MODULO 2
PARTE 6

Approfondimenti di PHP: Object-Oriented PHP e applicazioni basate su PHP

Object-Oriented Programming - I

Vi ricordate...? Introducendo la classe *XMLHttpRequest* (AJAX) abbiamo detto che:

- una **classe** è un'astrazione, un concetto, un modello (template), un tipo di dato, ... che definisce
  - la **struttura**, cioè le **proprietà**
  - il **comportamento**, cioè le **funzioni** (metodi)

  di un'insieme di **istanze** (oggetti), cioè entità concrete
- ogni classe è dotata di un **costruttore**, cioè un metodo (speciale) che, quando viene invocato (mediante la keyword *new*), **crea una nuova istanza** della classe e assegna alle proprietà dei valori specifici

Object-Oriented Programming - II

- l'insieme dei **metodi** (pubblici) definiti da una classe (e che possono essere invocate sulle istanze di quella classe) costituiscono le **API** (*Application Programming Interface*) della classe stessa

Introduciamo adesso alcuni ulteriori concetti dell'*Object-Oriented Programming*, per poi vederne l'applicazione all'interno della programmazione server-side con PHP

↓

ultimo tratto del *fil rouge*: finora abbiamo utilizzato classi (ed API) definite da altri (es: XMLHttpRequest, GMaps, ...), adesso **definiamo noi una classe**

↓

scopo primncipale: rendere **modulare** un'applicazione
⇒ rendere **riusabile** (disponibile ad altri) una certa funzionalità

OOP: Interfaccia e implementazione - I

Quando definisco una **classe** (**tipo**) ne definisco:

- l'**interfaccia** = la "vista esterna" = il comportamento = l'insieme di operazioni (funzioni, metodi) che le sue istanze potranno fare, cioè... le **API**!
- l'**implementazione** = la "vista interna" = la definizione dei meccanismi che realizzano le operazioni definite nell'interfaccia

Per esempio, in una biblioteca consideriamo il **bibliotecario**:

- quali **servizi** (**operazioni**) offre al pubblico?

prestito(libro)
restituzione(libro)
prenotazione(libro)

} questi servizi (metodi, funzioni) sono **accessibili al pubblico**

= interfaccia (API)

OOP: Interfaccia e implementazione - II

- come li **implementa** (realizza)?

prestito(libro) → procedura per trovare il libro, prenderlo, darlo all'utente, registrare il prestito sulla scheda, ...

} queste procedure sono "segrete", **non sono visibili al pubblico**

= implementazione

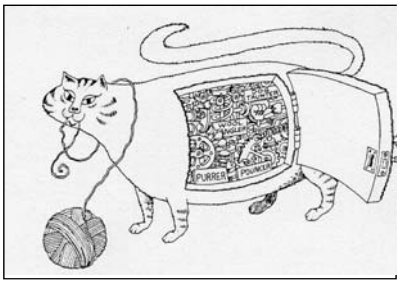
⇒ l'**interfaccia** definisce il **comportamento** di un oggetto (**API**): nell'es. le operazioni di *prestito(libro)*, *restituzione(libro)*, *prenotazione(libro)* definiscono il comportamento dei bibliotecari (cioè di tutte le istanze della classe Bibliotecario)

Le **operazioni** sono generalmente eseguite dalle **istanze** [salvo l'eccezione dei metodi statici: vedi slides precedenti...]

Per chiedere ad un oggetto l'esecuzione di un operazione si **invoca** il corrispondente **metodo** (attraverso una notazione apposita, per es. la ***dot notation***)

OOP: *information hiding*

Incapsulamento (*information hiding*)



L'incapsulamento è il principio secondo cui la struttura interna, il funzionamento interno, di un oggetto **non deve essere visibile** all'esterno

L'incapsulamento (o *information hiding*) è il processo che **nasconde** quei dettagli, relativi al funzionamento di un oggetto, che non costituiscono le sue caratteristiche essenziali e distintive [Booch, p. 46]

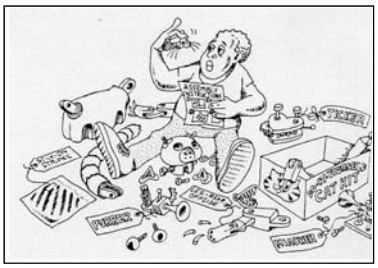
⇒ ogni oggetto è costituito da 2 parti:

- l'*interfaccia* (vista "esterna") → visibile
- l'*implementazione* (vista "interna") → nascosta

OOP: modularità

Modularità

La modularità consiste nella suddivisione di un sistema in una serie di **componenti indipendenti**, che interagiscono tra loro per ottenere il risultato



⇒ ogni oggetto (modulo) è **specializzato** in un certo compito: tutti gli altri oggetti (moduli) possono rivolgersi a lui (inviandogli un messaggio) se hanno bisogno dei suoi servizi (in cui lui è specializzato)

Per esempio, pensate a com'è organizzata la nostra società, attraverso la **modularizzazione delle funzioni** e la **collaborazione** tra gli individui, ognuno specializzato in un compito specifico: se stiamo male ci rivolgiamo ad un medico, se si è rotta l'auto ad un meccanico, per la spesa a verdurieri, macellai, ecc...,

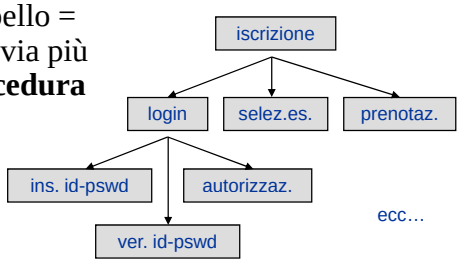
Paradigma procedurale vs OOP - I

Metodo classico di software design = *top-down functional (structured) design* = scomposizione gerarchica funzionale

↓

paradigma procedurale: procedura (algoritmo) = sequenza di passi per raggiungere il risultato

Pes es. iscrizione ad un appello = scomposizione in passi via via più semplici, elementari = **procedura** per iscriversi ad un appello



PROGRAMMI = PROCEDURE + STRUTTURE-DATI

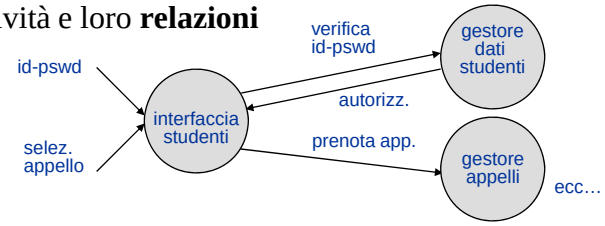
Paradigma procedurale vs OOP - II

Metodo alternativo = ***object-oriented design*** = si parte dagli oggetti, non dalle funzionalità!

↓

paradigma ad oggetti: oggetti che interagiscono tra loro scambiandosi dei messaggi = collaborazione per raggiungere il risultato

Pes es. iscrizione ad un appello = **entità** coinvolte nell'attività e loro **relazioni**



PROGRAMMI = OGGETTI (DATI + PROCEDURE) **+ COLLABORAZIONE (MESSAGGI)**

OOP: vantaggi

Alcuni **vantaggi** fondamentali dell'approccio *Object-Oriented*:

- Migliore organizzazione, strutturazione del software
- Maggiore possibilità di riuso
- Maggiore leggibilità
- Maggiore robustezza
- Estensione, modifica e manutenzione più semplici
- Migliore gestione del team di lavoro

Nella sua visione più semplice, l'approccio OO è una forma di **modularizzazione** del software... in qualche modo un'estensione della tecnica di inclusione di file esterni, usata in molti linguaggi di scripting (vedi `include(file)` di PHP)

Parentesi: Open API vs Open Source

(

L'introduzione della **distinzione interfaccia/implementazione** ci consente di capire meglio il significato di **Open API** (soprattutto in relazione con **Open Source**...):

API = interfaccia

Source = tutto il codice sorgente (interfaccia + implementazione)

⇒ **Open Source**: posso **modificare** il programma

Open API posso solo **utilizzarlo** = invocare i metodi (funzioni) offerti dalla sua interfaccia pubblica (API)

)

OOP con PHP - I

La possibilità di definire classi e creare istanze era già presente nelle versioni precedenti di PHP (rif. ver. 4), ma con **PHP 5** vengono introdotte alcune novità che rendono più sensato l'utilizzo di un'approccio OO nella programmazione con PHP

La principale innovazione è l'introduzione degli ***access modifiers***

I due principali *access modifiers* sono ***public*** e ***private***:

- *public* serve a definire "pubblica" l'accessibilità dei metodi (funzioni) che rappresentano il comportamento dell'oggetto e fanno parte della sua *interfaccia*
- *private* serve a definire "private" le proprietà dell'oggetto, proteggendo quindi i dati interni (la sua *implementazione*)

Gli *access modifier* ci permettono di implementare il **principio dell'incapsulamento** (*informtion hiding*)

OOP con PHP - II

Vediamo un **esempio**: creiamo una **classe** che gestisce la visualizzazione di una **galleria fotografica** a partire da un **elenco di file** in formato grafico, contenuti in una **cartella**:

Nell'**implementazione** della classe ci saranno:

- **strutture dati** (variabili) → le variabili dichiarate a questo livello (all'interno della classe, ma fuori da ogni suo metodo/funzione) si chiamano ***variabili d'istanza*** e rappresentano le **proprietà** della classe stessa

  Ma allora perché si chiamano "variabili d'istanza"?!?

  La **classe** definisce quali sono (es: *nome*), **tutte le istanze** della classe ne avranno una loro copia, "riempita" con il valore opportuno (es: *nome="Luca"*, *nome = "Marco"*, ...)
- un **costruttore**
- **metodi (funzioni)**, che a loro volta saranno accessibili pubblicamente (**interfaccia**), ma il loro funzionamento interno sarà "nascosto" (**implementazione**)

Esempio: definire una classe - I

Photogallery.php [commentiamo le parti di codice rilevanti…]

la keyword *class* indica che stiamo definendo una classe

```
class Photogallery {
  ...
}
```

le parentesi graffe racchiudono la definizione della classe

(1) **Proprietà** (variabili d'istanza):

- il contenuto di una cartella (elenco di nomi di file):

```
private $elencoFoto = array();
private $types = array("jpg", "jpeg", ...);
```

`$types` contiene i tipi (le estensioni) ammissibili

NB: dichiariamo ***private*** le variabili d'istanza per rispettare **il principio dell'*information hiding***: "le mie strutture dati le gestisco io; se ne hai bisogno me le chiedi, utilizzando degli appositi metodi pubblici (***set*** e ***get***)"; se non le dichiarassimo *private*, queste variabili sarebbero accessibili anche dall'esterno (cioè istanze di altre classi potrebbero accedervi impunemente!)

Esempio: definire una classe - II

(2) **Costruttore**: ha il compito di **creare nuove istanze** della classe, **inizializzando le variabili d'istanza** (cioè assegnando valori specifici alle proprietà)

NB: in PHP una classe può avere un solo costruttore!

il **costruttore** deve essere **pubblico** (NB: la keyword è opzionale: lo è comunque!)

```
public function __construct( $directory ) {
```

- il costruttore è una funzione (metodo) – anche se "speciale" ⇒ uso la keyword *function*
- si deve chiamare *__construct* e viene invocato "magicamente" con la keyword *new* (`new Photogallery(…)`)

parametro (formale) di tipo stringa: conterrà il nome di una cartella

Esempio: definire una classe - III

```
while($f = readdir($d)) {
```

scorro l'elenco dei file nella cartella...

`readdir` prende come argomento un *handle* di cartella e restituisce il nome del file "corrente" (stringa); praticamente, scorre l'elenco dei file: legge il nome del primo file da `$d` e lo mette in `$f`, poi sposta il puntatore sul secondo, ecc... finché il puntatore non viene spostato su *EOF* (*End Of File*) → a `$f` viene assegnato *false* (e il valore dell'assegnamento stesso diventa *false*)

```
 if (is_file("$directory/$f")) {
  $label = $this->creaLabel($f);
  $this->elencoFoto[$f] = $label;
 }
}
```

per ogni file:
- creo un'etichetta
- inserisco la coppia *<file, etichetta>* in *elencoFoto*

per ogni file, controllo che `$f` sia effettivamente il nome di un file (`is_file` funziona in modo analogo a `is_dir`), invoco il metodo `creaLabel($f)` e poi inserisco l'etichetta così generata nell'**array associativo** `elencoFoto` (chiavi=nomi dei file, valori=label)

Esempio: definire una classe - IV

1) Cos'è `->` ?
2) Cos'è `$this`?
3) Cosa fa `creaLabel($f)`?

1) Cos'è `->` ?

`->` equivale alla *dot notation*! (in PHP il . è usato per la concatenazione di stringhe...)

2) Cos'è `$this`?

`$this` indica un'istanza della classe stessa ⇒

- `$label = $this->creaLabel($f);`
  assegna a `$label` il risultato dell'invocazione del metodo (funzione) `creaLabel(...)` su un'istanza di *Photogallery*
- `$this->elencoFoto[$f] = $label;`
  fa riferimento alla variabile d'istanza `elencoFoto`

NB1: in PHP (5) l'uso di `$this` per far riferimento alle variabili d'istanza è obbligatorio!

NB2: i nomi delle variabili (d'istanza), quando usate insieme a `$this`, vanno **senza $**!!!

Esempio: definire una classe - V

3) Cosa fa `creaLabel($f)`?

Nella classe *Photogallery*, dobbiamo ancora definire (oltre a variabili d'istanza e costruttore) i metodi (funzioni)

(3) **Metodi** (funzioni):

```
private function creaLabel($nome) {
  $label = substr($nome, 0, strrpos($nome,"."));
  return $label;
}
```

NB: questo metodo non appartiene all'interfaccia della classe! E' un metodo interno, invisibile dall'esterno... infatti:
`$this->creaLabel($f)`

⇒ `elencoFoto`

nomefile	*label*
mare_1.jpg	mare_1
mare_2.jpg	mare_2

`strrpos(stringa,car)`
restituisce la posizione (numerica) dell'ultima occorrenza di *car* in *stringa*
es: `strrpos("anna.goy", ".")` → 4

`substr(stringa, inizio, lungh)`
restituisce la porzione di *stringa*, composta da *lungh* caratteri, a partire da *inizio* (NB: *lungh* è opzionale: se assente ⇒ fino alla fine)
es: `substr("annagoy", 0, 4)` → anna

Praticamente, per creare la label, prendiamo il nome del file senza l'estensione!

Esempio: definire una classe - VI

Introducendo la variabile d'istanza **`elencoFoto`** l'abbiamo dichiarata *private* per rispettare il principio dell'*information hiding*: "le mie strutture dati le gestisco io; se ne hai bisogno me le chiedi, utilizzando degli appositi metodi pubblici (*set* e *get*)"
→ ma cosa sono questi metodi ***set*** e ***get***?!?

Per ogni **variabile d'istanza** (privata), è buona norma avere due metodi pubblici:

- ***set*** (per assegnare alla variabile un valore)
- ***get*** (per leggere il valore della variabile)

```
public void setVar(par) {
    var = par;
}
public TipoVar getVar() {
    return var;
}
```


Esempio: definire una classe - VII

→ in questo modo la variabile è **accessibile** (visibile) **solo all'interno delle istanze della classe**: per assegnarle un nuovo valore o leggere il suo valore corrente dall'**esterno** si utilizzano i metodi *set/get*, che, in quanto **pubblici**, appartengono all'**interfaccia** dell'oggetto

```
public function getElencoFoto() {
   return $this->elencoFoto;
}
public function setElencoFoto($ef) {
  $this->elencoFoto = $ef;
}
```

questi metodi sono pubblici perché devono appartenere all'interfaccia della classe (possono essere invocati dall'esterno...)

Vediamo adesso gli **altri metodi pubblici** di *Photogallery.php*; per esempio, una **funzione** (**metodo**) che elimina da *elencoFoto* tutti gli elementi che non sono immagini (cioè con estensione non inclusa nell'array *types*)

Esempio: definire una classe - VIII

```
public function imagesOnly() {
 $ext = "";
 foreach ($this->elencoFoto as $key => $val) {
  $ext = strtolower(substr($key,(strrpos($key, ".")+1)));
  if(!in_array($ext, $this->types)) {
   unset($this->elencoFoto[$key]);
  }
 }
}
```

elimina da *elencoFoto* le coppie *<file, etichetta>* in cui il file NON è un'immagine (guarda l'estensione!)

Esempio: usare una classe - I

Una **classe**, in se stessa, è inutile: il codice scritto finora in *Photogallery.php* non visualizza niente sulla pagina web! Questo perché la **classe** è solo un **modello** di **oggetti** (**istanze** della classe appunto) che devono essere ancora creati

↓

Per **creare un'istanza** della classe *Photogallery* (cioè per creare una galleria fotografica) scriviamo uno script PHP, all'interno di una pagina web, che:

- include la classe nella pagina (richiamando il file *Photogallery.php*)
- crea un'istanza di *Photogallery*,
- apre una cartella
- aggiunge i file in essa contenuti alla variabile `elencoFoto`
- visualizza il contenuto di tale variabile sulla pagina

Esempio: usare una classe - II

Per creare un'istanza della classe *Photogallery* abbiamo bisogno del **nome di una cartella** che contenga le immagini (rif. costruttore di *Photogallery*)

↓

Creiamo un form per chiedere all'utente il nome della cartella

home.html

```
<form method="get" action="galleria.php">
  Cartella: <input type="text" name="cart">
  <input type="submit" value="ok">
</form>
```

galleria.php

```
$cartella = $_GET["cart"];
```
→ leggo la cartella da caricare

Vantaggi...

Ma perché non inserire il codice di *Photogallery.php* direttamente in *galleria.php*?!?

Principalmente (vedi *Vantaggi dell'OOP*) perché in questo modo la nostra applicazione è più **modulare** ⇒ la classe è molto più facilmente **riusabile**!

Per **esempio**, immaginate di dover inserire una galleria fotografica (caricata da una cartella) nel vostro sito: scrivete la vostra pagina web (sulla falsariga di *galleria.php*) **includendo e utilizzando direttamente** *Photogallery.php*

se il codice di *Photogallery.php* fosse scritto direttamente in *galleria.php* dovreste andare a "dissezionare" il codice per estrarre le parti che vi servono...!

Il **manuale** su PHP Object-Oriented di *www.php.net*:

http://us.php.net/manual/en/oop5.intro.php

FINE PROGRAMMA PER

Programmazione Web Avanzata
(a.a. 2008/09)

Esempio: upload di file - I

Usiamo il paradigma Object-Oriented per costruire un'altra classe "specializzata" e riusabile → esegue l'**upload di un file**, **salvandolo sul file system** del server e **inserendo un riferimento in un database** (sempre sul server)

[Rif: http://www.mrwebmaster.it/php/articoli/gestire-upload-tramite-form_184.html]

upload_form.html

```
<form enctype="multipart/form-data" action="upload.php"
      method="POST">
  Carica questo file: <input name="userfile" type="file">
  <input type="submit" value="carica file">
</form>
```

- l'attributo *enctype* specifica il tipo di strutturazione del contenuto utilizzato per inviare i dati al server quando il metodo prescelto è POST
- il valore di default è "application/x-www-form-urlencoded"
- per inviare **file** (o comunque dati non-ASCII o binari) occorre usare il valore "multipart/form-data", insieme al valore "file" per l'attributo *type* del tag INPUT

Esempio: upload di file - II

upload.php

```
require_once 'UploadProcessor.php';
$processor = new UploadProcessor("uploaded_files","userfile");
$ris = $processor->doUpload();
echo $ris;
```

- includo il file con la definizione della classe *UploadProcessor*
- creo un'istanza/oggetto della classe (nella variabile **`$processor`**), passandogli il nome della cartella in cui verranno salvati i file (NB deve esistere!) e il nome del campo del form
- invoco il metodo pubblico *doUpload*
- visualizzo il risultato

Esempio: upload di file - III

UploadProcessor.php

```
class UploadProcessor {
//variabili d'istanza:
 //cartella per file caricati
 private $upload_dir = "";
 //nome temporaneo del file
 private $userfile_tmp = "";
 //nome del file caricato
 private $userfile_name = "";
 //nome del campo del form
 private $form_field = "";
```

il **costruttore**:

- prende in input il nome della cartella in cui verranno salvati i file e il nome del campo del form
- **inizializza le variabili d'istanza**

```
//costruttore
 public function __construct($dir, $field) {
  $this->upload_dir = $dir;
  $this->form_field = $field;
  $this->userfile_tmp = $_FILES[$field]["tmp_name"];
  $this->userfile_name = $_FILES[$field]["name"];
 }
```

Esempio: upload di file - IV

Cos'è **`$_FILES`** ?

è una **variabile globale** di PHP che contiene le informazioni sul file caricato; è un **array associativo bidimensionale** (matrice), per cui (se *userfile* è il nome del campo del form):

`$_FILES['userfile']['name']` → nome del file
`$_FILES['userfile']['tmp_name']` → nome temporaneo del file
`$_FILES['userfile']['size']` → dimensioni ecc...

Definiamo ora il **metodo pubblico che effettua l'upload**:

```
public function doUpload() {
 $ok = "";
//connessione al database
include("connDB.php");
$conn = mysql_connect($host, $user, $pwd) or die...
mysql_select_db($db) or die...
```

NB: in *connDB.php* definisco host, user, pwd, **nome del DB** e **nome della tabella**: in questo modo, nella definizione della classe *UploadProcessor* non ci sono dati di configurazione!

Esempio: upload di file - V

```
$sql_check = "SELECT * FROM ".$tab." WHERE
                              name='".$this->userfile_name."'";
$ris_check = mysql_query($sql_check) or die...
if (mysql_num_rows($ris_check) == 0) {
  $sql = "INSERT INTO ".$tab." VALUES
                               ('".$this->userfile_name."')";
  $ris = mysql_query($sql) or die...
```

controllo se nel DB è già presente un file con quel nome: se NON è presente: (1) lo inserisco (NB assumo che la tabella abbia un solo campo)

```
  move_uploaded_file($this->userfile_tmp,
   $this->upload_dir."/".$this->userfile_name) or die...
  $ok = "File $this->userfile_name salvato con successo";
}
```

(2) salvo il file nella cartella *uploaded_files* (cioè lo sposto da una locazione temporanea alla cartella *uploaded_files*)

Esempio: upload di file - VI

NB: la funzione *move_uploaded_file* ha **due parametri**:

- il file da spostare (il cui nome è nella variabile d'istanza `$this->userfile_tmp`)
- la destinazione (`$this->upload_dir."/".$this->userfile_name."/"`, cioè – per es - *uploaded_files/pippo.pdf*)

Se il file è stato caricato correttamente, restituisce *true* e sposta il file, altrimenti restituisce *false* e non compie alcuna operazione

```
    else {
    $ok = "Il file $this->userfile_name esiste già!";
   }
   return $ok;
  } // chiusura metodo doUpload
} // chiusura classe
```

il metodo *doUpload* restituisce una **stringa** (variabile `$ok`) con un messaggio (positivo e negativo, in base a com'è andato l'upload)

Proposta di esercizio...

1. Riprendete il tagging system e definite una **classe** (es: TagCloudBuilder) **che costruisce tag clouds**

   Suggerimenti:

   - spostate le funzioni *tag_info()* e *tag_cloud()* nella classe
   - invocate *tag_cloud()* dallo script (es. in *tagging.php*)

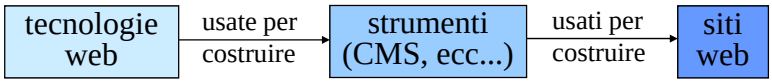
Applicazioni basate sulle tecnologie web

Le **tecnologie web** di cui abbiamo parlato sin qui

architettura client-server e HTTP, HTML, linguaggi di scripting client-side come Javascript o server-side come PHP, database, form, ecc..., tecnologie come AJAX e le Open API, ...

possono essere utilizzate **direttamente**, per costruire un sito dinamico, o (più spesso) **indirettamente**, per costruire **strumenti** che a loro volta **supportano la costruzione di siti dinamici**

Per esempio i *Content Managment Systems* (e sistemi per la costruzione e gestione di *Blog*, *Wiki*, *Social Network*, ...)



Blog - I

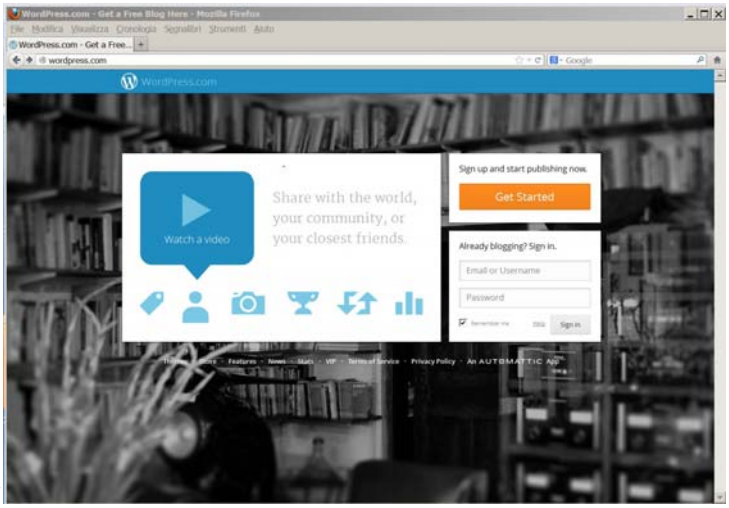
per esempio... **(1)** blog

Che cos'è un blog?

- *blog* = contrazione di *web-log* ("traccia sulla rete")
- molti blog sono **diari personali**, ma spesso diventano veri e propri **forum di discussione**
- un blog si crea e si gestisce attraverso un **programma** (applicazione web) che permette di **pubblicare contenuti** su una pagina web (anche senza conoscere HTML)
- l'**aspetto del blog** può essere personalizzato con diverse vesti grafiche (*template*)
- l'**autore** può pubblicare dei ***post*** (messaggi/articoli) e i lettori possono pubblicare i loro **commenti**
- esistono molte **applicazioni (web) per creare e gestire blog** (per es: *Blogger* - www.blogger.com, *Splinder* - www.splinder.com, *Io Bloggo* - www.iobloggo.com, *Libero* - blog.libero.it, *LiveJournal* - www.livejournal.com, o *Wordpress* - piattaforma scritta in PHP)

Blog - II

Esempio di applicazione web (servizo) per creare blog (e siti) web: http://wordpress.com/



Blog - III

Esempio di blog creato (e gestito) attraverso Wordpress: http://andreasardo.wordpress.com/

Content Management Systems - per esempio... (2) CMS

Che cos'è un CMS (Content Management System)?

Un CMS è un **software** che supporta e facilita la **gestione collaborativa di un sito web**

Nota: la nozione di CMS può avere un'accezione più ampia e riferirsi non solo a siti web, ma alla gestione di contenuti (es. documenti) in genere; qui consideriamo solo i CMS come sistemi di supporto alla gestione dei contenuti sul web

Attraverso un CMS è possibile **costruire** e **aggiornare** un sito web anche di notevoli dimensioni, senza necessariamente conoscere il linguaggio HTML o altri linguaggi di programmazione web e senza dover progettare autonomamente un apposito database per la gestione delle informazioni dinamiche ⇒ Un CMS consente

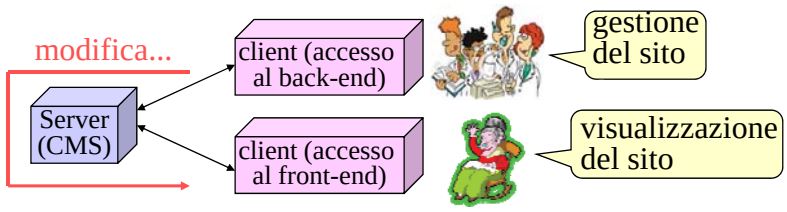
- al *web master* di costruire (ed eventualmente modificare) il ***template*** del sito
- agli *autori* di inserire (ed aggiornare) i **contenuti**

Content Management Systems - II

I CMS possono essere costruiti con tecnologie molto diverse, ma spesso utilizzano **linguaggi di scripting server-side** (PHP, ASP, JSP) e interagiscono con un **database**

⇒ un CMS è un'**applicazione web** che si installa sul **server** che ospita il sito web e si suddivide in **due parti**:

- ***back-end*** = sezione da cui si gestisce il sito; viene utilizzata dal *web master* per controllare e modificare l'**aspetto** e le **funzionalità** del sito e dagli *autori* per gestire i **contenuti**
- ***front-end*** = la parte visibile agli utenti finali (il sito web)



Content Management Systems: Joomla - I

Un esempio: Joomla! (ma ce ne sono molti altri...!)

- www.joomla.org (sito web ufficiale del progetto Joomla)
- www.joomla.it (sito web italiano di supporto del progetto Joomla)
- www.joomlaitalia.com (sito web italiano di supporto del progetto Joomla)

- CMS open source **web-based**
- **sviluppato in PHP** (e MySQL)
- distribuito gratuitamente sotto licenza GNU/GPL
- molto noto e molto usato
- supporta la gestione di molti aspetti di un sito web (aggiunta e modifica di contenuti, creazione di gallerie di immagini, realizzazione di sistemi di prenotazioni o vendita on-line)
- esiste una comunità mondiale molto attiva (utilizzatori, designers, traduttori e sostenitori) suddivisa in comunità nazionali, che forniscono supporto e documentazione agli utilizzatori del CMS

Content Management Systems: Joomla - II

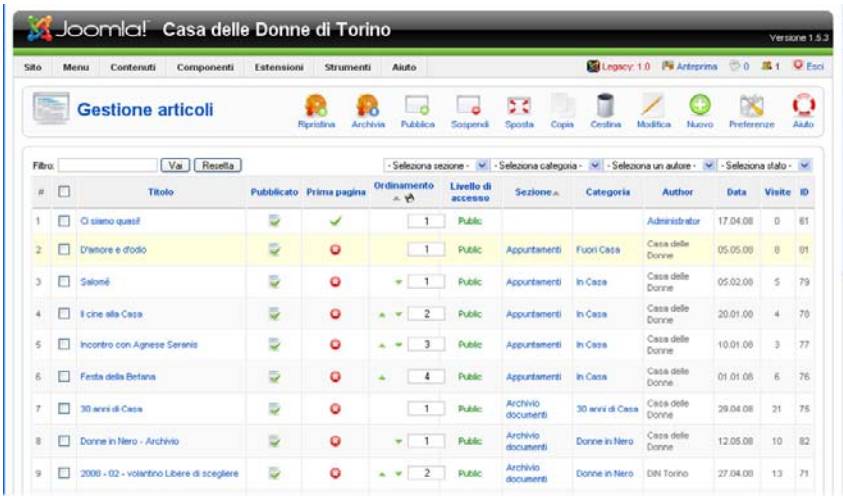
Esempio di sito creato (e gestito) con Joomla!:
http://www.casadelledonnetorino.it/

Content Management Systems: Joomla - III

Esempio: il pannello di controllo dell'amministratore del sito

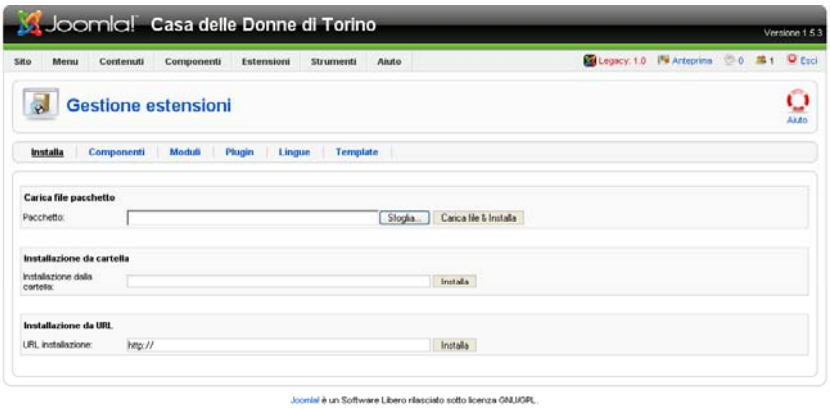
Content Management Systems: Joomla - IV

Esempio: pagina per la gestione dei contenuti



Content Management Systems: Joomla - V

Esempio: pagina per l'installazione e la gestione di estensioni aggiuntive (moduli e componenti)



Un articolo…

Un consiglio prima di chiudere:

Cosimo Baviera, *Professionalità nel web*, giugno 2009
(www.joomla.it/articoli-della-community/3338-professionalita-nel-web.html)
[PDF disponibile anche in bacheca]

A parte qualche svista, dà una visione complessiva e contiene vari spunti interessanti, per es:

- le competenze coinvolte nello sviluppo di un sito web
- il ruolo di un CMS (es. Joomla)
- nomi di dominio e ruolo degli Internet Provider nella pubblicazione di un sito
- …