherà periodicamente
        //tutti i dati
        $.getJSON("Index?cmd=data", function(response)
        {
            // $ (jQuery) ha eseguito una chiamata HTTP Get
            // verso Index?cmd=data, vale a dire, alla mappatura che produce
            // un vettore di oggetti Javascript anonimi (JSON)
            // la chiamata è riuscita e abbiamo avuto una response
            // parametro response
            // che è stato passato alla function anonima (callback)
            // che viene eseguita alla ricezione della response
            // questo per dire che ora "response" è un vettore di JSON
            // che Javascript può scorrere in maniera naturale.
            // ripuliamo i vecchi oggetti azione:
        })
    }
}
```

```

controller.stocks = [];
// e creiamone di nuovi.
// response[i] è lo STATO dello stock i-esimo che arriva dal server
// lo usiamo per creare la stessa entità lato client.
for(var i=0; i <response.length;i++)
{
    controller.stocks.push(new Stock(response[i].id, response[i].name, response[i].value));
}
// e ora stampiamo!
// accumulando il render di tutte le azioni, una per una, in una stringa
var res = "";
for(var i=0; i<controller.stocks.length; i++)
    res+=controller.stocks[i].render();
// e stampandole in un div con id "stocklist"
$("#stocklist").html(res);
// ripianifichiamo l'esecuzione di reload fra 10 secondi:
setTimeout(controller.reload, 10000);
});
}

```

Alcuni punti da ricordare:

- Con questa metodica (getJSON, metodo di jQuery, usato per fare Ajax), la response viene interpretata come formata da oggetti JSON. E' per questo che abbiamo potuto scrivere response.length. Javascript vede la response letteralmente come un vettore di JSON.
- La servlet (Java) invia lo *stato* degli oggetti Stock, ma per Javascript sono oggetti anonimi e *senza metodi*. Per "appiccicare" loro il render abbiamo dovuto passare lo stato al costruttore per ricreare oggetti Stock Javascript completi, con metodi e affini.
- Niente di tutto questo accadrà se prima non invochiamo reload almeno una volta. Questo si può fare in \$(document).ready.

Il lavoro finale, mal graficato ma funzionante, potrebbe essere il seguente:

```

<html>
    <head>
        <title> Stock Market Javascript Client </title>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
    </head>
    <script>
        // model:
        class Stock
        {
            constructor(id, name, value)
            {
                this.id = id;
                this.name = name;
                this.value = value;
            }

            render()
            {
                return $("#stocktemplate")
                    .html()
                    .replace("[id]",this.id)
                    .replace("[name]",this.name)
                    .replace("[value]",this.value);
            }
        }

        // controller
        var controller =
        {
            // lista delle stocks
            // verrà caricata dal server
            stocks:[],
            reload:function()
            {
                //il cuore di tutto. Il controller ricaricherà periodicamente
                //tutti i dati
                $.getJSON("Index?cmd=data", function(response)
                {
                    // $ (jQuery) ha eseguito una chiamata HTTP Get
                    // verso Index?cmd=data, vale a dire, alla mappatura che produce
                    // un vettore di oggetti Javascript anonimi (JSON)
                    // la chiamata è riuscita e abbiamo avuto una response
                    // parametro response
                    // che è stato passato alla function anonima (callback)
                    // che viene eseguita alla ricezione della response

                    // questo per dire che ora "response" è un vettore di JSON
                    // che Javascript può scorrere in maniera naturale.
                });
            }
        };
    </script>

```

```

// ripuliamo i vecchi oggetti azione:
controller.stocks = [];
// e creiamone di nuovi.
// response[i] è lo STATO dello stock i-esimo che arriva dal server
// ha i campi id, name e value perchè questi campi sono presenti
// sulle entità lato server (Java)
// lo usiamo per creare la stessa entità lato client.
for(var i=0; i < response.length;i++)
{
    controller.stocks.push
    (
        new Stock
        (
            response[i].id,
            response[i].name,
            response[i].value
        )
    );
}
// e ora stampiamo!
// accumulando il render di tutte le azioni, una per una, in una stringa
var res = "";
for(var i=0; i < controller.stocks.length; i++)
    res+=controller.stocks[i].render();

// e stampandole in un div con id "stocklist"
$("#stocklist").html(res);

// ripianifichiamo l'esecuzione di reload fra 10 secondi:
setTimeout(controller.reload, 10000);
});

}

// Inizializzazione:
$(document).ready(function()
{
    // caricamento iniziale delle azioni dal server.
    // verrà poi ripetuto ogni 10 secondi.
    controller.reload();
});

</script>
<body>

<h1> Stock list </h1>
<div id="stocklist">
    <!--
        vista: le azioni verranno visualizzate qui
        stampate dopo ogni controller.reload
        quindi dopo ogni chiamate agetJSON
    -->
</div>
<!-- template per le classi Stock di Javascript -->
<template id="stocktemplate">
    <div>
        [name] - [value] $euro;
    </div>
</template>
</body>
</html>

```

Lo studente attento potrebbe chiedersi: perchè mantenere in memoria degli oggetti (controller.stocks) che vengono solo usati per essere renderizzati una volta?

Questo esempio è didattico. Nella pratica non ci vengono reinviata tutte le azioni ogni volta (sarebbe uno spreco di banda, di tempo, di hard disk, in generale di risorse), ma solo quelle che vengono cambiate. A tale scopo ci conviene mantenere una lista di azioni lato client, e aggiornare solo quelle che cambiano. Su quella lista potremo inoltre fare operazioni di ricerca, di raggruppamento, statistiche ecc...

Nella pratica in Ajax si cerca di inviare solo ciò che è cambiato, o che serve strettamente inviare. Non è neanche detto che si invii JSON. Lo vedremo a breve con un esempio di scrittura AJAX: l'acquisto di un'azione.

23.4 Acquisto di una azione - AJAX senza JSON

Cominciamo a predisporre il server perchè possa registrare l'acquisto di una azione. Modifichiamo semplicemente il metodo doGet della servlet: gli oggetti sottostanti vengono lasciati come esercizio allo studente.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
{
    // ipotizzo di avere uno stockDAO
    // ipotizzo di avere un componente gson visto in precedenza
    String cmd = request.getParameter("cmd");
    if(cmd==null) cmd = "";
    switch(cmd)
    {
        case "data":
        {
            // simulo il cambiamento
            for(Stock s:stockdao.list()) s.change();
            //invio al client in forma di JSON.
            //questi dati non saranno letti dal browser, ma da Javascript! response.getWriter().append(gson.toJson(stockdao.list()));
        }
        case "buy":
        {
            // mi arrivano l'id di una azione
            // per semplicità immaginiamo di comprare una per volta
            int id = Integer.parseInt(request.getParameter("id"));
            // orderdao... un esercizio di ricostruzione
            // l'ordine viene creato a partire da uno stock e una quantità
            // a scopo didattico non registriamo l'acquirente
            boolean saved = orderdao.save(new Order(stockdao.load(id), 1));
            response.getWriter().append(saved ? "OK": "KO");
        }
        break;
        default:
        {
            request.getRequestDispatcher("client.jsp").forward(request,response);
        }
    }
}
```

A partire da questo controller, navigando a Index?cmd=buy&id=1 verrà acquistato un titolo dello stock con id=1. Se l'acquisto riesce, produrrà la stringa "OK", altrimenti "KO". Non esattamente due grandi pagine web, ma abbastanza per Javascript. Andiamo a modificare il nostro client, partendo ora dal template del singolo Stock:

```
<template id="stocktemplate">
    <div onclick="controller.buy([id])">
        [name] - [value] $euro;
    </div>
</template>
```

Ogni entity renderizzerà questo template e genererà un gestore di evento: un richiamo a una funzione, ancora da scrivere, di nome controller.buy, a cui verrà passato l'id dello stock stesso. In effetti ora ogni stock ha il proprio piccolo model, la propria piccola vista e il proprio piccolo "controller", che si ricollega a quello più grande. In effetti ogni oggetto di "classe" Stock in Javascript è quasi un *componente* autonomo, con un proprio stato, un proprio comportamento e un proprio aspetto, che gli permette di lavorare in maniera QUASI indipendente. In questo caso particolare abbiamo preferito ricollegarci al controller principale, ma solo per comodità.

Questo modo di lavorare è molto popolare in framework più recenti, come React JS, e vale la pena di vederlo in azione.
Resta da scrivere questo fantomatico "buy":

```
var controller =
{
    // omitto il resto, che è invariato
    buy:function(id)
    {
        // sono arrivato qui perchè l'utente ha cliccato su una azione
        // ed è stato attivato l'evento onclick definito nel template
        // faccio una chiamata al server: è lui ad avere il db dopo tutto
        // e dichiaro di volere acquistare un titolo di quella determinata azione
        // non mi sarà restituito JSON, ma OK o KO, a seconda dell'esito
        // non uso, quindi, .getJSON, ma semplicemente .get
        // usando sempre jQuery per fare ajax
        $.get("Index?cmd=buy&id==" + id, function(response)
        {
            //response a questo giro non è un json, ma un testo, una stringa
            if(response=='OK')
                alert("Success!");
            else
```

```

        alert("Unable to buy!");
    });
}

```

E abbiamo finito con questo esempio minimo. Il resto è identico, non dobbiamo cambiare niente. Notiamo che avremmo potuto usare anche \$.post, e in generale variare verbo. AJAX non pone vincoli sul formato della response, sul verbo da usare e neanche sulla libreria. Noi abbiamo usato jQuery, ma ci sono librerie AJAX di vario tipo, ed è possibile usare anche Javascript liscio (oggetto XMLHttpRequest, la forma originale di AJAX).

Notiamo che da principio Ajax era legato ad XML, ma nella pratica questa limitazione è decaduta da tempo, e conviene non soffermarsi troppo sull'acronimo, salvo che sulla prima lettera: la A.

La A di AJAX sta per Asynchronous, asincrona.

Questo significa che, dopo avere eseguito \$.get, o \$.getJSON, il programma non si ferma ad aspettare, ma va avanti. Alla consegna della response (potrebbero volerci un paio di secondi, a volte), verrà eseguita la callback specificata come secondo parametro del \$.get. Noi *dobbiamo aspettare* di avere i dati per poterli lavorare: lo studente attento avrà notato che la stampa avviene *dentro* la callback di JSON. Non avremmo potuto fare qualcosa di questo genere:

```

$.getJSON("Index?cmd=data", function(...){});
var res = "";
for(var i=0; i < controller.stocks.length; i++)
    res+=controller.stocks[i].render();
$("#stocklist").html(res);

```

Dopo \$.getJSON il programma va avanti, non aspetta il ritorno. Provare a stampare subito le stocks produrrà una lista vuota. Bisogna attendere il ritorno della request http per avere i dati. Questo può creare, nella pratica, dei problemi di *sincronizzazione*. Framework più avanzati risolvono il problema con l'aggiornamento automatico delle viste al ricaricamento del model (Angular, ad esempio), ma esulano da questo corso. Lo studente prenda solo nota: prima di lavorare su qualcosa dobbiamo essere sicuri di averlo, e in questo caso stiamo aspettando la risposta del server.